28th International Conference on Theory and Applications of Satisfiability Testing

SAT 2025, August 12–15, 2025, Glasgow, Scotland

Edited by

Jeremias Berg

Jakob Nordström



LIPIcs – Vol. 341 – SAT 2025 www.dagstuhl.de/lipics

Editors

Jeremias Berg
University of Helsinki, Finland
jeremias.berg@helsinki.fi

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

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

ISBN 978-3-95977-381-2

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/978-3-95977-381-2.

Publication date
August, 2025

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.

License
This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): https://creativecommons.org/licenses/by/4.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.SAT.2025.0

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Christel Baier (TU Dresden, DE)
- Roberto Di Cosmo (Inria and Université Paris Cité, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Holger Hermanns (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)
- Daniel Král' (Leipzig University, DE and Max Planck Institute for Mathematics in the Sciences, Leipzig, DE)
- Sławomir Lasota (University of Warsaw, PL)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN – Chair)
- Chih-Hao Luke Ong (Nanyang Technological University, SG)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Pierre Senellart (ENS, Université PSL, Paris, France)
- Alexandra Silva (Cornell University, Ithaca, US)

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface
Jeremias Berg and Jakob Nordström 0:ix

Awards
..... 0:xi

Program Committee Members
..... 0:xiii

External Reviewers
..... 0:xv

List of Authors
..... 0:xvii

Invited Talks

Privacy-Preserving SAT Solving
Ruzica Piskac 1:1–1:2

Anytime and Exact Search for Planning Problems: How to Explore a DP-based State Transition Graph with A*, CP and LS?
Christine Solnon 2:1–2:2

Regular Papers

Problem Partitioning via Proof Prefixes
Zachary Battleman, Joseph E. Reeves, and Marijn J. H. Heule 3:1–3:18

Bit-Precise Reasoning with Parametric Bit-Vectors
Zvika Berger, Yoni Zohar, Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli 4:1–4:24

Semi-Algebraic Proof Systems for QBF
Olaf Beyersdorff, Ilario Bonacina, Kaspar Kasche, Meena Mahajan, and Luc Nicolas Spachmann 5:1–5:19

An Algebraic Approach to MaxCSP
Ilario Bonacina and Jordi Levy 6:1–6:17

Redundancy Rules for MaxSAT
Ilario Bonacina, Maria Luisa Bonet, Sam Buss, and Massimo Lauria 7:1–7:17

Certifying Projected Knowledge Compilation
Randal E. Bryant, Yong Kiam Tan, and Marijn J. H. Heule 8:1–8:22

CNFs and DNFs with Exactly k Solutions
L. Sunil Chandran, Rishikesh Gajjala, and Kuldeep S. Meel 9:1–9:15

28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025).
Editors: Jeremias Berg and Jakob Nordström
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Fine-Grained Complexity Analysis of Dependency Quantified Boolean Formulas
Che Cheng, Long-Hin Fung, Jie-Hong Roland Jiang, Friedrich Slivovsky, and Tony Tan 10:1–10:20

Better Extension Variables in DQBF via Independence
Leroy Chew and Tomáš Peitl 11:1–11:24

SAT-Metropolis: Combining Markov Chain Monte Carlo with SAT/SMT Sampling
Maja Aaslyng Dall, Raúl Pardo, Thomas Lumley, and Andrzej Wąsowski 12:1–12:18

Random Local Access for Sampling k-SAT Solutions
Dingding Dong and Nitya Mani 13:1–13:24

Learn to Unlearn
Bernhard Gstrein, Florian Pollitt, André Schidler, Mathias Fleury, and Armin Biere 14:1–14:12

RustSAT: A Library for SAT Solving in Rust
Christoph Jabs 15:1–15:13

Depth-Optimal Quantum Layout Synthesis as SAT
Anna B. Jakobsen, Anders B. Clausen, Jaco van de Pol, and Irfansha Shaik 16:1–16:17

Core-Guided Linear Programming-Based Maximum Satisfiability
George Katsirelos 17:1–17:17

Towards Practical First-Order Model Counting
Ananth K. Kidambi, Guramrit Singh, Paulius Dilkas, and Kuldeep S. Meel 18:1–18:18

An Application of SAT Solvers in Integer Programming Games
Pravesh Koirala, Aditya Shrey, and Forrest Laine 19:1–19:12

Scalable Precise Computation of Shannon Entropy
Yong Lai, Haolong Tong, Zhenghang Xu, and Minghao Yin 20:1–20:19

Improving Reduction Techniques in Pseudo-Boolean Conflict Analysis
Orestis Lomis, Jo Devriendt, Hendrik Bierlee, and Tias Guns 21:1–21:17

Enumerating All Boolean Matches
Alexander Nadel and Yogev Shalmon 22:1–22:21

Symbolic Conflict Analysis in Pseudo-Boolean Optimization
Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Rui Zhao 23:1–23:18

SAT-Based CEGAR Method for the Hamiltonian Cycle Problem Enhanced by Cut-Set Constraints
Ryoga Ohashi, Takehide Soh, Daniel Le Berre, Hidetomo Nabeshima, Mutsunori Banbara, Katsumi Inoue, and Naoyuki Tamura 24:1–24:10

QRP+Gen: A Framework for Checking Q-Resolution Proofs with Generalized Axioms
Mark Peyrer and Martina Seidl 25:1–25:10

Analyzing Reformulation Performance in Core-Guided MaxSAT Solving
André Schidler and Stefan Szeider 26:1–26:18

Streamlining Distributed SAT Solver Design
Dominik Schreiber, Niccolò Rigi-Luperti, and Armin Biere 27:1–27:23

CNOT-Optimal Clifford Synthesis as SAT
Irfansha Shaik and Jaco van de Pol 28:1–28:21

Reencoding Unique Literal Clauses
Aeacus Sheng, Joseph E. Reeves, and Marijn J. H. Heule 29:1–29:21

Bridging Language Models and Symbolic Solvers via the Model Context Protocol
Stefan Szeider 30:1–30:12

On Top-Down Pseudo-Boolean Model Counting
Suwei Yang, Yong Lai, and Kuldeep S. Meel 31:1–31:10

Efficient Certified Reasoning for Binarized Neural Networks
Jiong Yang, Yong Kiam Tan, Mate Soos, Magnus O. Myreen, and Kuldeep S. Meel 32:1–32:22

Preface

This volume contains (extended) abstracts of invited talks and full papers presented at the 28^{th} International Conference on Theory and Applications of Satisfiability Testing (SAT 2025). SAT 2025 was held August 12–15, 2025, in Glasgow, Scotland, and was hosted by the University of Glasgow. The conference was co-located with the 31^{st} International Conference on Principles and Practice of Constraint Programming (CP) and the 18^{th} International Symposium on Combinatorial Search (SoCS).

The International Conference on Theory and Applications of Satisfiability Testing (SAT) is the leading annual meeting for researchers working on all aspects of the propositional satisfiability problem broadly construed. In addition to standard propositional satisfiability, the scope of the conference also includes Boolean optimization problems using, e.g., Max-SAT and pseudo-Boolean (PB) solving, quantified Boolean formulas (QBF), satisfiability modulo theories (SMT), model counting, constraint programming (CP), and integer linear programming (ILP) for problems with clear connections to Boolean-level reasoning.

SAT 2025 welcomed original contributions addressing different aspects of SAT, interpreted in a broad sense. This included theoretical advances in areas like exact algorithms and proof complexity, practical search algorithms, knowledge compilation, implementation-level details of 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 77 submissions were received by the submission deadline. After an initial filtering of submissions that were clearly incomplete or out of scope, a total of 56 long papers, 9 short papers, and 8 tool papers were reviewed, each by at least 4 program committee members or external expert reviewers invited by members of the program committee. The review process included an author response period, during which the authors were provided an opportunity to respond to initial reviews and to specific queries posed by reviewers. The author response period was followed by an extensive discussion among the reviewers in order to reach a final decision on the list of accepted papers. In the end, 23 long papers, 4 short papers, and 3 tool papers were accepted for publication and presentation at the conference.

In addition to presentations of accepted papers, the technical program of SAT 2025 also included invited talks by eminent researchers in the area. Ruzica Piskac (Yale University), Christine Solnon (CITI, INSA Lyon / INRIA), and Sylvie Thiebaux (Australian National University and University of Toulouse) delivered invited talks on important and contemporary topics of significant interest to the community. The invited talks were joint with the two other co-located conferences.

In connection with the three conferences there were a multitude of associated events. These included the SAT/SMT/AR Summer school organized August 6–8 in St Andrews by Joan Espasa Arxer, Ian Gent, and Ruth Hoffmann and a two-day joint SAT-CP doctoral program organized by Katalin Fazekas and Mun See Chang. Additionally, nine different workshops were held concurrently on August 10 and 11:

- *The 23rd International Workshop on Satisfiability Modulo Theories*, with Sophie Tourret, and Jochen Hoenicke as PC chairs.
- *Pragmatics of SAT*, organized by Mikoláš Janota, and Aina Niemetz.
- *Explanations with Constraints and Satisfiability*, organized by Bart Bogaerts, Tias Guns, Matti Järvisalo, and Jussi Rintanen.

28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025).
Editors: Jeremias Berg and Jakob Nordström
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- *Machine Learning for Solvers and Provers*, organized by Vijay Ganesh and Stefan Szeider.
- *LLMs meet Constraint Solving*, organized by Tias Guns, Serdar Kadioglu, Stefan Szeider, and Dimos Tsouros.
- *Constraint Modelling and Reformulation*, organized by Ian Gent and María Andreína Francisco Rodríguez.
- *Model Counting, Sampling and Synthesis*, organized by Paulius Dilkas and Priyanka Golia.
- *Quantified Boolean Formulas and Beyond*, organized by Hubie Chen, Leroy Chew, Friedrich Slivovsky, and Martina Seidl.
- *Progress Towards the Holy Grail*, organized by Lars Kotthoff.

The results of the following competitions were also announced during the week:

- *SAT Competition 2025*, organized by Cayden Codel, Katalin Fazekas, Marijn Heule, and Markus Iser.
- *Pseudo-Boolean Competition 2025*, organized by Olivier Roussel.
- *Model Counting Competition 2025*, organized by Johannes Fichte, Markus Hecher, and Arijit Shaw.
- *MiniZinc Challenge*, judged by Jimmy H.M. Lee, Barry O'Sullivan, and Roland Yap.
- *XCSP Competition*, organized by Gilles Audemard, Christophe Lecoutre and Emmanuel Lonca.

We are thankful to all individuals, institutions, and organizations that contributed to making SAT 2025 a success, and extend a special thanks to the chairs and organizers of CP and SoCS for smooth collaboration. We thank all authors for submitting their work to SAT 2025, and all participants for actively participating in the conference and its affiliated events. Our deepest thanks go to all program committee members and the external expert reviewers for carefully reading the submissions, providing constructive comments, and participating in discussions that helped in selecting the strongest papers for the technical program of the conference. We are especially grateful to Daniel Le Berre, Massimo Lauria, and Supratik Chakraborty for serving on the award committee, chaired by Daniel Le Berre. Based on nominations by the program committee members, the award committee selected which papers should receive the best paper and best student paper awards.

We are thankful to the three invited speakers for accepting our invitation and enlightening the audience with their insights and perspectives, and to the organizers of all affiliated events. We express a special thanks to Marc Vinyals and Zeynep Kiziltan for serving as the workshop chairs of SAT and CP. We would also like to thank Ciaran McCreesh, who served as the local organizer of SAT 2025, as well as the entire local organizing committee that worked tirelessly in order to keep the conference running smoothly. We are grateful to the SAT Association and its board members for their invaluable advice and support in organizing the conference. We are also indebted to the EasyChair conference management system for facilitating the submission and selection of papers, as well as scheduling the final program, and the staff at LIPIcs for coordinating and assisting with the assembly of these proceedings.

Finally, we wish to thank all sponsors and supporters of SAT 2025. We gratefully acknowledge the financial support provided by the Journal of Artificial Intelligence (AIJ), Cadence, MinLP / The Optimization Firm, JP Morgan, Potassco, Amazon AWS ARG, and Synopsys, which made it possible to keep the conference accessible to as many researchers as possible, particularly to PhD students. Our sincere thanks also goes to the University of Glasgow, the Glasgow City Council, and the Glasgow Convention Bureau for their assistance in organizing the conference and their role in making SAT 2025 a success.

August 2025 Jeremias Berg & Jakob Nordström

Awards

In recognition of outstanding and impactful contributions in areas of relevance to SAT 2025, five awards were presented at the conference. Members of the program committee nominated candidates for best paper and best student paper, where in the latter category the lead author should be a student at the time of submission. Taking these nominations into consideration, the final decisions were then taken by an award committee chaired by Daniel Le Berre.

The **Best Paper Award** of SAT 2025 was conferred to "Streamlining Distributed SAT Solver Design" by Dominik Schreiber, Niccolò Rigi-Luperti and Armin Biere for *"for providing new bases for the practical development of distributed SAT solving"*. The **Best Student Paper Award** was conferred on "Enumerating All Boolean Matches" by Alexander Nadel and Yogev Shalmon for *"for providing a new elegant and efficient SAT-based solution to a problem of interest to both academia and industry"*.

The award committee also selected to highlight two runners-up for the Best Student Paper Award, namely "Fine-Grained Complexity Analysis of Dependency Quantified Boolean Formulas" by Che Cheng, Long-Hin Fung, Jie-Hong Roland Jiang, Friedrich Slivovsky and Tony Tan, *"for exploring tractable fragments and subclasses of DQBF problem"*, and "Efficient Certified Reasoning for Binarized Neural Networks" by Jiong Yang, Yong Kiam Tan, Mate Soos, Magnus O. Myreen and Kuldeep S. Meel, *"for introducing conditional cardinalities as an effective tool in Binarized Neural Network verification"*.

In 2024, the SAT Association established the ***Fahiem Bacchus PhD Award in Satisfiability***. This award distinguishes one outstanding PhD thesis from the past year 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. The 2025 Fahiem Bacchus Award was conferred at SAT 2025 on Xindi Zhang for his PhD thesis "Hybrid Algorithms for SAT and SMT and Their Applications", defended at the University of the Chinese Academy of Sciences in 2024.

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 2025 Test-of-Time Award was conferred on the paper "Effective Preprocessing in SAT Through Variable and Clause Elimination", which appeared in the proceedings of SAT 2005 and was authored by Niklas Eén and Armin Biere.

Finally, the SAT Association conferred a **Distinguished Service Award** on Hans Kleine Büning in honour of his long-lasting and foundational contributions to the series of International Conferences on Theory and Applications of Satisfiability Testing (SAT) and the SAT Association.

28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025).
Editors: Jeremias Berg and Jakob Nordström
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committee Members

Erika Abraham	RWTH Aachen University
Carlos Ansótegui	Universitat de Lleida
Olaf Beyersdorff	Friedrich Schiller University Jena
Armin Biere	University of Freiburg
Nikolaj Bjørner	Microsoft
Bart Bogaerts	KU Leuven
Ilario Bonacina	UPC Universitat Politecnica de Catalunya
Shaowei Cai	Institute of Software, Chinese Academy of Sciences
Supratik Chakraborty	IIT Bombay
Leroy Chew	University of Leeds
Katalin Fazekas	TU Wien
Johannes K. Fichte	Linköping University
Noah Fleming	Memorial University
Mathias Fleury	University of Freiburg
Priyanka Golia	Indian Institute of Technology Delhi
Marijn Heule	Carnegie Mellon University
Markus Iser	Karlsruhe Institute of Technology
Jie-Hong Roland Jiang	National Taiwan University, Taiwan
Matti Järvisalo	University of Helsinki
George Katsirelos	MIA Paris, INRAE, AgroParisTech
Daniela Kaufmann	TU Wien
Massimo Lauria	Sapienza University of Rome
Daniel Le Berre	CNRS - Université d'Artois
Jordi Levy	IIIA - CSIC
Meena Mahajan	The Institute of Mathematical Sciences, HBNI, Chennai
Stefan Mengel	CNRS, CRIL
Aina Niemetz	Stanford University
Albert Oliveras	Universitat Politècnica de Catalunya
Kristin Yvonne Rozier	Iowa State University
Andre Schidler	University of Freiburg
Tanja Schindler	University of Basel
Dominik Schreiber	Karlsruhe Institute of Technology
Roberto Sebastiani	University of Trento
Martina Seidl	Johannes Kepler University Linz
Laurent Simon	Labri, Bordeaux Institute of Technology
Friedrich Slivovsky	University of Liverpool
Takehide Soh	Information Science and Technology Center, Kobe University
Stefan Szeider	TU Wien
Marc Vinyals	University of Auckland

28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025).
Editors: Jeremias Berg and Jakob Nordström

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

External Reviewers

S. Akshay
Alexis Aurandt
Mutsunori Banbara
Christoph Berkholz
Simone Bianco
Benjamin Böhm
Carlos Cantero
Florent Capelli
Daimy Van Caudenberg
Sravanthi Chede
Cayden Codel
Alexis de Colnet
Dror Fried
Nils Froleyks
Johanna Groven
Ashutosh Gupta
Markus Hecher
Simone Heisinger
Malte Helmert
Tim Hoffmann
Clemens Hofstadler
Hannes Ihalainen
Gonzalo E. Imaz
Deniz Imrek
Christoph Jabs
Brendan Juba
Kaspar Kasche
Markus Kirchweger
Sumit Lahiri
Yong Lai
Anna Latour
Arnaud Lequen
Chunxiao Li
Emmanuel Lonca
Kumar Madhukar
Christophe Marciot
Gabriele Masina
Matthew McIlree
Neeldhara Misra
Sasank Mouli
Hidetomo Nabeshima
Jasper Nalbach
Andreas Niskanen
Andy Oertel
Davide Pastorello
Tobias Paxian
Tomáš Peitl
Mark Peyrer
Peter Pfeiffer
Tobias Philipp
Andreas Plank
Florian Pollitt
Valentin Promies
Joseph Reeves
Franz-Xaver Reichl
Agnes Schleitzer
Arijit Shaw
Anil Shukla
Gaurav Sood
Luc Spachmann
Giuseppe Spallitta
Bernardo Subercaseaux
Tony Tan
Dieter Vandesande
Jorke de Vlas
Romain Wallon
Zili Wang
Hai Xia
Suwei Yang
Emre Yolcu

28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025).
Editors: Jeremias Berg and Jakob Nordström
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Mutsunori Banbara (24)
Nagoya University, Nagoya, Japan

Clark Barrett (4)
Stanford University, CA, USA

Zachary Battleman (3)
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Zvika Berger (4)
Bar-Ilan University, Ramat Gan, Israel

Olaf Beyersdorff (5)
Friedrich Schiller University Jena, Germany

Armin Biere (14, 27)
University of Freiburg, Germany

Hendrik Bierlee (21)
DTAI, KU Leuven, Belgium

Ilario Bonacina (5, 6, 7)
UPC Universitat Politècnica de Catalunya, Barcelona, Spain

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

Randal E. Bryant (8)
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Sam Buss (7)
University of California, San Diego, CA, USA

L. Sunil Chandran (9)
Indian Institute of Science, Bengaluru, India

Che Cheng (10)
Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan

Leroy Chew (11)
TU Wien, Austria

Anders B. Clausen (16)
Department of Computer Science, Aarhus University, Denmark

Maja Aaslyng Dall (12)
IT University of Copenhagen, Denmark

Jo Devriendt (21)
Nonfiction Software, Leuven, Belgium

Paulius Dilkas (18)
University of Toronto, Canada;
Vector Institute, Toronto, Canada

Dingding Dong (13)
Harvard University, Cambridge, MA, USA

Mathias Fleury (14)
University of Freiburg, Germany

Long-Hin Fung (10)
Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan

Rishikesh Gajjala (9)
Indian Institute of Science, Bengaluru, India

Bernhard Gstrein (14)
University of Freiburg, Germany

Tias Guns (21)
DTAI, KU Leuven, Belgium

Marijn J. H. Heule (3, 8, 29)
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Katsumi Inoue (24)
National Institute of Informatics, Tokyo, Japan

Christoph Jabs (15)
University of Helsinki, Finland

Anna B. Jakobsen (16)
Department of Computer Science, Aarhus University, Denmark

Jie-Hong Roland Jiang (10)
Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan

Kaspar Kasche (5)
Friedrich Schiller University Jena, Germany

George Katsirelos (17)
Université Paris-Saclay, AgroParisTech, INRAE, UMR MIA Paris-Saclay, 91120, Palaiseau, France

Ananth K. Kidambi (18)
Indian Institute of Technology Bombay, Mumbai, India

Pravesh Koirala (19)
Department of Computer Science, Vanderbilt University, Nashville, TN, USA

28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025).
Editors: Jeremias Berg and Jakob Nordström
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Yong Lai (20, 31)
Key Laboratory of Symbolic Computation and Knowledge Engineering Ministry of Education, Jilin University, Changchun, China

Forrest Laine (19)
Department of Computer Science, Vanderbilt University, Nashville, TN, USA

Massimo Lauria (7)
Sapienza Università di Roma, Rome, Italy

Daniel Le Berre (24)
Univ. Artois, CNRS, UMR 8188 CRIL, Lens, France

Jordi Levy (6)
IIIA, CSIC, Bellatera, Spain

Orestis Lomis (21)
DTAI, KU Leuven, Belgium

Thomas Lumley (12)
University of Auckland, New Zealand

Meena Mahajan (5)
The Institute of Mathematical Sciences (A CI of Homi Bhabha National Institute), Chennai, India

Nitya Mani (13)
Massachusetts Institute of Technology, Cambridge, MA, USA

Kuldeep S. Meel (9, 18, 31, 32)
Georgia Institute of Technology, Atlanta, GA, USA;
University of Toronto, Canada

Magnus O. Myreen (32)
Chalmers University of Technology, Gothenburg, Sweden;
University of Gothenburg, Sweden

Hidetomo Nabeshima (24)
Yamanashi University, Kofu, Japan

Alexander Nadel (22)
Intel Corporation, Haifa, Israel;
Faculty of Data and Decision Sciences, Technion, Haifa, Israel

Aina Niemetz (4)
Stanford University, CA, USA

Robert Nieuwenhuis (23)
Barcelogic.com, Barcelona, Spain

Ryoga Ohashi (24)
Kobe University, Kobe, Japan

Albert Oliveras (23)
Technical University of Catalonia, Barcelona, Spain

Raúl Pardo (12)
IT University of Copenhagen, Denmark

Tomáš Peitl (11)
TU Wien, Austria

Mark Peyrer (25)
Institute for Symbolic Artificial Intelligence, Johannes Kepler University, Linz, Austria

Ruzica Piskac (1)
Yale University, New Haven, CT, USA

Florian Pollitt (14)
University of Freiburg, Germany

Mathias Preiner (4)
Stanford University, CA, USA

Joseph E. Reeves (3, 29)
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Andrew Reynolds (4)
The University of Iowa, Iowa City, IA, USA

Niccolò Rigi-Luperti (27)
Karlsruhe Institute of Technology, Germany

Enric Rodríguez-Carbonell (23)
Technical University of Catalonia, Barcelona, Spain

André Schidler (14, 26)
University of Freiburg, Germany;
TU Wien, Austria

Dominik Schreiber (27)
Karlsruhe Institute of Technology, Germany

Martina Seidl (25)
Institute for Symbolic Artificial Intelligence, Johannes Kepler University, Linz, Austria

Irfansha Shaik (16, 28)
Department of Computer Science, Aarhus University, Denmark;
Kvantify Aps, Copenhagen S, Denmark

Yogev Shalmon (22)
Intel Corporation, Haifa, Israel;
Faculty of Data and Decision Sciences, Technion, Haifa, Israel

Aeacus Sheng (29)
Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA, USA

Aditya Shrey (19)
Department of Computer Science,
Vanderbilt University, Nashville, TN, USA

Guramrit Singh (18)
Indian Institute of Technology Bombay,
Mumbai, India

Friedrich Slivovsky (10)
Department of Computer Science,
University of Liverpool, UK

Takehide Soh (24)
Kobe University, Kobe, Japan

Christine Solnon (2)
Univ Lyon, INSA Lyon, Inria, CITI, EA3720,
69621 Villeurbanne, France

Mate Soos (32)
University of Toronto, Canada

Luc Nicolas Spachmann (5)
Friedrich Schiller University Jena, Germany

Stefan Szeider (26, 30)
TU Wien, Austria

Naoyuki Tamura (24)
Kobe University, Kobe, Japan

Tony Tan (10)
Department of Computer Science,
University of Liverpool, UK

Yong Kiam Tan (8, 32)
Institute for Infocomm Research (I²R), A*STAR,
Singapore;
Nanyang Technological University, Singapore

Cesare Tinelli (4)
The University of Iowa, Iowa City, IA, USA

Haolong Tong (20)
College of Computer Science and Technology,
Jilin University, Changchun, China

Jaco van de Pol (16, 28)
Department of Computer Science,
Aarhus University, Denmark

Andrzej Wąsowski (12)
IT University of Copenhagen, Denmark

Zhenghang Xu (20)
School of Computer Science and Information
Technology, Northeast Normal University,
Changchun, China

Jiong Yang (32)
Georgia Institute of Technology,
Atlanta, GA, USA

Suwei Yang (31)
Grabtaxi Holdings, Singapore;
National University of Singapore, Singapore

Minghao Yin (20)
School of Computer Science and Information
Technology, Northeast Normal University,
Changchun, China

Rui Zhao (23)
Technical University of Catalonia,
Barcelona, Spain

Yoni Zohar (4)
Bar-Ilan University, Ramat Gan, Israel

Privacy-Preserving SAT Solving

Ruzica Piskac
Yale University, New Haven, CT, USA

Abstract

This is an extended abstract of the invited talk presented at the joint conferences "28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025)" and "31st International Conference on Principles and Practice of Constraint Programming (CP 2025)". The talk is based on a series of three papers published previously, and it provides a unified overview of their key ideas and results.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases SAT solving, Privacy-preserving reasoning, Zero-knowledge proofs, Propositional unsatisfiability

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.1

Category Invited Talk

Related Version
Full Version: `https://www.usenix.org/conference/usenixsecurity22/presentation/luo` [2]
Full Version: `https://dl.acm.org/doi/10.1145/3548606.3559373`
Full Version: `https://dl.acm.org/doi/10.1145/3576915.3616621`

Funding *Ruzica Piskac*: NSF award CNS-1562888 and CCF-2131476.

1 Extended Abstract

Formal methods offer a vast collection of techniques to analyze and ensure the correctness of software and hardware systems against a given specification. In fact, modern formal methods tools scale to industrial applications. Despite this significant success, privacy requirements are not considered in the design of these tools. For example, when using automated reasoning tools, the implicit requirement is that the formula to be proved is public. This raises an issue if the formula itself reveals information that is supposed to remain private to one party. To overcome this issue, we propose the concept of privacy-preserving automated reasoning.

We first consider the problem of privacy-preserving Boolean satisfiability [2]. In this problem, two mutually distrustful parties each provides a Boolean formula. The goal is to decide whether their conjunction is satisfiable without revealing either formula to the other party. We present an algorithm to solve this problem. Our algorithm is an oblivious variant of the classic DPLL algorithm and can be integrated with existing secure two-party computation techniques.

We next turn to the problem where one party wants to prove to another party that their program satisfies a given specification without revealing the program. We split this problem into two subproblems: (1) proving that the program can be translated into a propositional formula without revealing either the program or the formula; (2) proving that the obtained formula entails the specification. To solve the latter subproblem, we developed a zero-knowledge protocol for proving the unsatisfiability of formulas in propositional logic [1] (ZKUNSAT). Our protocol is based on a resolution proof of unsatisfiability. We encode verification of the resolution proof using polynomial equivalence checking, which enables us to use fast zero-knowledge protocols for polynomial satisfiability.


© Ruzica Piskac;
licensed under Creative Commons License CC-BY 4.0

We will also outline Ou [3], the first programming framework that provides fully automated and optimal parallelization for zero-knowledge proofs. Zero-knowledge proofs are a powerful cryptographic primitive used in many decentralized or privacy-focused applications. However, the high overhead of ZKPs can restrict their practical applicability. The backend of Ou efficiently and automatically parallelizes ZKPs by formulating program parallelization as integer linear programming problems.

We will conclude by discussing future directions towards fully automated privacy-preserving program verification.

References

1 Ning Luo, Timos Antonopoulos, William R. Harris, Ruzica Piskac, Eran Tromer, and Xiao Wang. Proving UNSAT in zero knowledge. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2203–2217. ACM, 2022. doi:10.1145/3548606.3559373.

2 Ning Luo, Samuel Judson, Timos Antonopoulos, Ruzica Piskac, and Xiao Wang. ppsat: Towards two-party private SAT solving. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 2983–3000. USENIX Association, 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/luo.

3 Yuyang Sang, Ning Luo, Samuel Judson, Ben Chaimberg, Timos Antonopoulos, Xiao Wang, Ruzica Piskac, and Zhong Shao. Ou: Automating the parallelization of zero-knowledge protocols. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pages 534–548, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3576915.3616621.

Anytime and Exact Search for Planning Problems

How to Explore a DP-based State Transition Graph with A*, CP and LS?

Christine Solnon
Univ Lyon, INSA Lyon, Inria, CITI, EA3720, 69621 Villeurbanne, France

Abstract

Many planning problems may be solved with Dynamic Programming (DP) by decomposing the problem into subproblems which are recursively solved. These decompositions induce state transition graphs which are closely related to decision diagrams [5], and where optimal solutions correspond to best paths in these graphs. A* is a well known algorithm which extends Djikstra's algorithm with heuristics for guiding the path search [4]. It is exact (provided that the heuristic function is admissible), but it is not anytime. In other words, it computes a best path but it does not output sub-optimal paths while computing it. Hence, when state transition graphs have exponential sizes, A* may run out of time or memory without producing any solution. Various anytime extensions of A* have been proposed to compute a sequence of paths of increasing quality until finding an optimal path and proving its optimality.

In this talk, we will provide an overview of these exact and anytime extensions of A*, with a more detailed focus on *Anytime Column Search* (ACS) [10], and *Iterative Memory Bounded A** (IMBA*) [6]. Both approaches iterate A* searches while bounding the number of states that are stored or expanded at each iteration. We will also show how to combine them with Local Search (LS) in order to find better paths faster, and with bounding and constraint propagation in order to prune the graph, as proposed in [3].

This will be illustrated using the Travelling Salesman Problem (TSP) as a running example. The DP formulation introduced by Bellman in [1] for the TSP has been extended to handle Time Windows (TWs) in [2], and Time Dependent (TD) cost functions in [7]. It has also been extended to Vehicle Routing Problems (VRPs) in [11] and to TD-VRPs in [8]. We will finish by presenting an experimental comparison with state-of-the-art approaches for solving the TSP with TWs on classical benchmarks and on a new benchmark which contains hard Euclidean instances located in the phase transition zone [9].

2012 ACM Subject Classification Computing methodologies → Artificial intelligence

Keywords and phrases Dynamic Programming, A*, Anytime search, Time Dependent TSP-TW

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.2

Category Invited Talk

Funding The author acknowledges the support of the French Agence Nationale de la Recherche (ANR), under grant ANR-22-CE22-0016-01 (project MAMUT).

References

1 R. Bellman. Dynamic Programming Treatment of the Travelling Salesman Problem. *Journal of the ACM (JACM)*, 9(1):61–63, 1962. doi:10.1145/321105.321111.

2 N. Christofides, A. Mingozzi, and P. Toth. State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11(2):145–164, 1981. doi:10.1002/NET.3230110207.

3 R. Fontaine, J. Dibangoye, and C. Solnon. Exact and anytime approach for solving the time dependent traveling salesman problem with time windows. *Eur. J. Oper. Res.*, 311(3):833–844, 2023. doi:10.1016/J.EJOR.2023.06.001.

4 P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2):100–107, 1968. doi:10.1109/TSSC.1968.300136.


© Christine Solnon;
licensed under Creative Commons License CC-BY 4.0



5 J. N. Hooker. Decision diagrams and dynamic programming. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013*, volume 7874 of *LNCS*, pages 94–110. Springer, 2013. doi:10.1007/978-3-642-38171-3_7.

6 L. Libralesso and F. Fontan. An anytime tree search algorithm for the 2018 ROADEF/EURO challenge glass cutting problem. *Eur. J. Oper. Res.*, 291(3):883–893, 2021. doi:10.1016/J.EJOR.2020.10.050.

7 C. Malandraki and R. B. Dial. A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem. *Eur. J. Oper. Res.*, 90(1):45–55, 1996. doi:10.1016/0377-2217(94)00299-1.

8 O. Rifki, N. Chiabaut, and C. Solnon. On the impact of spatio-temporal granularity of traffic conditions on the quality of pickup and delivery optimal tours. *Transportation Research Part E: Logistics and Transportation Review*, 142, 2020. doi:10.1016/j.tre.2020.102085.

9 O. Rifki and C. Solnon. On the phase transition of the euclidean travelling salesman problem with time windows. *J. Artif. Intell. Res.*, 82:2167–2188, 2025. doi:10.1613/JAIR.1.18334.

10 S. G. Vadlamudi, P. Gaurav, S. Aine, and P. P. Chakrabarti. Anytime Column Search. In *AI 2012: Advances in Artificial Intelligence - 25th Australasian Joint Conference*, volume 7691 of *LNCS*, pages 254–265. Springer, 2012. doi:10.1007/978-3-642-35101-3_22.

11 J. J. van Hoorn. *Dynamic Programming for Routing and Scheduling: Optimizing Sequences of Decisions*. PhD thesis, Amsterdam: Vrije Universiteit, 2016. URL: https://research.vu.nl/files/42163578/complete%20dissertation.pdf.

Problem Partitioning via Proof Prefixes

Zachary Battleman
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Joseph E. Reeves
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Marijn J. H. Heule
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Satisfiability solvers have been instrumental in tackling hard problems, including mathematical challenges that require years of computation. A key obstacle in efficiently solving such problems lies in effectively partitioning them into many, frequently millions of subproblems. Existing automated partitioning techniques, primarily based on lookahead methods, perform well on some instances but fail to generate effective partitions for many others.

This paper introduces a powerful partitioning approach that leverages prefixes of proofs derived from conflict-driven clause-learning solvers. This method enables non-experts to harness the power of massively parallel SAT solving for their problems. We also propose a semantically-driven partitioning technique tailored for problems with large cardinality constraints, which frequently arise in optimization tasks. We evaluate our methods on diverse benchmarks, including combinatorial problems and formulas from SAT and MaxSAT competitions. Our results demonstrate that these techniques outperform existing partitioning strategies in many cases, offering improved scalability and efficiency.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Satisfiability solving, parallel computing, problem partitioning

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.3

Supplementary Material *Software*: `https://github.com/zaxioms0/proofix`
archived at `swh:1:dir:021ba6e9dc38762dc28c4ecf9df973344138898d`

Funding Supported by the National Science Foundation (NSF) under grant CCF-2415773 and funding from AFRL and DARPA under Agreement FA8750-24-9-1000.
Joseph E. Reeves: Supported in part by a fellowship award under contract FA9550-21-F-0003 through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR) and the Army Research Office (ARO).

1 Introduction

Satisfiability (SAT) solvers have proven to be invaluable tools for solving large problems of interest to both theorists and industry practitioners. Over the last decade and a half, substantial efforts have focused on parallelizing SAT, leading to two prominent paradigms: clause-sharing portfolios [11] and cube-and-conquer (CnC) [14]. In a clause-sharing portfolio, multiple solver threads are run on the same input formula, typically with slightly varied heuristics, and critical clauses are shared among them. This approach can significantly reduce runtime for problems where sequential SAT methods are already effective, as illustrated by the state-of-the-art cloud solver Mallob [29]. In contrast, a CnC solver partitions the input formula into numerous subproblems that can be solved independently in parallel. This strategy has successfully tackled longstanding open problems in mathematics, including the Pythagorean Triples problem [13], Schur Number Five [12], and the Empty Hexagon problem [15].


© Zachary Battleman, Joseph E. Reeves, and Marijn J. H. Heule;
licensed under Creative Commons License CC-BY 4.0



Cube-and-conquer comprises two main phases: *cubing* and *solving*. In the cubing phase, a splitting variable is chosen to partition the current formula into two subformulas, one with the variable assigned to `true` and the other with the variable assigned to `false`. Ideally, each split produces two distinctly simpler subproblems. Historically, lookahead techniques proved remarkably effective at selecting splitting variables, often enabling superlinear speedups even on thousands of cores [14]. However, poorly chosen splits can result in substantial redundant work or significant imbalances in subproblem difficulty, which in turn may lead to diminished parallel efficiency or total running times that exceed the time required to solve the original formula.

Although early advances with CnC relied on automated partitioning via lookaheads [14], most of the significant successes in the last five years have depended on expert-crafted manual partitions [6,15,32,34]. Manual approaches were favored primarily due to the prohibitive costs or limited effectiveness of automated partitioning with lookaheads. The manual approaches used a combination of trial-and-error and high-level insight into the problem and its encoding. This prevents many potential users of CnC to solve their problems effectively. We propose two novel partitioning methods to overcome these issues.

Our first cubing approach builds on the information contained in clausal proofs produced by SAT solvers. A *clausal proof* is a sequence of redundant clauses (i.e., clauses whose addition preserves satisfiability) ending with the empty clause to prove unsatisfiability. Proofs have been used to assess and compare the usefulness of learned clauses enabling proof summarization [27], solver heuristic tuning [31], and causal reasoning over combinations of solver heuristics [37]. These works rely on access to complete proofs and strictly evaluate clause usefulness within the proofs. A central insight of this paper is that *prefixes* of clausal proofs can serve as effective stand-ins for complete proofs, and that the variables occurring in these proof prefixes provide a powerful heuristic for guiding partitioning decisions.

Although splitting based on proof prefixes turns out to be effective, it can be computationally expensive. To mitigate this cost, our method relies on *static* partitions rather than the *dynamic* partitions typically used in prior work. In a static partition, the same splitting variable is used at every level of the partition tree. For deeper levels, we compute proof prefixes for only a subset of nodes, then aggregate these prefixes to select the next splitting variable.

Despite the inherent restrictions of a static partition, our results show that it performs well across a broad range of problems, including benchmarks from the SAT Competition. In fact, our tool efficiently generates strong partitions for many formulas on which the state-of-the-art partitioning tool, March [14], fails to produce a result in reasonable time.

Our second cubing approach is for problems containing a set of clauses and one large cardinality constraint. Such problems appear frequently in the constraint optimization setting with the cardinality constraint representing some resource bound. The cubing approach assumes that the problem is presented in a cardinality-based input format. We developed a tool that transforms the problem into CNF by encoding the cardinality constraint with the totalizer [3], and during encoding, produces a set of selected auxiliary variables from the totalizer for splitting. In contrast to the first approach, this technique uses a semantic understanding of auxiliary variables to produce a good problem partition.

Unlike many existing cubing-solvers, these techniques completely compartmentalize the cubing and solving phases, allowing the usage of these techniques on top of any off-the-shelf solver. This compartmentalization assists in the goal of learning information about optimal partitions over a class of problems, rather than specific instances, and applying these findings to solve larger instances of the problem.

1.1 Contributions

This work provides 3 main contributions:

1. We developed a tool that performs partitioning based on proof prefixes during partial application of a solver.
2. We developed a tool that performs cardinality-based partitioning with the k-totalizer encoding, selecting auxiliary variables to split on based on their semantic meanings in the encoding.
3. We performed an experimental evaluation comparing our approaches to the widely used partitioning tools, March. We considered SAT and MaxSAT competition formulas, as well as several combinatorial problems and found that our approaches often performed better than existing tools.

2 Background

2.1 Satisfiability

We consider propositional formulas in *conjunctive normal form* (CNF). A CNF formula φ is a conjunction of *clauses* where each clause is a disjunction of *literals*. A literal ℓ is either a variable x (positive literal) or a negated variable $\overline{x}$ (negative literal). An *assignment* α is a mapping from variables to truth values 1 (true) and 0 (false). An assignment α *satisfies* a positive (negative) literal ℓ if α maps $\mathsf{var}(\ell)$ to true (false, respectively), and *falsifies* it if α maps $\mathsf{var}(\ell)$ to false (true, respectively).

An assignment satisfies a clause if the clause contains a literal satisfied by the assignment, and satisfies a formula if every clause in the formula is satisfied by the assignment. A formula is *satisfiable* if there exists a satisfying assignment, and *unsatisfiable* otherwise.

2.2 Cardinality Constraints

A cardinality constraint on Boolean variables has the form $\ell_1 + \ell_2 + \cdots + \ell_s \geq k$ and is satisfied by a partial assignment if the sum of the satisfied literals is at least k. The size of the cardinality constraint is the number of literals (s) it contains. Variables occurring in the cardinality constraint are *data variables*, and new variables added in a clausal encoding are *auxiliary variables*. For a general cardinality constraint with $1 < k < s$, unit propagation should lead to a conflict when $s - k + 1$ data literals in the constraint are falsified. Cardinality constraints occur often in SAT problems, and commonly represent the bound in an optimization problem, for example "at least k packages must be delivered using at most j trucks." There are many ways to encode a cardinality constraint as clauses [1, 3, 23, 24, 30]. In this work, we will use the totalizer encoding, explained further in Section 3.2.

2.3 CDCL and Clausal Proofs

To evaluate the satisfiability of a formula, a CDCL solver [22] alternates between conflict-driven search and inprocessing. In search, the solver performs a series of variable decisions [5, 20] and unit propagations. If no conflict is reached, the formula is satisfiable. If the solver encounters a conflict, the solver performs conflict analysis potentially learning a clause. In case this clause is the empty clause, the formula is unsatisfiable. Otherwise, the solver revokes some of its variable assignments ("backjumping") and then repeats the whole procedure. Additionally, modern solvers incorporate pre- and inprocessing techniques that change the formula in some way, usually reducing the number of variables and clauses or shrinking

the sizes of clauses. These techniques are intermittently interleaved with search during the solving. Some of the most common inprocessing techniques are bounded variable elimination (BVE) [9], vivification [19], and probing [10]

CDCL solvers produce satisfying assignments for satisfiable formulas and proofs of unsatisfiability for unsatisfiable formulas. A clause C is *redundant* w.r.t. a formula φ if φ and $\varphi \cup \{C\}$ are *satisfiability equivalent*. The clause sequence $\varphi, C_1, C_2, \ldots, C_m$ is a clausal proof of C_m if each clause C_i $(1 \leq i \leq m)$ is redundant w.r.t. $\varphi \cup \{C_1, C_2, \ldots, C_{i-1}\}$. The proof is a refutation of φ if C_m is $\bot$. Clausal proof systems may also allow deletion.

The strength of a clausal proof system is determined by the syntactic criterion it enforces when checking clause redundancy. The standard SAT solving paradigm CDCL learns clauses that are logically implied by the formula and fall under the *reverse unit propagation* (RUP) proof system. A clause is RUP if unit propagation on the falsified literals of the clause results in a conflict. The *Resolution Asymmetric Tautology* (RAT) proof system generalizes RUP and all commonly used inprocessing techniques can be compactly expressed using RAT steps. Proofs are typically transformed to a format with hints, e.g. LRAT, before being passed to a formally-verified checker like cake-lpr [35].

When a CDCL solver logs a proof, the only distinction made between proof steps is if they are clause additions or clause deletions. Without additional processing there is no way to determine if a clause addition in the proof originated from conflict learning or an inprocessing technique. In this work, we consume proofs externally without modifying the solver, so the proof-based heuristics described in Section 3.1 will consider both learned clauses and inprocessed clauses.

2.4 Cube and Conquer

Cube-and-Conquer (CnC) [14] is a powerful methodology for solving difficult SAT formulas. In typical usage, a formula φ is partitioned into many sub-formulas, $\varphi_1, \ldots, \varphi_n$, such that φ is satisfiable if and only if at least one φ_i is satisfiable. The *cubes* to which the name refers are conjunctions of literals. In particular, if a disjunction of cubes $\psi := \psi_1 \vee \cdots \vee \psi_n$ is a tautology, then

$$\varphi \iff \varphi \wedge \psi \iff \bigvee_i \varphi \wedge \psi_i$$

Thus, the choice of partition from which the technique gets its name is $\varphi_i := \varphi \wedge \psi_i$. With this partition, one can dispatch independent CDCL solvers in parallel, as depicted in Figure 1, and enjoy substantial speedups.

In the best case, each $\varphi \wedge \psi_i$ is of equal difficulty and substantially easier to solve than φ. In the worst case, however, each $\varphi \wedge \psi_i$ can be no easier, and sometimes harder, than just φ. In this case, the wall-clock time is no better than just running φ, and the CPU time can blow up exponentially. As a result, the choice of ψ is of great importance and the focus of this paper. Historically, this has been done with expert insight and domain knowledge on a problem-by-problem basis [15, 32, 34], or with lookahead techniques [14]. In this paper, we will propose several new, automatic methods for finding good choices of ψ.

The state-of-the-art tool for automatically computing partitions is March, which combines the David-Putnam-Logemann-Loveland (DPLL) algorithm [8] with lookaheads. For a general discussion, see the Handbook of Satisfiability [16, 18], while we describe here an exemplary scheme. Given a CNF formula φ, a lookahead on literal ℓ works as follows: First, ℓ is assigned to `true`, followed by unit propagation. Second, in case there was no conflict, the difference between φ and the reduced formula φ' is measured. The quality of look-ahead techniques

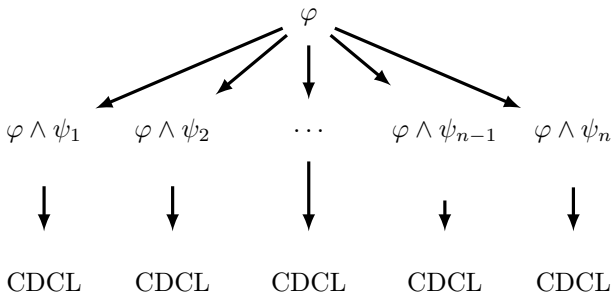


Figure 1 Cube-and-conquer heuristically splits a formula φ into n subformulas $\varphi \wedge \psi_1$ to $\varphi \wedge \psi_n$ and solves the subformulas using CDCL.

depends heavily on the measurement used. A frequently used method weighs the clauses in $\varphi' \setminus \varphi$ (i.e., the clauses that are reduced but not satisfied). Third, all simplifications are reversed to return to φ. If a conflict was detected during the lookahead, then ℓ is forced to `false` and is called a failed literal. The measurements are used to determine the splitting variable in each node of the tree. In general, a variable x is chosen for which both the lookahead on x and $\overline{x}$ result in a large reduction of the formula.

3 Partitioning Techniques

3.1 Proof Prefix Based Splitting

In this section, we describe a partitioning technique based on proof prefixes. A proof, in this context, is a series of clauses that are redundant with respect to the original formula. A formula is unsatisfiable when the empty clause is derived. A *proof prefix* is a sequence of addition steps starting from the beginning of a proof. The key metrics we extrapolate from proof prefixes are variable occurrences, or the number of times a variable appears in a clause addition step. The partitioning technique arises from two observations regarding proof prefixes.

1. Variables which occur frequently in a non-trivial proof prefix will often continue to appear frequently in the remainder of the proof. By non-trivial we mean a proof prefix with a large enough number of steps such that the solver has performed reasoning over several restarts to explore the search space.
2. In problems for which effective partitions are known, splitting variables often occur frequently in a proof generated from the original (unpartitioned) formula.

We ground these intuitions in Figure 2, which shows the variable occurrences in the DRAT proof over time for the $\mu_5(13)$ problem. Despite the proof taking more than 10^7 clause addition steps, the known best variables have risen to near the top by only 10^5. This observation is what allows us to use prefixes as an adequate substitute for the entire proof.

At a high level, we compute proof prefixes, find the most occurring variables in these prefixes, and use theses variables as splitting variables. From the solver's brief search we infer which variables are important in the problem. More concretely, given a formula φ, we run it with an off-the-shelf solver until a desired number of clauses are added to the proof emitted. This is achieved by piping the proof output of the solver, waiting for a desired number of addition steps, and then killing the solver. No modifications to the solver are required. Once the proof prefix is known, we count the variable occurrences in the proof, both positive and negative, and pick the most frequently occurring variable as the next variable to split on.

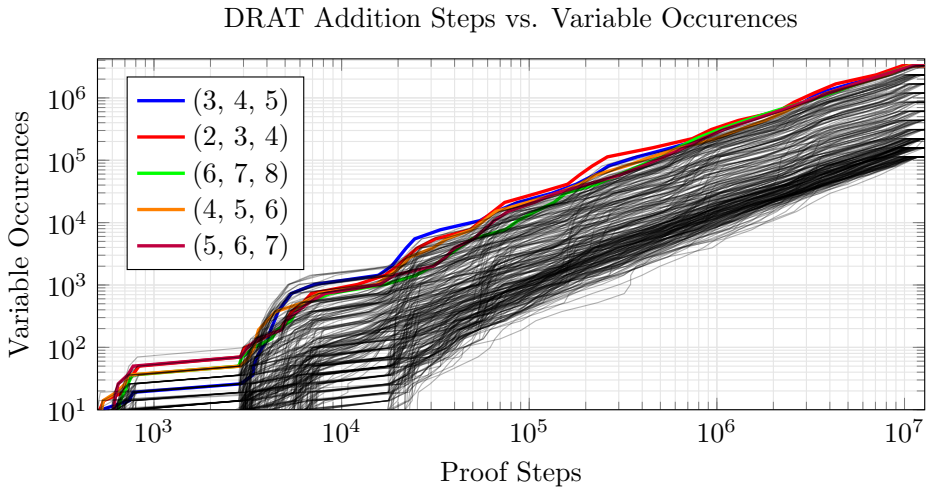


Figure 2 The variable occurrences over the course of a DRAT proof generated by solving the $\mu_5(13)$ problem. This demonstrates that when variables rise to the top, they tend to stay there, and also that the best variables make good splitting choices. In this case, the top 5 variables by the end are exactly the ones found to be good by Subercaseaux et al. [34] and the top 5 variables after 10^5 steps approximate them well enough to create a good partition.

Once a splitting variable x is obtained, we create new formulas, $\varphi \wedge x$ as well as $\varphi \wedge \overline{x}$, and restart the process on both formulas. Naively, generating a complete partition in this manner, where each cube has size d, would require generating $O(2^d)$ proof prefixes, which would be prohibitively expensive. To get around this, we introduce the notion of a *static* partition. A static partition is one where we start with a static set of d splitting variables and then generate cubes with all 2^d combinations of polarities. This can be viewed as a balanced binary tree depicted in the left of Figure 3. In contrast, a *dynamic* partition, depicted in the right of Figure 3, may have unique variables at every vertex in the tree and is not necessarily balanced.

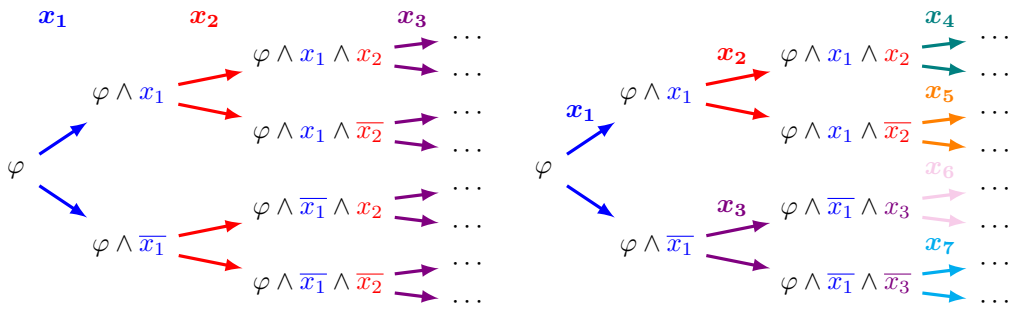


Figure 3 Left: A static partition on variables $\{x_1, x_2, x_3, \ldots\}$. Right: A dynamic partition on variables $\{x_1, \ldots, x_7, \ldots\}$.

Given a static partition set of size d, to generate the $d+1$th variable, we sample a constant number of cubes from the current partition, find the best variable by summing the occurrences across all of the proof prefixes, and use it to extend the set. In this way, we can

generate a static partition of depth d by only generating $O(sd)$ proof prefixes, where s is the number of samples per layer. The generation of proof prefixes can be parallelized, so if the number of samples is fewer than the number of cores, which is often sufficient, the cube generation time is a linear function of depth. However, we must synchronize upon solver completion, before processing the proof prefixes. One "layer" of this process is depicted in Figure 4. The specific number of clauses in each proof prefix that produces the best results is formula-dependent, and future work is needed to determine this automatically. However, in our experimental evaluation, we found $100,000$ is an effective cutoff for a wide range of competition formulas, and for many problems this is sufficient. Moreover, the required proof prefix size seems to scale far slower than problem size as demonstrated by the technique's ability to effectively partition very difficult problems with comparatively little preproccesing. Despite the restriction to static partitions, our technique is highly effective across a variety of benchmarks. Moreover, finding an effective static split for a problem has numerous additional benefits, as will be discussed in Sections 4 and 5. This technique was developed into an automated tool, Proofix.

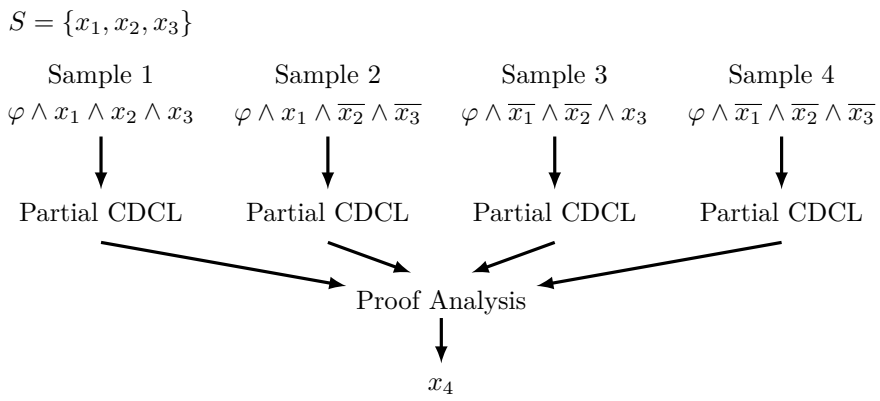


Figure 4 Example generating x_4 from static partition $\{x_1, x_2, x_3\}$ using 4 samples.

3.2 Totalizer Based Splitting

In this section, we present a partitioning technique based on auxiliary variables from the totalizer encoding. We assume that the problem is given in the cardinality-based input at-least-k conjunctive normal form (KNF [26]), but the cardinality constraint may be encoded as an at-most-k constraint by negating the literals and modifying the bound. This technique makes no assumptions about the clauses in the formula and forms a static partition solely by selecting auxiliary variables from the totalizer encoding.

The totalizer is one of the most widely used classes of cardinality constraint encodings, with incremental variants appearing in modern MaxSAT solvers. The totalizer is structured as a binary tree that incrementally counts the number of true data literals at each node. Data literals form the leaves, and each node has auxiliary variables representing the unary count from the sum of its children counters. For example, in Figure 5 with counters indexed by depth and node id, the counter $c_1^{3,1}$ is set true if either ℓ_1 or ℓ_2 is true, and $c_2^{3,1}$ is true if both ℓ_1 and ℓ_2 are true. Further, if both counters $c_2^{3,1}$ and $c_2^{3,2}$ are true, then the counter $c_4^{2,1}$ is true.

Output variables denoted by o_i at the root of the tree represent the count of true data literals across the entire cardinality constraint. The bound k for an at-most-k constraint is enforced by adding the unit $\overline{o}_{k+1}$. An at-least-k cardinality constraint would enforce the bound with the positive unit o_k. The encoding can be simplified by only encoding the count up to $k+1$ at each node [23], and we adopt this simplification for our encoding.

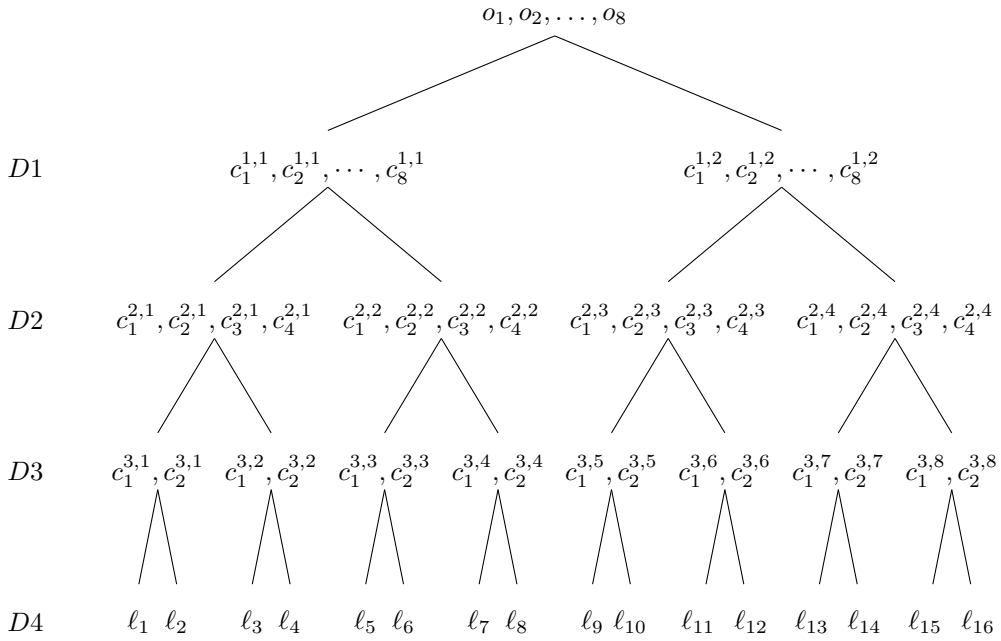


Figure 5 Visualization of a totalizer for $\ell_1 + \ell_2 + \cdots + \ell_{16} \leq 7$, where $\overline{o}_8$, and both $\overline{c}_8$s would be unit to enforce the bound. Each node has an independent set of auxiliary variable counters c_{cnt}. The nodes are numbered left to right for each depth, and a counter for a given node id at depth dep can be distinguished as $c_{cnt}^{dep,id}$.

To motivate our splitting heuristic, first consider the totalizer in Figure 5. Selecting $c_4^{1,1}$ as a splitting variable will result in the following two cases:

- If $c_4^{1,1}$ is true, at least 4 of the data literals from the set $\ell_1, \ell_2, \ldots, \ell_8$ are true. This also means that at most 3 data literals from the set $\ell_9, \ell_{10}, \ldots, \ell_{16}$ are true due to the bound of at most 7.
- If $c_4^{1,1}$ is false, at most 3 of the data literals from the set $\ell_1, \ell_2, \ldots, \ell_8$ are true. This provides no information about the data literals from the set $\ell_9, \ell_{10}, \ldots, \ell_{16}$.

Intuitively, splitting on an auxiliary variable at an internal node in the totalizer will designate how many true data literals are among its children and potentially its sibling nodes. Furthermore, the closer the auxiliary variable is to the root, the more data literals it will impact. However, a bad selection can create unbalanced subproblems. For instance, if we split on $c_7^{1,1}$, when $c_7^{1,1}$ is true all of the data literals from $\ell_9, \ell_{10}, \ldots, \ell_{16}$ are set false due to the bound. This may create a trivial cube. On the other hand, when $c_7^{1,1}$ is false, we obtain very little information other than that at least one of $\ell_1, \ell_2, \ldots, \ell_8$ is false, and this would likely not benefit the solver. Therefore, we focus our splitting on auxiliary variables with a large impact that also correlate with the cardinality constraint's bound. To achieve this, we use the ratio $Rk = \frac{k}{s}$ for a cardinality constraint with bound k and s data literals. We select a counter from a node (id) with n counters as $\mathsf{SelectCounter}(id) = \lfloor Rk \times n \rfloor$. The selection procedure is as follows:

1. Select a starting depth d and desired number of splitting variables v.
2. Sort the nodes at depth d by number of counters, largest to smallest.
3. For each node (id) in d, select the counter $\mathsf{SelectCounter}(id) + 1$ for odd nodes and $\mathsf{SelectCounter}(id)$ for even nodes.
4. If v variables are selected, break. Otherwise, proceed to depth $d+1$ and return to step 2 to select the remaining variables.

Consider the totalizer in Figure 5 with $Rk = \frac{7}{16} = 0.4375$. Assume that we want 6 splitting variables, starting from depth 1. Start with node 1 which contains 8 counters. The counter from node 1 will be $\mathsf{SelectCounter}(1) + 1 = \lfloor 8 \times 0.4375 \rfloor + 1 = 4$, which is variable $c_4^{1,1}$. Then move on to node 2 and select $c_3^{1,2}$. Next, move to depth 2, and since all nodes have the same number of counters we process them in order. At node 1 with 4 counters take $\mathsf{SelectCounter}(1) + 1 = \lfloor 4 \times 0.4375 \rfloor + 1 = 2$, which is variable $c_2^{2,1}$. Proceed with the remaining nodes at depth 2 selecting $c_1^{2,2}$, $c_2^{2,3}$, and $c_1^{2,4}$. These 6 variables are then used as the splitting variables to generate a static partition of the formula.

The reason we alternate between $\mathsf{SelectCounter}(id)$ and $\mathsf{SelectCounter}(id) + 1$ is to ensure that the sum of the counts of selected splitting variables across all nodes for a given depth resembles the ratio. This reduces the number of unhelpful or trivial cubes. For example, the selected variables at depth 2: $c_2^{2,1}$, $c_1^{2,2}$, $c_2^{2,3}$, and $c_1^{2,4}$ will sum to 6 in the cube where they are all set to true. If we only selected the c_1 counters, the sum would be 4 and this is far from the bound of 7, and if we only selected c_2 counters, the sum would be 8 and this is trivially unsatisfiable.

The starting depth is assigned based on the number of desired splitting variables such that all nodes in a given depth are processed if possible. In our experimental configuration, we chose 12 splitting variables and therefore started at depth 2, giving 4 variables at this depth and 8 variables at depth 3.

This approach could be adapted to the more compressed modulus totalizer [24], which uses a quotient and remainder at each node to reduce the number of auxiliary variables required to count the sum, by selecting quotient variables at each node. Other hierarchical encodings, for example, those used in pseudo-Boolean solving, would also be candidates for this approach. However, it is not clear how splitting could be achieved with a non-hierarchical encoding like a sequential counter. Also, we do not consider splitting on formulas with many cardinality constraints. It is possible to select a subset of variables from each cardinality constraint encoding, but this would require heuristics for determining the relative importance of each cardinality constraint. Lastly, we focus on unsatisfiable problems with a tight bound, which means that adding 1 to the bound would make the problem satisfiable. If the bound is not tight, we may need to adjust the ratio Rk by sampling for trivial cubes.

4 Explainable Splitting

One of the most common criticisms of large computer-aided proofs is their opacity: many of these SAT-based proofs are massive. For example, the proof of Schur Number Five is roughly 2 petabytes [12], and even CDCL-based proofs for problems that are solvable in seconds can remain impenetrable. It may not be surprising that many basic steps are hard to grasp, but even the high-level structure is completely unclear. However, an effective *static split* can drastically improve our high-level understanding of these proofs. Because our approach extracts a static split directly from CDCL-generated proofs, it exposes the solver's underlying reasoning and highlights the variables it deems most significant.

We have observed that static splits can offer several key advantages:

- *Demystifying solver reasoning.* While the solver's detailed steps may remain intricate, a static split reveals the high-level structure of the proof, making it easier to follow how subproblems branch and simplify.
- *Pinpointing crucial variables.* Intuitively, crucial variables are the ones which always partition a problem into subproblems with equal difficulty. A static split can help identify crucial variables as their introduction would observably split every vertex in half. This contrasts with a dynamic split where it is difficult to identify the crucial variables as they may only occur a few times at unrelated nodes.
- *Facilitating generalization.* Once it is clear which variables matter most, users can readily adapt or replicate that partitioning scheme for related problems.
- *Ease of extension.* Dynamic splits often yield complex trees that are hard to modify. In contrast, a static split is straightforward to expand by adding additional variables that resemble those already used in the partition.

5 Experimental Evaluation

All experiments were performed at the Pittsburgh Supercomputing Center on nodes with 128 cores and 256 GB RAM [7]. We ran 32 solver executions in parallel per node with 10,000 second timeouts. Therefore, each solver process held approximately 8GB of memory.

We compared our splitting techniques with March, the state-of-the-art partitioning tool which uses lookaheads and is not constrained to static partitions. Our experimental comparison focuses exclusively on CnC solvers, primarily due to the fairness of comparison, but also because many of the explainability and usability benefits of our techniques only make sense in the context of CnC partitions, so we felt it warranted to restrict our focus to CnC.

5.1 Stability Under Search Parameters

One of the properties that makes the proof prefix technique powerful is the fact that the preprocessing time required scales far slower than problem difficulty. Indeed, looking at several difficult problems, if Proofix is able to find known good splitting variables, it can do so without needing to search deep into the proof.

The $\mu_n(k)$ problem asks to find the minimum number of convex n-gons induced by placing k points in the plane with no three points in a line. Recently, Subercaseaux et al. [34] conjectured that $\mu_5(n) = \binom{\lfloor n/2 \rfloor}{5} + \binom{\lceil n/2 \rceil}{5}$. The $\mu_5(15)$ benchmark asks whether it is possible to construct only 76 convex pentagons in the plane with 15 points, one less than the known minimum of 77. It should be noted that $\mu_5(15)$ is substantially easier for state-of-the-art MaxSAT solvers with an out-of-the-box `wcnf` formula than for CaDiCaL using the totalizer encoding. This problem takes at least 48 CPU hours.

The $\chi_\rho(\mathbb{Z}^2)$ problem is a heavily optimized formula which was shown to be logically equivalent to asking whether there exists a packing chromatic number of the subset of the plane, $\{(x, y) \in \mathbb{Z}^2 \mid |x| + |y| \leq 15\}$, using 14 colors and center color 6. The "good" variables are precisely the ones corresponding to the "plus" tiling of the graph as described by Subercaseaux et al. [33]. With a manual partition of over $5,000,000$ cubes, this problem was solved in $4,851$ CPU hours.

The $7gon$-$6hole$ problem asks whether every set of 24 points with no 3 three points in a line contains either a convex 7-gon, or a 6-hole (a convex 6-gon with no points inside of it). This problem is estimated to have a single-core runtime of $1,000$ CPU hours and a manual partition was found reducing it to 200 CPU hours (on a single core) [15].

Table 1 Stability of quality of variables under increasing prefix lengths. Each formula was statically partitioned into cubes of depth 15. The number of "good", as determined by their manual partitions in previous works, is shown as well as how many of the variables in the final partition are "good." If there are fewer "good" variables than 15, the score is at most the number of "good" variables.

Formula	# "Good"	# Variables	Prefix Length				
			10^3	10^4	10^5	5×10^5	10^6
$\chi_\rho(\mathbb{Z}^2)$	63	7,669	0/15	0/15	12/15	12/15	11/15
$\mu_5(15)$	13	58,826	7/15	5/15	4/15	3/13	4/15
7gon-6hole	20	28,878	2/15	8/15	13/15	12/15	12/15

Table 1 depicts the tool looking for variables that are known to be good on 3 very difficult problems. Using just 32 samples per variable, on two of them Proofix is able to pick out many known good variables after only searching for a tiny fraction of the time that the problem itself would take. While it is possible that Proofix was able to come up with an alternative set of "good" variables for the $\mu_5(15)$ problem, it more likely provides evidence that for certain proofs the prefix is not an adequate substitute. Understanding why this is the case is the subject of future work.

One caveat to note is that while finding good splitting variables can now be done automatically for many problems, for large problems such as these, it is not necessarily the case that the cubes found will always achieve comparable performance to the ones found manually due to factors such as variable ordering or the specific subset of "good" variables used in the static partition. The partition used to solve the $\chi_\rho(\mathbb{Z}^2)$ problem, for example, used a dynamic partition with an unbalanced binary tree that Proofix cannot capture directly. Future work is required on how to use proof prefixes to efficiently capture dynamic partitions.

5.2 Maximum Satisfiability Problems

In this section, we evaluate the totalizer-based splitting and proof-based splitting on problems containing one large cardinality constraint. These types of problems occur naturally as unweighted maximum satisfiability (MaxSAT) problems. Each MaxSAT problem consists of a set of hard clauses and soft units. The objective is to find the optimum (minimum) number of soft units that must be falsified while satisfying all hard clauses. MaxSAT problems can be converted to SAT by combining the hard clauses with one, often large, cardinality constraint stating at most k soft units are falsified. We can generate an unsatisfiable SAT problem by making the cardinality constraint's bound the optimum minus one, and a satisfiable SAT problem by making the bound the optimum.

In our evaluation, we consider MaxSAT benchmarks from the 2023 competition unweighted track [17]. We generate unsatisfiable SAT problems from formulas with known bounds. The SAT problem is transformed into CNF by encoding the cardinality constraint as a totalizer. The cardinality constraint is transformed from at-least-k to at-most-k if this makes the encoding size smaller. Before encoding the cardinality constraint, we sort data literals using the best literal sorting found in recent work [25]. This step improves the default solver performance, creating a better baseline for partitioning.

We filter out problems with a CaDiCaL solving time of less than 500 seconds, leaving five benchmark families: judgment aggregation (judge) [17], model-based diagnosis (mbd) [21], approximately propagation complete CNF for an all-different encoding of pigeon hole (optic) [2],

Table 2 Comparison of partitioning methods on combinatorial problems and MaxSAT competition benchmarks. Evaluation is in CPU time.

formula	baseline	March			Proofix			Totalizer Splitting	
	1 core	1 core	32 core	Pre.	1 core	32 core	Pre.	1 core	32 core
judge	3,654	4,893	3,162	219	3,437	2,447	19	7,851	4,289
mbd	2,170	2,914	313	287	2,512	409	37	3,784	290
optic	1,236	908	195	23	708	22	45	1,150	135
uaq	2,520	1,408	453	4	970	62	43	1,960	129
mindset	2,162	18,018	1,372	357	16,375	2,252	42	19,002	1,164
max10	6,282	1,939	920	0	7,645	238	29	2,498	269
cross13	> 80,000	?	> 10,000	43	64,610	2,206	71	77,664	3,125
$\mu_5(13)$	2,317	2,526	181	8	1,367	84	80	2,355	543

user query authorization (uaq) [2], and minimum rule set for labeling data (mindset) [2]. For each family, we selected the hardest problem for CaDiCaL to solve, and present the results at the top of Table 2.

Proofix outperforms the other techniques in both single and 32 core performance on the majority of the problems. Totalizer split performs adequately, which is notable due to its lack of preprocessing time other than time (less than a second) required to encode the cardinality constraint into CNF. Of all the selected problems, only mindset is not successfully partitioned by any technique, with totalizer split producing the best 32 core speedup of only 2×.

The preprocessing time of Proofix is consistent across problems because there is only minor variation in the time it takes for the solver instances to produce 100,000 proof steps. On the other hand, March has a large range of preprocessing times. For example, in mbd March spends as much time in preprocessing as totalizer split spends solving the problem on 32 cores.

In addition, we examined the splitting variables selected by Proofix to find out if it used any auxiliary variables from the totalizer encoding. Proofix used some auxiliary variables for optic and mbd, a single auxiliary variable for judge, and no auxiliary variables in the other problems. This suggests that Proofix can discover auxiliary variables when they are more useful, as is the case for mbd based on the superior performance of totalizer split. Furthermore, Proofix achieves this without any additional insight into the problem or variable semantics. Many of the other problems can be effectively partitioned by either auxiliary variables or data variables, as seen by the similar performance of Proofix and totalizer split.

Next we discuss three additional hard combinatorial problems to test the limits of these splitting techniques. We describe the problems and their encodings at a high level below.

The *Max Squares* problem [36] asks whether you can set m cells to true in an $n \times n$ grid such that no set of four true cells form the corners of a square. Problem variables denote whether a cell is in the solution. The problem is encoded with clauses containing 4 literals blocking the 4 corners of each possible square on the grid and one cardinality constraint over all of the problem variables. For a 10×10 grid the optimal value is 61, so a bound of 62 on the cardinality constraint makes the formula unsatisfiable.

The crossing number $cr(G)$ of a graph G is the lowest number of edge crossings of a plane drawing of the graph G. In 1960, Guy conjectured that the crossing number of the complete graph is $cr(K_n) = \frac{1}{4} \cdot \lfloor \frac{n}{2} \rfloor \cdot \lfloor \frac{n-1}{2} \rfloor \cdot \lfloor \frac{n-2}{2} \rfloor \cdot \lfloor \frac{n-3}{2} \rfloor$ [28]. It is known that $cr(K_{13}) = 225$. The benchmark *cross13* asks whether there is a K_{13} drawing with 224 crossings.

On the max10 problem, March exhibits good single-core performance but cannot match the improved 32 core times from both of our techniques. Likewise, March fails to solve the cross13 problem with a 10,000 second cube timeout. For each of these problems Proofix produces a partition with the best 32 core performance, and it does so without using the auxiliary variables from the totalizer encoding. The totalizer split can approximate the performance of Proofix with slightly worse performance on each of the problems. Notably, the preprocessing time for Proofix on the much harder cross13 is lower than for $\mu_5(13)$, displaying the potential of a static splitting technique to scale and achieve a good partitioning with relatively low preprocessing overhead.

5.3 SAT Competition Formulas

To measure the generalizability of Proofix, we compared it to both CaDiCaL [4] and March on SAT competition formulas from the years 2022, 2023, and 2024. We considered problems which were both UNSAT and took CaDiCaL more than $1,000$ seconds.

Proofix was used with settings where it learned the first 10^5 clauses of the proof and generated a static partition of depth 10, resulting in 2^{10} subprobems. Comparing Proofix to CaDiCaL directly, as shown in Figure 6, we see that 63% of competition formulas perform better when first split and run on 32 cores than when run with CaDiCaL alone, including the preprocessing time for splitting. If we disregard preprocessing time, this number increases to 75%. Including preprocessing, there are 26 problems where Proofix performed more than $10\times$ better than CaDiCaL, and even one which performed $100\times$ better.

Unfortunately, March timed out or crashed while generating the cubes on multiple instances. We suspect that this is due to March learning a lot of "local" clauses and exceeding the available memory on formulas with many small clauses. To ensure a fair comparison, and to eliminate many additional cases where March failed to generate a partition before timing out, it was limited to a split of depth at most 10.

As can be seen in Figure 7, Proofix performs favorably compared to March on many problems. Considering only problems where March did not segfault, Proofix performed better than March on 53% of problems, and considering problems where March failed for any reason, that number increases to 61%. While there are a couple of cases where March beats Proofix by a large margin, there are far more where the opposite is true, and even two cases where Proofix produces a partition which is over $1,000\times$ better than March.

Although Proofix always succeeds in generating the cubes, there are 10 cases where at least one cube takes more than $10,000$ seconds. This is in contrast to March, where a cube times out in 5 cases, March itself times out in generating cubes in 8 cases, and it segfaults while generating cubes in 35 cases. The segfaults persisted across multiple compiled March binaries and multiple computers. In addition, it is worth noting that segfaults seem to be roughly correlated with partitioning difficulty, as 6 of the 10 instances for which Proofix times out on a cube also cause segfaults in March.

It can also be seen that while both tools can fail to find a good split in some cases, the amount that Proofix can fail by is far less than March. In the worst case, March's 32 core time is $55,000$ seconds slower than CaDiCaL, whereas Proofix's 32 core time is at most $21,800$ seconds worse. Moreover, there are 9 problems where March produces a partition that has a time on 32 cores worse than CaDiCaL, compared to Proofix for which there are only 4 (conditioned on March generating a partition at all).

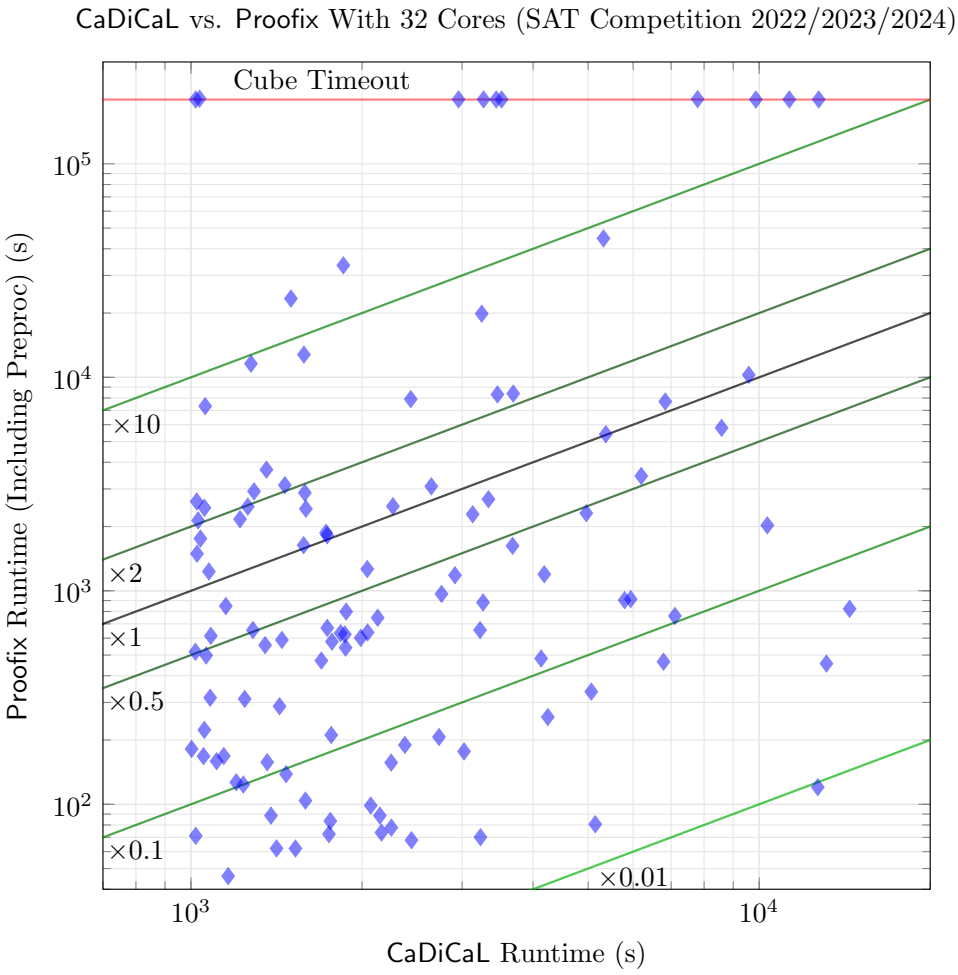


Figure 6 CaDiCaL runtimes vs. Proofix cube generation and runtimes on 32 cores. Lower is better.

6 Applications and Results

We believe that the proof prefix cubing technique is of interest to people solving formulas of all sizes. For competition-sized problems, one way to make use of an n-core machine is to dedicate $n-1$ cores to cubing and solving in parallel, reserving the last core to run an out-of-the-box solver. The last core is used in this manner to account for the case that either the overhead of partitioning outweighs the difficulty of the formula itself or an effective partition cannot be found. If one were to take this approach for the SAT competition problems in 2022, 2023, and 2024, they would see improvements on nearly 65% of the problems from those years. Similar improvements can be seen for MaxSAT, although the formulas tended to be easier overall. However, this is due to the kinds of formulas in the MaxSAT competition, rather than the applicability of Proofix itself. We believe that the main application is toward resolving large combinatorial problems. We repeatedly observe that the partial-proof technique is able to recreate the results of manual effort [15,32,34], and inform optimal partitions automatically. Thus, we believe that Proofix's primary use-case will be returning near-optimal, easily interpretable, static partitions, which can help users quickly identify semantic patterns in cubes in order to solve large problems.

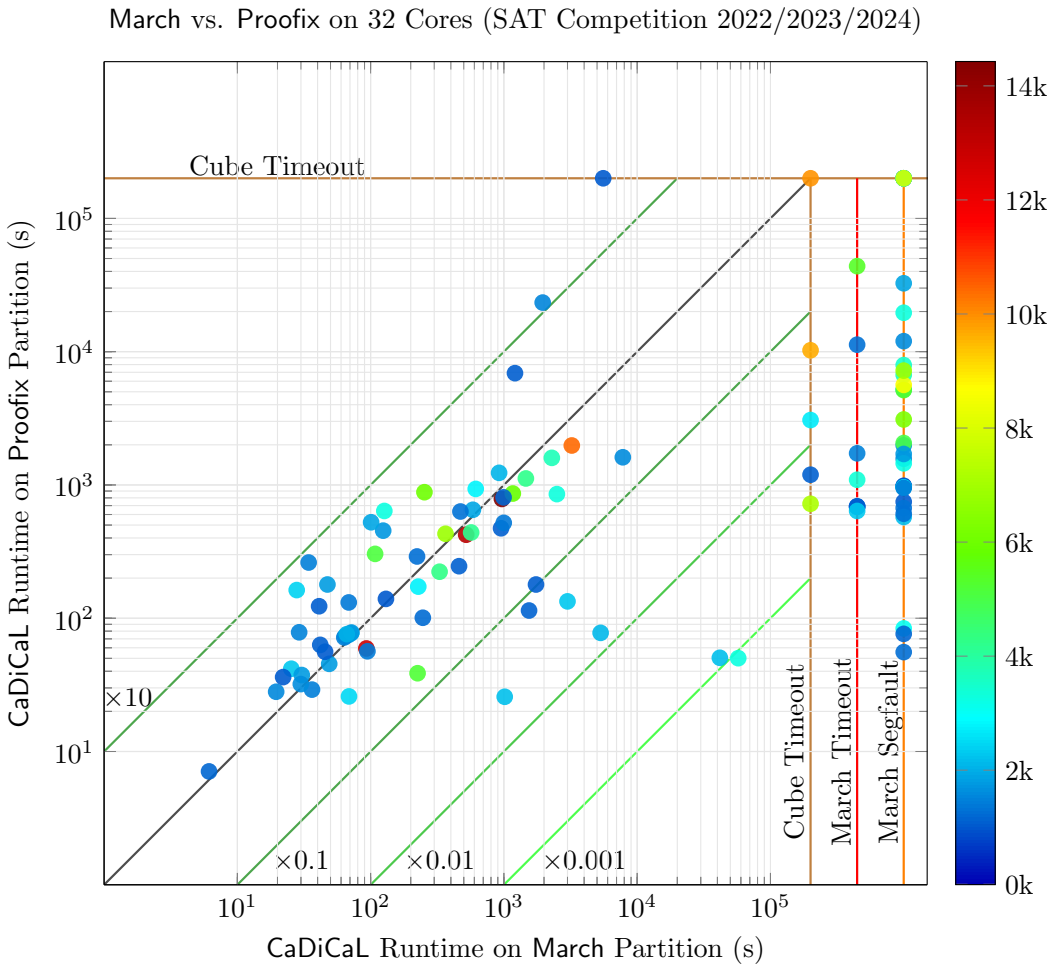


Figure 7 A comparison of March vs. Proofix on SAT competition problems using 32 cores. The color of a point indicates how long the original problem takes CaDiCaL to solve with no partitioning. The diagonal lines compare orders of magnitude of performance between March and Proofix for solving the partition. Several lines are added on the right to denote cases where March failed for one of a few of reasons. Being lower on the graph is better.

7 Conclusion and Future Work

In this paper, we presented two novel techniques for automatically partitioning SAT formulas, one based on proof prefixes and the other based on the totalizer encoding. We demonstrated that the limitation to static partitions is not a major setback, and also provides numerous qualitative benefits towards explainability. Finally, we developed tools for both splitting techniques and demonstrated that these techniques outperform the state-of-the-art tool on numerous problems. There are several questions left open for future work:

- What makes some proofs more amenable than others for splitting, and is this property identifiable in a prefix? In other words, can we detect when a prefix is unlikely to yield a good partition?
- Given that we can find good variables for a partition, is there a way to automatically turn them into a dynamic partition, or generalize them from their semantic meaning?
- How do the splitting techniques in this paper compare to clause-sharing solvers, both on competition and difficult combinatorial problems?

References

1 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)*, pages 167–180. Springer, 2009. doi:10.1007/978-3-642-02777-2_18.

2 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. MaxSAT Evaluation 2018 Benchmarks. https://helda.helsinki.fi/items/7c51d9bd-20e8-428f-8926-b2bbf151dc4c, 2018. Accessed: 2025-04-02.

3 Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming (CP)*, pages 108–122, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/978-3-540-45193-8_8.

4 Armin Biere. Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018. *Proceedings of SAT Competition*, 14:316–336, 2017.

5 Armin Biere and Andreas Fröhlich. Evaluating CDCL variable scoring schemes. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 of *LNCS*, pages 405–422, 2015. doi:10.1007/978-3-319-24318-4_29.

6 Joshua Brakensiek, Marijn J. H. Heule, John Mackey, and David Narváez. The resolution of keller's conjecture. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 48–65, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-51074-9_4.

7 Shawn T. Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A. Nystrom. *Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research*, pages 1–4. Association for Computing Machinery, New York, NY, USA, 2021. doi:10.1145/3437359.3465593.

8 Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. doi:10.1145/368273.368557.

9 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005. doi:10.1007/11499107_5.

10 Jon William Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, USA, 1995.

11 Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modelling and Computation*, 6(4):245–262, 2010. doi:10.3233/SAT190070.

12 Marijn J. H. Heule. Schur number five. In *AAAI Conference on Artificial Intelligence*, 2018.

13 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)*, volume 9710 of *LNCS*, pages 228–245. Springer, 2016. doi:10.1007/978-3-319-40970-2_15.

14 Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing*, pages 50–65, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

15 Marijn J. H. Heule and Manfred Scheucher. Happy ending: An empty hexagon in every set of 30 points. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 61–80, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-57246-3_5.

16 Marijn J. H. Heule and Hans van Maaren. *Look-Ahead Based SAT Solvers*, volume 185 of *FAIA*, chapter 5, pages 155–184. IOS Press, February 2009. doi:10.3233/978-1-58603-929-5-155.

17 Matti Järvisalo, Jeremias Berg, Ruben Martins, and Andreas Niskanen. MaxSAT Evaluation 2023 Benchmarks. `https://maxsat-evaluations.github.io/2023/benchmarks.html`, 2023. Accessed: 2024-08-13.

18 Oliver Kullmann. *Fundaments of Branching Heuristics*, volume 185 of *FAIA*, chapter 7, pages 205–244. IOS Press, February 2009. `doi:10.3233/978-1-58603-929-5-205`.

19 Chu-Min Li, Fan Xiao, Mao Luo, Felip Manyà, Zhipeng Lü, and Yu Li. Clause vivification by unit propagation in CDCL SAT solvers. *Artificial Intelligence*, 279(C), February 2020. `doi:10.1016/J.ARTINT.2019.103197`.

20 Jia Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 of *LNCS*, pages 123–140, 2016. `doi:10.1007/978-3-319-40970-2_9`.

21 Joao Marques-Silva, Mikoláš Janota, Alexey Ignatiev, and Antonio Morgado. Efficient model based diagnosis with maximum satisfiability. In *International Joint Conference on Artificial Intelligence (IJCAI)*, IJCAI'15, pages 1966–1972. AAAI Press, 2015. URL: `http://ijcai.org/Abstract/15/279`.

22 João Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, pages 131–153. IOS Press, 2009. `doi:10.3233/978-1-58603-929-5-131`.

23 Antonio Morgado, Alexey Ignatiev, and Joao Marques-Silva. Mscg: Robust core-guided maxsat solving: System description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:129–134, December 2015.

24 Toru Ogawa, Yangyang Liu, Ryuzo Hasegawa, Miyuki Koshimura, and Hiroshi Fujita. Modulo based cnf encoding of cardinality constraints and its application to maxsat solvers. In *Tools with Artificial Intelligence (ICTAI)*, pages 9–17, 2013. `doi:10.1109/ICTAI.2013.13`.

25 Joseph E Reeves, Joao Filipe, Min-Chien Hsu, Ruben Martins, and Marijn JH Heule. The impact of literal sorting on cardinality constraint encodings. In *Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2025.

26 Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. From Clauses to Klauses. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification (CAV)*, volume 14681 of *Lecture Notes in Computer Science*, pages 110–132. Springer, 2024.

27 Joseph E Reeves, Benjamin Kiesl-Reiter, and Marijn JH Heule. Propositional proof skeletons. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 329–347. Springer, 2023.

28 R.K.Guy. A combinatorial problem. *Nabla (Bulletin of the Malaysian Mathematical Society)*, 7, 1960.

29 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. `doi:10.1007/978-3-030-80223-3_35`.

30 Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Principles and Practice of Constraint Programming (CP)*, volume 3709 of *LNCS*, pages 827–831, 2005. `doi:10.1007/11564751_73`.

31 Mate Soos, Raghav Kulkarni, and Kuldeep S Meel. : gazing in the black box of sat solving. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 371–387. Springer, 2019.

32 Bernardo Subercaseaux and Marijn Heule. Toward optimal radio colorings of hypercubes via sat-solving. In Ruzica Piskac and Andrei Voronkov, editors, *Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 94 of *EPiC Series in Computing*, pages 386–404. EasyChair, 2023. `doi:10.29007/QRMP`.

33 Bernardo Subercaseaux and Marijn J. H. Heule. The packing chromatic number of the infinite square grid is 15. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–406, Cham, 2023. Springer Nature Switzerland. `doi:10.1007/978-3-031-30823-9_20`.

34 Bernardo Subercaseaux, John Mackey, Marijn J. H. Heule, and Ruben Martins. Automated mathematical discovery and verification: Minimizing pentagons in the plane. In Andrea Kohlhase and Laura Kovács, editors, *Intelligent Computer Mathematics*, pages 21–41, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-66997-2_2.

35 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. Verified propagation redundancy and compositional UNSAT checking in CakeML. *International Journal on Software Tools for Technology Transfer*, 25(2):167–184, 2023. doi:10.1007/S10009-022-00690-Y.

36 Ed Wynn. A comparison of encodings for cardinality constraints in a SAT solver. *ArXiv*, abs/1810.12975, 2018. arXiv:1810.12975.

37 Jiong Yang, Arijit Shaw, Teodora Baluta, Mate Soos, and Kuldeep S. Meel. Explaining SAT Solving Using Causal Reasoning. In Meena Mahajan and Friedrich Slivovsky, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.SAT.2023.28.

Bit-Precise Reasoning with Parametric Bit-Vectors

Zvika Berger
Bar-Ilan University, Ramat Gan, Israel

Yoni Zohar
Bar-Ilan University, Ramat Gan, Israel

Aina Niemetz
Stanford University, CA, USA

Mathias Preiner
Stanford University, CA, USA

Andrew Reynolds
The University of Iowa, Iowa City, IA, USA

Clark Barrett
Stanford University, CA, USA

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

Abstract

The SMT-LIB theory of bit-vectors is restricted to bit-vectors of fixed width. However, several important applications can benefit from reasoning about bit-vectors of symbolic widths, i.e., parametric bit-vectors. Recent work has introduced an approach for solving formulas over parametric bit-vectors, via an eager translation to quantified integer arithmetic with uninterpreted functions. The approach was shown to be successful for several applications, including the bit-width independent verification of compiler optimizations, invertibility conditions, and rewrite rules. We extend and improve that approach in several aspects. Theoretically, we improve expressiveness by defining a new theory of parametric bit-vectors that supports more operators and allows reasoning about the bit-widths themselves. Algorithmically, we introduce a lazy algorithm that avoids the use of uninterpreted functions and quantified axioms for them. Empirically, we show a significant improvement by implementing and evaluating our approach, and comparing it experimentally to the previous one.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Software and its engineering → Formal methods

Keywords and phrases Satisfiability Modulo Theories, Bit-precise Reasoning, Parametric Bit-vectors

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.4

Supplementary Material *Other (Zenodo)*: `https://zenodo.org/records/15143242`

Funding This research was supported in part by the Stanford Center for Automated Reasoning, the Stanford Center for Blockchain Research, Defense Advanced Research Projects Agency (DARPA) contract FA875024-2-1001, Israel Science Foundation (ISF) grant number 619/21, National Science Foundation (NSF) grant number 2303489, and a gift from Amazon Web Services.

1 Introduction

Bit-precise reasoning as provided by Satisfiability Modulo Theories (SMT) for the theory of fixed-size bit-vectors [3] is a key requirement for a wide range of verification applications. In particular, it allows verification engines to reason about machine integers and hardware registers and supports a variety of operators (including arithmetic, bitwise, and word operators such as concatenation and extraction). It is also very useful for other theories, such as floating-point arithmetic and non-linear integer arithmetic, where reasoning over certain fragments of those theories can be reduced to reasoning in the theory of fixed-size bit-vectors [7, 17].

One of the biggest limitations of this theory is already evident in its name: it only allows reasoning about bit-vectors whose size (or, equivalently, width) is *fixed*. Thus, when declaring a bit-vector sort in the SMT-LIB language, its width must be specified as a numeral. This limitation poses a serious expressiveness issue: when a verification tool verifies a property using this theory of bit-vectors, the verification result is only valid for the explicitly specified bit-width(s). In the context of software verification, this means that some programs are


© Zvika Berger, Yoni Zohar, Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli;
licensed under Creative Commons License CC-BY 4.0

proven correct in the presence of, say, 32-bit integer variables, but if their type is changed to 64-bit integers, the verification process has to be repeated. Similar issues occur during the development of specialized solvers for this theory, where much effort is dedicated to the design of *rewrite rules* and the generation of auxiliary *lemmas* [23, 28, 31, 34]. Before adding new rules or lemmas in the solver, it is valuable to be able to verify that they are correct, and their correctness is usually independent of bit-width. Being able to do so fast and automatically using SMT solvers (as opposed to following the lengthier process of developing interactive proofs [12, 19]) is highly beneficial for the development process. However, state-of-the-art SMT solvers do not currently support reasoning over bit-vectors of parametric size. Thus, it is common to verify the correctness of such rewrite rules and lemmas only for fixed bit-widths, from 1 up to some reasonably large value. This does not yield a proof of correctness in general although it increases confidence that the rules and lemmas are also correct for arbitrary bit-widths [28, 31].

Early work on bit-precise reasoning supported reasoning about parametric bit-vectors, but only for a small fragment of the theory (see, e.g., [5, 15, 16, 35]). In all cases, however, the supported fragment is not expressive enough for modern verification efforts.

More recently, an approach for solving formulas over parametric bit-vectors via an *eager* translation to quantified non-linear integer arithmetic with uninterpreted functions was proposed by Niemetz et al. [29, 30]. In that work, bitwise operators and the exponentiation function with base 2, which is required for translating arithmetic bit-vector operators to the integers, are axiomatized by means of uninterpreted functions and quantifiers. In particular, [30] introduced a theory of parametric bit-vectors where bit-widths are implicit, and so it is not possible to reason about then. The translation was implemented outside of an SMT solver and evaluated on three case studies: bit-width independent verification conditions for compiler optimizations [20], invertibility conditions [28], and rewrite rules [34]. In addition, since all three case studies only require reasoning about a *single* parametric bit-width, the implementation of the translation targets a fragment of the SMT-LIB theory of bit-vectors that does not involve multiple bit-widths. As a result, the possibility of verifying, e.g., compiler optimization reductions was limited, as it excluded optimizations involving bit-vector concatenations, extractions and extensions, all of which involve multiple bit-widths.

Later work [39] presented a translation of *fixed-size* bit-vectors to quantifier-free non-linear integer arithmetic with uninterpreted functions (coined *int-blasting*, generalizing earlier approaches, see, e.g., [6, 8, 38]) with the goal of improving the scalability of reasoning over large bit-vectors. Unlike [30], the approach of [39] is *lazy*, does not introduce any quantifiers and does not support reasoning over parametric bit-vectors. Further, it is integrated in cvc5 [1]. While not in general competitive with state-of-the-art bit-vector solvers, int-blasting showed promising performance improvements on problems involving large bit-widths arising from smart contract verification.

In this paper, we propose a procedure for reasoning about parametric bit-vectors based on the eager approach presented in [30]. Our technique extends and combines this eager approach with lazy techniques introduced for reasoning about fixed size bit-vectors via a reduction to the integers in [39]. In particular, we make the following contributions:

- **(i)** We introduce a theory of parametric bit-vectors with symbolic bit-widths as part of its signature. Our theory definition simplifies yet strengthens the theory of parametric bit-vectors introduced in [30] while enabling the reasoning over bit-widths.
- **(ii)** We present a *lazy* algorithm for determining satisfiability of parametric bit-vectors constraints that does not rely on quantifiers (as opposed to [30]) and generalizes the lazy handling of bitwise *and* from [39] to parametric bit-vectors. Additionally, instead of eagerly axiomatizing exponentiation with base 2 (as in [30]), we handle this operator lazily.

(iii) We provide an implementation of a solver for parametric bit-vectors. The implementation is generic: it supports reasoning over arbitrary parametric bit-vector formulas (which goes beyond the prototype implementation presented in [30]). In particular, it fully supports reasoning over bit-widths, constraints on bit-vectors with multiple widths, and operators combining or returning bit-vectors of different widths, such as extraction, concatenation and extension.

(iv) We evaluate our approach on a large and diverse set of benchmarks and show that our technique significantly improves over the eager approach introduced in [30].

For brevity and simplicity, and similarly to previous work [30,39], our formal presentation focuses on *quantifier-free* formulas. We stress however, that our implementation fully supports quantifiers.[1]

The remainder of this paper is organized as follows. In Section 2, we provide the necessary background on first-order logic. In Section 3, we present our definition for a theory of parametric bit-vectors. In Section 4, we describe our new solver for parametric bit-vectors. We provide a detailed evaluation of the solver in Section 5 and conclude with directions for future research in Section 6.

2 Preliminaries

We briefly review notions of many-sorted first-order theories with equality (see [13,37] for more details). Let S be a set of *sort symbols*, and for every sort $\sigma \in S$, let X_σ be an infinite set of *variables of sort* σ. We assume that sets X_σ are pairwise disjoint and define X as the union of sets X_σ. A *signature* Σ consists of a set $\Sigma^s \subseteq S$ of sort symbols and a set Σ^f of function symbols. Arities of function symbols are defined in the usual many-sorted way. Constants are treated as nullary functions. We assume that Σ includes a Boolean sort Bool and the Boolean constants $\top$ (true) and $\bot$ (false). Function symbols whose return sort is Bool are also called *predicate* symbols. We assume that for each sort $\sigma \in \Sigma^s$, a binary equality predicate symbol $\approx_\sigma$ is included in the signature. We will write just $\approx$ when σ is clear or not important.

We assume the usual definitions of well-sorted terms, literals, and formulas, and refer to them as Σ-terms, Σ-literals, and Σ-formulas, respectively. We include the ternary if-then-else operator ite_σ of arity $\mathsf{Bool} \times \sigma \times \sigma \to \sigma$ for each $\sigma \in \Sigma^s$, and omit σ when it is clear from the context.

A Σ-*interpretation* $\mathcal{I}$ maps: each $\sigma \in \Sigma^s$ to a distinct non-empty set of values $\sigma^\mathcal{I}$ (the *domain* of σ in $\mathcal{I}$); each $x \in X_\sigma$ to an element $x^\mathcal{I} \in \sigma^\mathcal{I}$; and each $f^{\sigma_1 \cdots \sigma_n \sigma} \in \Sigma^f$ to a total function $f^\mathcal{I}\colon \sigma_1^\mathcal{I} \times \ldots \times \sigma_n^\mathcal{I} \to \sigma^\mathcal{I}$ if $n > 0$, and to an element in $\sigma^\mathcal{I}$ if $n = 0$. We use the usual inductive definition of the satisfiability relation $\models$ between Σ-interpretations and Σ-formulas.

A *theory* T is a pair (Σ, I), where Σ is a signature and I is a non-empty class of Σ-interpretations that is closed under variable reassignment, i.e., if interpretation $\mathcal{I}'$ only differs from an $\mathcal{I} \in I$ in how it interprets variables, then also $\mathcal{I}' \in I$. Each interpretation $\mathcal{I} \in I$ interprets Bool as the two-element set $\{\top^\mathcal{I}, \bot^\mathcal{I}\}$ and $\approx_\sigma$ as the identity relation over each sort $\sigma \in \Sigma^s$. A Σ-formula φ is *T-satisfiable* (resp. *T-unsatisfiable*) if it is satisfied by some (resp. no) interpretation in I; it is *T-valid* if it is satisfied by all interpretations in I.

The theory $T_{IA} = (\Sigma_{IA}, I_{IA})$ of integer arithmetic is defined as in the SMT-LIB 2 standard [2,4]. Its signature Σ_{IA} is shown in Table 1 and includes a single sort Int, function and predicate symbols $\{+, -, \cdot, \mathrm{div}, \mathrm{mod}, <, \leq, >, \geq\}$, and a constant symbol for every integer

[1] In fact, one of the benchmark sets in our experimental evaluation contains quantified formulas.

Table 1 Signature Σ_{IA} of the theory of integer arithmetic T_{IA}.

Symbol	SMT-LIB Syntax	Arity
$\approx_{\mathsf{Int}}, \not\approx_{\mathsf{Int}}$	`=`, `distinct`	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$
$0, 1, 2, \ldots$	`0`, `1`, `2`, ...	Int
$+, -, \cdot$, div, mod	`+`, `-`, `*`, `div`, `mod`	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$
$\leq, \geq, <, >$	`<=`, `>=`, `<`, `>`	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$

value. Given a set $\mathcal{F}$ of function symbols disjoint from those of Σ_{IA}, the signature $\Sigma_{IA}(\mathcal{F})$ is obtained from Σ_{IA} by the addition of the symbols in $\mathcal{F}$. The theory $T_{IA}(\mathcal{F})$ consists of all $\Sigma_{IA}(\mathcal{F})$-interpretations whose reduct to Σ_{IA} is a T_{IA}-interpretation (i.e., the symbols in $\mathcal{F}$ are freely interpreted). We may remove set braces when listing the elements of $\mathcal{F}$, e.g., by writing $\Sigma_{IA}(f, g)$ instead of $\Sigma_{IA}(\{f, g\})$.

3 A Theory of Parametric Bit-vectors

The SMT-LIB theory of bit-vectors T_{BV} does not support parametric bit-vectors. Neimetz et al. [30] define a theory of parametric bit-vectors based on auxiliary functions (outside of the signature of the theory). These functions provide an implicit mapping from bit-vector variables to their symbolic bit-width and from symbolic parametric bit-vector constants to Σ_{IA}-terms. A notion of *admissibility* is introduced to exclude mappings that assign non-standard interpretations (e.g., interpretations with zero or negative bit-widths). Furthermore, a parametric bit-vector formula is viewed as a *class* of fixed-size bit-vector formulas, i.e., a class of instances with concrete values for the bit-widths. This allows for a notion of *well-sortedness* based on the well-sortedness of these instances. As a consequence of relying on auxiliary mappings for symbolic bit-widths and constants, however, it is not possible to reason about bit-widths within that theory.

In the following, we introduce a new formal definition of the theory of parametric bit-vectors T_{PBV}. As in [30], we have a single sort PBV for bit-vectors of parametric size. Compared to that work, however, our definition of T_{PBV} does not rely on meta-level functions to maintain well-sortedness constraints. Instead, we make symbolic bit-widths of parametric bit-vector terms explicit in the signature via the new operator $|\sqcup|$ and introduce an explicit representation of parametric bit-vector constants via the conversion operator to-pbv. This enables reasoning about both parametric bit-vectors and their bit-widths within the theory.

3.1 Theory Definition

We define the theory of parametric bit-vectors T_{PBV} as the pair (Σ_{PBV}, I_{PBV}), where the signature Σ_{PBV} is the extension of Σ_{IA} described in Table 2. In addition to the integer sort Int, its set of sort symbols includes a single, new sort PBV for bit-vectors of parametric size. The set of function and predicate symbols of Σ_{PBV} consists of parametric variants of a strict subset of the fixed-size bit-vector operators defined in SMT-LIB 2. This set of *parametric bit-vector operators*, however, is complete in the sense that it suffices to express parametric variants of the remaining bit-vector operators from SMT-LIB 2. Additionally, we introduce two new function symbols, $|\sqcup|$ of arity $\mathsf{PBV} \to \mathsf{Int}$ and to-pbv of arity $\mathsf{Int} \times \mathsf{Int} \to \mathsf{PBV}$, in the signature.

In all the interpretations of I_{PBV}, the domain of Int is the set of integer numbers, and the domain of PBV is the set of all bit-vectors of all possible positive widths. The operators in Σ_{IA} are interpreted in the same way as in T_{IA}. The new symbol $|\sqcup|$ is interpreted as the

Table 2 Signatures Σ_{PBV} and $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$, defined as extensions of signature Σ_{IA}.

Symbol	SMT-LIB Syntax	Arity
$\Sigma_{PBV} = \Sigma_{IA}$ + the following operators		
$\approx_{\mathsf{PBV}}$, $\not\approx_{\mathsf{PBV}}$	`=`, `distinct`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{Bool}$
$<_{\text{u}}$, $>_{\text{u}}$, $<_{\text{s}}$, $>_{\text{s}}$	`bvult`, `bvugt`, `bvslt`, `bvsgt`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{Bool}$
$\leq_{\text{u}}$, $\geq_{\text{u}}$, $\leq_{\text{s}}$, $\geq_{\text{s}}$	`bvule`, `bvuge`, `bvsle`, `bvsge`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{Bool}$
$\sim$, $-^{\mathcal{B}}$	`bvnot`, `bvneg`	$\mathsf{PBV} \to \mathsf{PBV}$
$\&$, $\mid$, $\oplus$	`bvand`, `bvor`, `bvxor`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{PBV}$
$\ll$, $\gg$, $\gg_a$	`bvshl`, `bvlshr`, `bvashr`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{PBV}$
$+^{\mathcal{B}}$, $-^{\mathcal{B}}$	`bvadd`, `bvsub`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{PBV}$
$\cdot^{\mathcal{B}}$, $\text{mod}^{\mathcal{B}}$, $\text{div}^{\mathcal{B}}$	`bvmul`, `bvurem`, `bvudiv`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{PBV}$
$\sqcup[\sqcup:\sqcup]$	`pextract`	$\mathsf{PBV} \times \mathsf{Int} \times \mathsf{Int} \to \mathsf{PBV}$
$\circ$	`concat`	$\mathsf{PBV} \times \mathsf{PBV} \to \mathsf{PBV}$
ext_{z}	`pzero_extend`	$\mathsf{Int} \times \mathsf{PBV} \to \mathsf{PBV}$
ext_s	`psign_extend`	$\mathsf{Int} \times \mathsf{PBV} \to \mathsf{PBV}$
$\lvert\sqcup\rvert$	`bvsize`	$\mathsf{PBV} \to \mathsf{Int}$
to-pbv	`int_to_pbv`	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{PBV}$
$\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}}) = \Sigma_{IA}$ + the following operators		
$\&^{\mathbb{N}}$	`piand`	$\mathsf{Int} \times \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$
pow_2	`pow2`	$\mathsf{Int} \to \mathsf{Int}$

function that maps each bit-vector x to its bit-width expressed as an integer. The new symbol to-pbv is interpreted as the function that maps any two integers n and m to the bit-vector of size n that represents $m \bmod 2^n$ in binary notation if $n > 0$, and to an arbitrary bit-vector otherwise. It can model constants of parametric bit-widths, e.g., to-pbv$(k, 0)$ represents the bit-vector that consists of k zeros when $k \geq 0$. Operator $\circ$ is interpreted as the function that maps any two bit-vectors (of any two possible widths) to the bit-vector consisting of their concatenation. Operator $\sqcup[\sqcup:\sqcup]$ denotes the function that maps a bit-vector x, an integer i, and an integer j, in that order, to the bit-vector of width $i - j + 1$ that ranges from the j-th bit of x to the i-th bit of x (inclusive) if $0 \leq j \leq i < |x|$, and is interpreted arbitrarily otherwise. For a bit-vector x, we refer to the i-th bit of x as $x[i]$, which abbreviates $x[i:i]$. We interpret $x[0]$ as the least significant bit (LSB) and $x[|x|-1]$ as the most significant bit (MSB). Operators ext_{z} and ext_s take an integer and a bit-vector argument and are interpreted as in the SMT-LIB theory of fixed-size bit-vectors whenever the first argument (the number of added bits) is non-negative: the former extends the second argument with leading 0s, while the latter extends it with its most significant bit. If the number of added bits is negative, the interpretation is arbitrary. Operators $\sim$ and $-^{\mathcal{B}}$ correspond to one's and two's complement. All other operators of Σ_{PBV} are interpreted according to the following principle: if both operands have the *same* (positive) bit-width, then their interpretation is the same as in the theory of fixed size bit-vectors. Otherwise, the interpretation is arbitrary.

▶ Remark 1. Recall that the signature of the theory of fixed-size bit-vectors T_{BV} has a unique sort $\mathsf{BV}(i)$ for each bit-width $i \in \{1, 2, 3, \ldots\}$. A more natural choice for a theory of parametric bit-vectors might then be the generalization of T_{BV} to sorts of the form $\mathsf{BV}(t)$ where t is an arbitrary integer term. However, this introduces the complication that syntactically distinct sort terms with equivalent integer arguments, such as $\mathsf{BV}(1+3)$ and

$\mathsf{BV}(4)$ or $\mathsf{BV}(x+x)$ and $\mathsf{BV}(2\cdot x)$, would have to denote the same domain (bit-vectors of 4 bits) or domain family (bit-vectors of even bit-width). This exceeds the expressiveness of many-sorted logic where, in effect, distinct sorts denote disjoint domains. For this reason, we define a single sort PBV to represent bit-vectors of all bit-widths. As a consequence, in our signature Σ_{PBV}, terms can be well-sorted that do not match well-sorted terms in T_{BV}. For example, the term $x+^{\mathcal{B}}y$ is well-sorted in our case even when x and y denote two bit-vectors of different bit-widths, something that is not permitted in the language of T_{BV}. Thus, we establish restrictions on the interpretations of Σ_{PBV} (described in Section 3.2) to have semantics that coincide with the semantics of the theory of fixed-size bit-vectors. In the example above, the restriction will be that x and y can only be interpreted as bit-vectors of the same width.

Note that parametric bit-vectors are expected to be supported more faithfully in the upcoming Version 3 of SMT-LIB, which is based on a higher-order logic with dependent types.

3.2 Satisfiability and Admissible Satisfiability

The way it is defined, theory T_{PBV} contains *non-standard* interpretations as it has to account for terms like $(x\circ y)+^{\mathcal{B}}x$ and formulas like $x\circ x\approx x$ that are not well sorted in the theory of fixed-size bit-vectors. To address this laxness of T_{PBV}'s sort system, we therefore define a notion of *admissible* T_{PBV}-satisfiability that excludes such spurious interpretations. This definition relies on function Adm, defined in Algorithm 1. Function Adm collects width constraints while traversing over a given Σ_{PBV}-term. It is defined via a function Bw as given in Algorithm 2, which constructs an integer term representing the symbolic bit-width of a parametric bit-vector term.

▶ **Definition 2.** *Let φ be a Σ_{PBV}-formula and $\mathcal{I}$ a T_{PBV}-interpretation. Let Adm be a function as defined in Algorithm 1. Interpretation $\mathcal{I}$ is* admissible w.r.t. *φ if $\mathcal{I}\models \textsc{Adm}(\varphi)$. Formula φ is* admissibly T_{PBV}-satisfiable *if it is satisfied by a T_{PBV}-interpretation that is admissible w.r.t. φ.*

Thus, a formula φ is *admissibly* T_{PBV}-satisfiable only if it is satisfied by an *admissible* T_{PBV}-interpretation $\mathcal{I}$. We require that an admissible interpretation $\mathcal{I}$ assign bit-widths to Σ_{PBV}-terms in φ while satisfying the *admissibility condition* described by the function Adm defined in Algorithm 1.

▶ **Example 3.** Consider a formula φ given as $y\approx z\circ w$. The admissibility condition $\textsc{Adm}(\varphi)$ is determined as $|y|\approx|z|+|w|\wedge|y|>0\wedge|z|>0\wedge|w|>0$. An admissible T_{PBV}-interpretation $\mathcal{I}$ satisfying φ is given by $y^{\mathcal{I}}=00$, $z^{\mathcal{I}}=0$ and $w^{\mathcal{I}}=0$. Thus, φ is admissibly T_{PBV}-satisfiable. Now, consider formula φ' given as $x\approx x+^{\mathcal{B}}(x\circ x)$. Its admissibility condition $\textsc{Adm}(\varphi')$ is defined as $|x|\approx|x|\wedge|x|>0\wedge|x|\approx|x|+|x|$ (after simplifications), which is T_{PBV}-unsatisfiable. That is, there exists no admissible T_{PBV}-interpretation for φ' and, thus, φ' is not admissibly T_{PBV}-satisfiable. However, φ' is T_{PBV}-satisfiable: consider a T_{PBV}-interpretation $\mathcal{I}$, given by $x^{\mathcal{I}}=0$. Then, $(x\circ x)^{\mathcal{I}}$ is the bit-vector 00. Since x and $x\circ x$ do not have the same bit-width, $\mathcal{I}$ may interpret $x+^{\mathcal{B}}(x\circ x)$ arbitrarily, and so we can set $(x+^{\mathcal{B}}(x\circ x))^{\mathcal{I}}$ to be again the bit-vector 0. We then get that $\mathcal{I}$ satisfies φ'.

Note that our notion of admissibility corresponds to the notion of admissibility introduced by Niemetz et al. [30]. However, its definition differs in one key aspect. In [30], admissibility was enforced by using auxiliary functions and by instantiating parametric bit-vector formulas for all possible fixed bit-widths via universal quantification over the parametric bit-widths.

Algorithm 1 Function ADM to recursively construct admissibility constraints. Symbol z denotes constants of sort Int and x constants of sort PBV. Symbol $\bullet$ ranges over symbols of Σ_{IA}, Boolean connectives, and ite_{Int}. Symbol $\diamond$ ranges over symbols in Σ_{PBV} not explicitly handled otherwise.

function $\text{ADM}(e)$
match e:

x	$\to$	$\text{BW}(e) > 0$
z	$\to$	$\top$
$\text{to-pbv}(k, t)$	$\to$	$\text{BW}(e) > 0 \wedge \text{ADM}(t)$
$\lvert t \rvert$	$\to$	$\text{ADM}(t)$
$t[i:j]$	$\to$	$0 \leq j \leq i < \text{BW}(t) \wedge \text{ADM}(t)$
$\text{ext}_z(n, t)$	$\to$	$n \geq 0 \wedge \text{ADM}(t)$
$\text{ext}_s(n, t)$	$\to$	$n \geq 0 \wedge \text{ADM}(t)$
$t_1 \circ t_2$	$\to$	$\text{ADM}(t_1) \wedge \text{ADM}(t_2)$
$\text{ite}_{\text{PBV}}(t_1, t_2, t_3)$	$\to$	$\text{BW}(t_2) \approx \text{BW}(t_3) \wedge \bigwedge_{i=1}^{3} \text{ADM}(t_i)$
$\bullet(t_1, \ldots, t_n)$	$\to$	$\bigwedge_{i=1}^{n} \text{ADM}(t_i)$
$\diamond(t_1, \ldots, t_n)$	$\to$	$(\bigwedge_{i=2}^{n} \text{BW}(t_1) \approx \text{BW}(t_i)) \wedge (\bigwedge_{i=1}^{n} \text{ADM}(t_i))$

Algorithm 2 Function BW to compute the bit-width of PBV-terms. Symbol x denotes variables of sort PBV. Symbol $\diamond$ ranges over the symbols of Σ_{PBV} of sort PBV not explicitly handled otherwise.

function $\text{BW}(e)$
match e:

x	$\to$	$\lvert x \rvert$
$\text{to-pbv}(k, t)$	$\to$	k
$t[i:j]$	$\to$	$i - j + 1$
$\text{ext}_z(n, t)$	$\to$	$\text{BW}(t) + n$
$\text{ext}_s(n, t)$	$\to$	$\text{BW}(t) + n$
$t_1 \circ t_2$	$\to$	$\text{BW}(t_1) + \text{BW}(t_2)$
$\text{ite}_{\text{PBV}}(t_1, t_2, t_3)$	$\to$	$\text{BW}(t_2)$
$\diamond(t_1, \ldots, t_n)$	$\to$	$\text{BW}(t_1)$

Bit-widths and mappings from PBV-constants to Σ_{IA}-terms were implicit in the signature, whereas in our definition of Σ_{PBV}, we explicitly include operators $\lvert \sqcup \rvert$ and to-pbv. This allows us to explicitly define admissibility conditions and to reason about symbolic bit-widths in Σ_{PBV}-formulas.

4 A Solver for Parametric Bit-vectors

In this section, we describe a solver for admissible T_{PBV}-satisfiability. Its main component is the translation of Σ_{PBV}-formulas to formulas over a signature $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$. As defined in Table 2, signature $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$ extends Σ_{IA} with two new function symbols, pow_2 and $\&^{\mathbb{N}}$.

We define a theory $T_{IA}(\text{pow}_2, \&^{\mathbb{N}})$ over $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$, where symbols pow_2 and $\&^{\mathbb{N}}$ are interpreted *arbitrarily*. Additionally, for the purpose of the translation, we consider a theory $T_{IA}(\text{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$ over $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$ in which the interpretation of pow_2 and $\&^{\mathbb{N}}$ is *fixed* and defined as follows. The term $\text{pow}_2(n)$ is interpreted as the n-th power of 2 whenever n is non-negative, and as 0 otherwise. The interpretation of $\&^{\mathbb{N}}(k, a, b)$ is defined as follows. If $k > 0$ and $a, b \in \{0, \ldots, 2^k - 1\}$, then $\&^{\mathbb{N}}(k, a, b)$ is the integer that corresponds to performing a bitwise conjunction on the unsigned bit-vector representations of a and b. That is, let a', b'

be the unsigned bit-vector representations of a and b of width k, then the interpretation of $\&^{\mathbb{N}}(k, a, b)$ is defined as $\Sigma_{i=0}^{k-1} a'[i] \cdot b'[i] \cdot 2^i$. If $k > 0$ and $a, b \notin \{0, \ldots, 2^k - 1\}$, then the interpretation of $\&^{\mathbb{N}}(k, a, b)$ is the same as $\&^{\mathbb{N}}(k, a \bmod 2^k, b \bmod 2^k)$. If $k \leq 0$, then its interpretation is 0.

▶ Remark 4. Notice that pow_2 is defined over the integers. Thus, we always interpret a negative exponent $-n$ as 0, as this is consistent with truncating the integer division in $2^{-n} \equiv \frac{1}{2^n}$ towards zero. Further, notice that $\&^{\mathbb{N}}$ defines a special case for bit-width $k \leq 0$. Our translation will never produce $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$-formulas that are satisfiable due to these corner cases because it ensures that: (1) the exponents of pow_2 are integer terms representing bit-widths and (2) integer terms that represent bit-widths are always positive. In fact, for our purposes, the special cases mentioned above could also be interpreted arbitrarily. We choose a fixed interpretation since it is easier to implement.

Starting with a Σ_{PBV}-formula, we first simplify the formula by applying rewrite rules $\mathcal{RW}_{\mathcal{B}}$, as described in Section 4.1. The rewritten Σ_{PBV}-formula is then translated to a $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$-formula using function Trans, as described in Section 4.2. Finally, the $T_{IA}(\text{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-satisfiability of the resulting $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$-formula is determined via procedure Solve$_{\star}$, described in Section 4.3.

4.1 The Parametric Bit-Vector Rewriter $\mathcal{RW}_{\mathcal{B}}$

State-of-the-art SMT solvers rely heavily on simplification techniques that are applies as a preprocessing step, before the main solver process begins. One such technique is based on term rewriting. For the theory of fixed-size bit-vectors (for which the dominant solving technique is bit-blasting, an eager reduction to propositional logic), SMT solvers implement hundreds of rewrite rules. However, rewrite rules for T_{BV} that target bit-blasting are not necessarily beneficial for alternative solving procedures. In particular, they may not be useful for our T_{PBV} procedure, which relies on a translation to integer arithmetic. Thus, some care is required in selecting rewrite rules for our procedure.

We implemented a set of rewrite rules for T_{PBV} based on rewrite rules for T_{BV} as implemented in the SMT solver cvc5 [1]. Rewrite rules implemented in cvc5 are documented in the domain-specific language RARE [33], which allows for easy adoption and lifting to T_{PBV}. We selected T_{BV}-rewrite rules for lifting to T_{PBV} based on the following principles: (i) a rewrite rule should not introduce a bitwise operator; (ii) the translation of the rewritten formula to the integers should not have more occurrences of operators mod and pow_2 than the original one. These principles are based on the fact that state-of-the-art solvers for integer arithmetic do not support bitwise operations over integers, and also offer very limited support for exponentiation. Further, mod is non-linear and one of the most expensive operations for integer arithmetic procedures. Interestingly, following these guiding principles, some T_{BV}-rewrite rules are useful for T_{PBV} when applied in the *opposite* direction.

The rewrite rules that we have implemented are presented in Section A.

▶ **Example 5.** Consider a T_{BV}-rule that rewrites $(x \,\&\, y)[i:j]$ to $(x[i:j]) \,\&\, (y[i:j])$. This is useful for bit-blasting since it reduces the size of the bit-level representation. However, for our translation to integer arithmetic, extractions are expensive as they introduce div, mod and pow_2 terms. Thus, we did not lift this rule to T_{PBV} but included its right-to-left variant in $\mathcal{RW}_{\mathcal{B}}$ instead.

Algorithm 3 Translation function TRANS and conversion function CONV. We use x for PBV variables, z for Int variables, $\mathsf{uts}(k, z)$ for $2 \cdot (z \bmod \text{pow}_2(k-1)) - z$, $\bowtie$ for any operator in $\{<, \leq, >, \geq\}$, and $\bullet$ for ite, logical connectives and symbols in Σ_{IA}.

function TRANS(φ)
  return CONV($\varphi \wedge$ RANGE(φ) $\wedge$ ADM(φ))

function CONV(e)
  match e:

$\underline{z}$	$\to$	$\underline{z}$
$\underline{\lvert t \rvert}$	$\to$	$\underline{\kappa(t)}$
$\underline{\text{to-pbv}(k, t)}$	$\to$	$\underline{\text{CONV}(t) \bmod \text{pow}_2(k)}$
$\underline{x}$	$\to$	$\underline{\chi(x)}$
$t_1 \approx t_2$	$\to$	$\text{CONV}(t_1) \approx \text{CONV}(t_2)$
$t_1 \bowtie_u t_2$	$\to$	$\text{CONV}(t_1) \bowtie \text{CONV}(t_2)$
$t_1 \bowtie_s t_2$	$\to$	$\mathsf{uts}(\kappa(t_1), \text{CONV}(t_1)) \bowtie \mathsf{uts}(\kappa(t_2), \text{CONV}(t_2))$
$t_1 +^{\mathcal{B}} t_2$	$\to$	$(\text{CONV}(t_1) + \text{CONV}(t_2)) \bmod \text{pow}_2(\kappa(t_1))$
$t_1 -^{\mathcal{B}} t_2$	$\to$	$(\text{CONV}(t_1) - \text{CONV}(t_2)) \bmod \text{pow}_2(\kappa(t_1))$
$t_1 \cdot^{\mathcal{B}} t_2$	$\to$	$(\text{CONV}(t_1) \cdot \text{CONV}(t_2)) \bmod \text{pow}_2(\kappa(t_1))$
$t_1 \text{div}^{\mathcal{B}} t_2$	$\to$	$\text{ite}(\text{CONV}(t_2) \approx 0, \text{pow}_2(\kappa(t_1)) - 1, \text{CONV}(t_1) \text{ div } \text{CONV}(t_2))$
$t_1 \text{mod}^{\mathcal{B}} t_2$	$\to$	$\text{ite}(\text{CONV}(t_2) \approx 0, \text{CONV}(t_1), \text{CONV}(t_1) \bmod \text{CONV}(t_2))$
$\sim t$	$\to$	$\text{pow}_2(\kappa(t)) - (\text{CONV}(t) + 1)$
$-^{\mathcal{B}} t$	$\to$	$(\text{pow}_2(\kappa(t)) - \text{CONV}(t)) \bmod \text{pow}_2(\kappa(t))$
$t_1 \ll t_2$	$\to$	$(\text{CONV}(t_1) \cdot \text{pow}_2(\text{CONV}(t_2))) \bmod \text{pow}_2(\kappa(t_1))$
$t_1 \gg t_2$	$\to$	$\underline{\text{CONV}(t_1) \text{ div } \text{pow}_2(\text{CONV}(t_2))}$
$t_1 \,\&\, t_2$	$\to$	$\underline{\&^{\mathbb{N}}(\kappa(t_1), \text{CONV}(t_1), \text{CONV}(t_2))}$
$t_1 \mid t_2$	$\to$	$\underline{\text{CONV}(t_1) + \text{CONV}(t_2) - \&^{\mathbb{N}}(\kappa(t_1), \text{CONV}(t_1), \text{CONV}(t_2))}$
$t_1 \oplus t_2$	$\to$	$\underline{\text{CONV}(t_1) + \text{CONV}(t_2) - 2 \cdot \&^{\mathbb{N}}(\kappa(t_1), \text{CONV}(t_1), \text{CONV}(t_2))}$
$t_1 \circ t_2$	$\to$	$\text{CONV}(t_1) \cdot \text{pow}_2(\kappa(t_2)) + \text{CONV}(t_2)$
$\underline{t[i:j]}$	$\to$	$\underline{(\text{CONV}(t) \text{ div } \text{pow}_2(j)) \bmod \text{pow}_2(i-j+1)}$
$\underline{\text{ext}_z(n, t)}$	$\to$	$\underline{\text{CONV}(t)}$
$\underline{\text{ext}_s(n, t)}$	$\to$	$\underline{\text{ite}(\text{CONV}(t[\kappa(t)-1]) \approx 1,}$
		$\underline{(\text{pow}_2(n) - 1) \cdot \text{pow}_2(\kappa(t)) + \text{CONV}(t), \text{CONV}(t))}$
$\underline{\bullet(t_1, \ldots, t_n)}$	$\to$	$\underline{\bullet(\text{CONV}(t_1), \ldots, \text{CONV}(t_n))}$

4.2 The Translation Function Trans

After obtaining the rewritten Σ_{PBV}-formula φ, we translate it into a $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$-formula φ' such that φ is admissibly T_{PBV}-satisfiable iff φ' is $T_{IA}(\text{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-satisfiable. The translation function TRANS is given in Algorithms 3 and 4. It is based on the one in [30], with changes underlined.

Prior to the actual conversion step, Σ_{PBV}-formula φ is augmented, conjunctively, with the admissibility constraints ADM(φ), constructed as described in Algorithm 1, as well as the *range and size constraints* RANGE(φ). The latter are defined over integer constants that represent either parametric bit-vector constants or symbolic bit-widths, as described in Algorithm 4. The latter traverses the formula and adds constraints for the range of the integer terms representing bit-vector terms, according to their original bit-widths.

The augmented Σ_{PBV}-formula is then converted via function CONV into an equisatisfiable $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$-formula as defined in Algorithm 3.

Algorithm 4 Function RANGE to recursively construct size and range constraints. We use x for PBV variables, z for Int variables, and $\bullet$ for ite, logical connectives and symbols in Σ_{PBV}.

function RANGE(e)
  match e:

x	$\rightarrow$	$0 \leq \chi(x) < \text{pow}_2(\kappa(x))$
$\underline{\lvert t \rvert}$	$\rightarrow$	$\underline{e \approx \text{BW}(t) \wedge \text{RANGE}(t)}$
z	$\rightarrow$	$\top$
$\bullet(t_1, \ldots, t_n)$	$\rightarrow$	$\bigwedge_{i=1}^{n} \text{RANGE}(t_i)$

We assume a one-to-one mapping χ from variables of sort PBV to variables of sort Int, as well as a one-to-one mapping κ from PBV-terms to variables of sort Int, such that their images are disjoint. Intuitively, χ translates variables of sort PBV to corresponding variables of sort Int while κ translates terms of the form $|t|$ to fresh integer variables representing the size of the parametric bit-vector term t. Compared to our previous work [30], our new translation handles the new operators $|\sqcup|$ and to-pbv, improves the encodings of the operators $\gg$, $|$, and $\oplus$ to eliminate applications of mod, and adds support for the operators $\sqcup[\sqcup : \sqcup]$, ext_s, and ext_z. Notice that the encoding of $\gg_a$ is not explicitly described, as that operator can be expressed in terms of $\gg$ and ite.

Following Zohar et al. [39], we do not handle $|$ and $\oplus$ natively. Instead, we eliminate them by means of $\&$, $+^{\mathcal{B}}$ and $-^{\mathcal{B}}$, based on the following identities derived from [18]: $x \mid y \approx (x +^{\mathcal{B}} y) -^{\mathcal{B}} (x \,\&\, y)$ and $x \oplus y \approx (x \mid y) -^{\mathcal{B}} (x \,\&\, y)$. This elimination is embedded in the translation function and improves upon the original elimination in [39] by avoiding the introduction of additional mod operations.

▶ **Example 6.** Consider again the formula $\varphi := y \approx z \circ w$ from Example 3. Then, $\text{TRANS}(\varphi) = \text{CONV}(\varphi \wedge \text{RANGE}(\varphi) \wedge \text{ADM}(\varphi))$. Now, $\text{CONV}(\varphi)$ is $y' \approx z' \cdot \text{pow}_2(k_w) + w'$, where $y' = \chi(y)$, $z' = \chi(z)$, $w' = \chi(w)$, and $k_w = \kappa(w)$. Further, $\text{CONV}(\text{RANGE}(\varphi))$ is $0 \leq y' < \text{pow}_2(k_y) \wedge 0 \leq z' < \text{pow}_2(k_z) \wedge 0 \leq w' < \text{pow}_2(k_w)$, where $k_y = \kappa(y)$ and $k_z = \kappa(z)$. The result of $\text{ADM}(\varphi)$ is given in Example 3, and then CONV replaces $|y|$, $|z|$, and $|w|$ by k_y, k_z, and k_w, respectively.

Our translation function TRANS makes heavy use of mod and pow_2 operations. In practice, some of these applications can be safely eliminated to optimize the encoding. For example, consider the term $(x +^{\mathcal{B}} x) +^{\mathcal{B}} x$, where $\kappa(x) = k$. The result of CONV on this term is $(((x' + x') \bmod 2^k) + x') \bmod 2^k$, where $x' = \chi(x)$. Instead, we directly translate it to $((x' + x') + x') \bmod 2^k$.

4.3 The $T_{IA}(\text{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$-Solver

After translating a Σ_{PBV}-formula to a $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$-formula, we check if it is $T_{IA}(\text{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$-satisfiable with a dedicated CEGAR-style procedure $\text{SOLVE}_\star$, which iteratively refines over-approximations of pow_2 and $\&^{\mathbb{N}}$ as given in Algorithm 5. This procedure takes an approach similar to the incremental linearization of non-linear arithmetic constraints [9,10]. It relies on a set of quantifier-free lemmas $\mathcal{L}$ which fully specify the semantics of operators pow_2 and $\&^{\mathbb{N}}$. The set is constructed by instantiating the (implicitly) universally quantified lemma schemas $\mathcal{L}_{\text{pow}_2}$, shown in Table 3, and $\mathcal{L}_{\&^{\mathbb{N}}}$, shown in Table 4, for all pow_2-terms and $\&^{\mathbb{N}}$-terms in φ. The instantiations of the lemmas also depend on the values of variables in the current interpretation $\mathcal{I}$. Note that when computing the instantiation for pow_2-lemmas, we also include pow_2-terms occurring in lemmas for $\&^{\mathbb{N}}$.

Algorithm 5 Procedure for $T_{IA}(\text{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$-satisfiability. We assume SOLVE is a procedure for $T_{IA}(\text{pow}_2, \&^{\mathbb{N}})$-satisfiability that returns "sat" or "unsat" and an interpretation $\mathcal{I}$ for satisfiable formulas. For a set X, $\mathit{Terms}(X)$ denotes the set of terms occurring in X.

function SOLVE$_\star(\varphi)$
  $\Gamma \leftarrow \{\varphi\}$
  loop
    $\mathit{result}, \mathcal{I} \leftarrow$ SOLVE(Γ)
    if *result* is "unsat" **return** "unsat"
    $\mathcal{L} \leftarrow \mathcal{L}_{\&^{\mathbb{N}}}(\{\&^{\mathbb{N}}(k,t_1,t_2) \mid \&^{\mathbb{N}}(k,t_1,t_2) \in \mathit{Terms}(\Gamma)\}, \mathcal{I})$
    $\mathcal{L} \leftarrow \mathcal{L} \cup \mathcal{L}_{\text{pow}_2}(\{\text{pow}_2(t) \mid \text{pow}_2(t) \in \mathit{Terms}(\mathcal{L} \cup \Gamma)\}, \mathcal{I})$
    if $\mathcal{I} \models \mathcal{L}$ **return** "sat"
    $\Gamma \leftarrow \Gamma \cup \{\psi \mid \psi \in \mathcal{L} \text{ and } \mathcal{I} \not\models \psi\}$

The procedure further assumes the availability of a subprocedure SOLVE to determine the $T_{IA}(\text{pow}_2, \&^{\mathbb{N}})$-satisfiability of a set of $\Sigma_{IA}(\text{pow}_2, \&^{\mathbb{N}})$-formulas Γ. Recall that in $T_{IA}(\text{pow}_2, \&^{\mathbb{N}})$, symbols pow_2 and $\&^{\mathbb{N}}$ are uninterpreted and thus, Γ is an over-approximation of φ in $T_{IA}(\text{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$. If SOLVE determines the unsatisfiability of Γ, we conclude with "unsat". If it determines its satisfiability, we conclude with "sat" only if the resulting interpretation $\mathcal{I}$ also satisfies all lemmas in $\mathcal{L}$. Otherwise, we refine Γ by adding to it all lemmas that are not satisfied by $\mathcal{I}$.

The set $\mathcal{L}_{\text{pow}_2}$ in Table 3 includes 4 lemma schemas from [30], which describe basic properties of the pow_2 operator. In addition, we consider the following three new lemmas. Lemma neg considers negative inputs, for which pow_2 is defined to be interpreted as 0 in $T_{IA}(\text{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$. Lemma bound strengthens a lemma from recent work proposing a CEGAR-style approach for a new SMT theory for integer arithmetic with exponentiation [14]. From there, we get that when $x \geq 3$, then $2 \cdot x + 1$ is a lower bound for $\text{pow}_2(x)$. We notice that when $x \geq 7$, we have $2 \cdot x^2$ as a tighter lower bound. In order to avoid non-linear multiplications, we linearize this bound via concrete values $v = x^{\mathcal{I}}$. We further include instances of value with $v = x^{\mathcal{I}}$, to block concrete model values.

The set of lemma schemas $\mathcal{L}_{\&^{\mathbb{N}}}$ in Table 4 includes 8 lemmas that were introduced in [30] and capture basic properties of the $\&^{\mathbb{N}}$ operator. In addition, we consider 4 lemmas that did not appear in [30]. Lemma empty captures the corner case when the bit-width is non-positive. Lemmas lsb and one capture the cases where one of the arguments is even or 1. Note that due to lemma sym, it is sufficient to consider only one variant of each lemma. Lemma $\text{sum}_{\geq}$ is based on the fact that for a fixed bit-width n, $\&^{\mathbb{N}}(n,x,y)$ is defined as $\Sigma_{i=0}^{n-1} \mathsf{ex}_i(x) \cdot \mathsf{ex}_i(y) \cdot 2^i$, with $\mathsf{ex}_i(x)$ defined as $(x \text{ div } 2^i) \bmod 2$ as in Table 4. We observe that for a bit-width $m \geq n$ with $0 \leq x < 2^n$ and $0 \leq y < 2^n$, the equation $\&^{\mathbb{N}}(m,x,y) \approx \&^{\mathbb{N}}(n,x,y)$ holds – consider, for example, $\&^{\mathbb{N}}(5,1,1) \approx \&^{\mathbb{N}}(4,1,1)$. Lemma $\text{sum}_{\geq}$ is a generalization of this observation. It is instantiated with $v = k^{\mathcal{I}}$.

▶ **Example 7.** Consider the formula $x \,\&\, x \approx x$. Its translation will be $\&^{\mathbb{N}}(k,x,x) \approx x \bmod \text{pow}_2(k)$, with the addition of the appropriate admissibility and range constraints. Then lemma div will be instantiated in set $\mathcal{L}$ as $k \geq 0 \Rightarrow k \text{ div } \text{pow}_2(k) \approx 0$. Lemma min will be instantiated as $x \approx 0 \Rightarrow \&^{\mathbb{N}}(k,x,x) \approx 0$. In contrast, the approach of [30] introduces the universal closure of such lemmas, instead of specific instantiations.

Note that procedure SOLVE$_\star$ resembles the lazy approach of [39], with several key differences. First, since we are dealing with parametric bit-widths, our algorithm does not necessarily terminate, as the set of possible bit-widths is unbounded. Second, our algorithm

Table 3 The set of lemma schemas defined by $\mathcal{L}_{\text{pow}_2}(S, \mathcal{I})$, over a set of pow_2-terms S and an interpretation $\mathcal{I}$. The variables x and y in the formulas above are instantiated for each pow_2-term in S. Symbol v is a schema variable standing for an arbitrary numeral. It is instantiated with $v := x^{\mathcal{I}}$.

Name	Lemma Schema	Source
	$\text{pow}_2(x) \in S$	
positive	$x \geq 0 \Rightarrow \text{pow}_2(x) > 0$	[30]
even	$x \geq 1 \Rightarrow \text{pow}_2(x) \bmod 2 \approx 0$	[30]
div	$x \geq 0 \Rightarrow x \operatorname{div} \text{pow}_2(x) \approx 0$	[30]
neg	$x < 0 \Rightarrow \text{pow}_2(x) \approx 0$	new
bound	$(7 \leq v \wedge v \leq x) \Rightarrow v \cdot x + v^2 < \text{pow}_2(x)$	new
value	$(0 \leq x \wedge x \approx v) \Rightarrow \text{pow}_2(x) \approx 2^v$	new
	$\text{pow}_2(x) \in S$, $\text{pow}_2(y) \in S$	
monotonicity	$(0 \leq x \wedge x < y) \Rightarrow \text{pow}_2(x) < \text{pow}_2(y)$	[30]

Table 4 The set of lemma schemas defined by $\mathcal{L}_{\&^{\mathbb{N}}}(S, \mathcal{I})$ over a set of $\&^{\mathbb{N}}$-terms S. The variables k, x, z, and w in the formulas above are instantiated for each $\&^{\mathbb{N}}$-term in S. We use the expression $\langle t \rangle_k$ to denote $0 \leq t < \text{pow}_2(k)$, the expression $\min(x, y)$ to abbreviate $\text{ite}(x < y, x, y)$, and $\mathsf{ex}_i(x)$ for $(x \operatorname{div} 2^i) \bmod 2$. Symbol v is a schema variable standing for an arbitrary numeral. It is instantiated with $v := k^{\mathcal{I}}$.

Name	Lemma Schema	Source
	$\&^{\mathbb{N}}(k, x, y) \in S$	
base	$(k > 0 \wedge x \approx 1 \wedge y \approx 1) \Rightarrow \&^{\mathbb{N}}(k, x, y) \approx 1$	[30]
max	$(k > 0 \wedge \langle x \rangle_k \wedge y \approx \text{pow}_2(k) - 1) \Rightarrow \&^{\mathbb{N}}(k, x, y) \approx x$	[30]
min	$y \approx 0 \Rightarrow \&^{\mathbb{N}}(k, x, y) \approx 0$	[30]
idem	$(k > 0 \wedge \langle x \rangle_k \wedge x \approx y) \Rightarrow \&^{\mathbb{N}}(k, x, y) \approx x$	[30]
contra	$(x + y) \bmod \text{pow}_2(k) \approx \text{pow}_2(k) - 1 \Rightarrow \&^{\mathbb{N}}(k, x, y) \approx 0$	[30]
range	$0 \leq x \wedge 0 \leq y \Rightarrow 0 \leq \&^{\mathbb{N}}(k, x, y) \leq \min(x, y)$	[30]
empty	$k \leq 0 \Rightarrow \&^{\mathbb{N}}(k, x, y) \approx 0$	new
lsb	$x \bmod 2 \approx 0 \Rightarrow \&^{\mathbb{N}}(k, x, y) \bmod 2 \approx 0$	new
one	$(k > 0 \wedge y \approx 1) \Rightarrow \&^{\mathbb{N}}(k, x, y) \approx x \bmod 2$	new
$\text{sum}_{\geq}$	$(k \geq v \wedge v > 0 \wedge \langle x \rangle_v \wedge \langle y \rangle_v) \Rightarrow \&^{\mathbb{N}}(k, x, y) \approx \Sigma_{i=0}^{v-1} \mathsf{ex}_i(x) \cdot \mathsf{ex}_i(y) \cdot 2^i$	new
	$\&^{\mathbb{N}}(k, x, z) \in S$, $\&^{\mathbb{N}}(k, y, w) \in S$	
sym	$(x \approx w \wedge y \approx z) \Rightarrow \&^{\mathbb{N}}(k, x, y) \approx \&^{\mathbb{N}}(k, z, w)$	[30]
diff	$(k > 0 \wedge z \approx w \wedge x \not\approx y \wedge \langle x \rangle_k \wedge \langle y \rangle_k \wedge \langle z \rangle_k) \Rightarrow (\&^{\mathbb{N}}(k, x, z) \not\approx y \vee \&^{\mathbb{N}}(k, y, w) \not\approx x)$	[30]

not only lazily handles operator $\&^{\mathbb{N}}$, but also pow_2. Operator pow_2 did not occur when int-blasting in [39] since there, all exponents were concrete constants. Thus, exponentiation was simply evaluated. Finally, our sets of lemmas for $\&^{\mathbb{N}}$ and pow_2 expand both the axioms from [30] and the lemmas from [39].

Putting all components described in this section together, we obtain an incomplete but sound procedure $\textsc{Solve}_\star(\textsc{Trans}(\mathcal{RW}_{\mathcal{B}}(\varphi)))$ for admissible T_{PBV}-satisfiability.

▶ **Theorem 8.** *Let φ be a Σ_{PBV}-formula and let $\varphi' = \textsc{Trans}(\mathcal{RW}_{\mathcal{B}}(\varphi))$. If $\textsc{Solve}_\star(\varphi')$ terminates, then φ is admissibly T_{PBV}-satisfiable if and only if $\textsc{Solve}_\star(\varphi')$ returns "sat".*

4.4 Proof of Theorem 8

The correctness argument for Theorem 8 is similar to those from [30, 39]: assuming a T_{PBV}-interpretation that satisfies a Σ_{PBV}-formula φ, while also being admissible w.r.t. φ, we can construct a $T_{IA}(\text{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-interpretation that satisfies the translation of φ, as well as the relevant lemmas from Tables 3 and 4, and vice versa. Compared to [30], the auxiliary admissibility condition that was assumed is now replaced by the satisfaction of $\textsc{Adm}(\varphi)$. What is left to show is:

1. The new translation of $\gg$ is correct.
2. The eliminations of $|$ and $\oplus$ are correct.
3. The treatment of extraction and extensions is correct.
4. The rules of the $\mathcal{RW}_{\mathcal{B}}$ rewriter are T_{PBV}-valid.
5. All lemmas of Tables 3 and 4 are $T_{IA}(\text{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-valid.

We prove the first item in Section 4.4.1 and the second in Section 4.4.2. The third follows directly from the SMT-LIB standard. For the fourth, notice that we only use rewrite rules that are already used in cvc5 and are bit-width independent. For their correctness proofs, see, e.g., [19]. For the fifth, most lemmas are taken from [30]. Of the new lemmas that we add, correctness is trivial for lemmas neg, value, empty, lsb and one. We prove the correctness of the remaining lemmas in Section 4.4.3.

4.4.1 Correctness of Translation of $\gg$

It suffices to show that in every $T_{IA}(\text{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-interpretation that satisfies the formulas $\textsc{Conv}(x) \geq 0$, $\text{pow}_2(\textsc{Conv}(y)) > 0$, and $\text{pow}_2(k) > \textsc{Conv}(x)$, the expressions $\textsc{Conv}(x)$ div $\text{pow}_2(\textsc{Conv}(y))$ and $(\textsc{Conv}(x)$ div $\text{pow}_2(\textsc{Conv}(y)))$ mod $\text{pow}_2(k)$ are interpreted the same. This holds, as every such interpretation also satisfies $\textsc{Conv}(x)$ div $\text{pow}_2(\textsc{Conv}(y)) \leq \textsc{Conv}(x) < \text{pow}_2(k)$.

4.4.2 Correctness of Elimination of $|$ and $\oplus$

We start with the following lemma:

▶ **Lemma 9.** *Suppose $k > 0$, $x \geq 0$ and $y < 2^k$. Then, $0 \leq x + y - \&^{\mathbb{N}}(k, x, y) < 2^k$ and $0 \leq x + y - 2 \cdot \&^{\mathbb{N}}(k, x, y) < 2^k$.*

Proof. In this proof we treat $\&^{\mathbb{N}}$ and pow_2 as arithmetic operators with a fixed interpretation as given to them by all $T_{IA}(\text{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$-interpretations. This saves us from superscripting interpretations to these symbols, which is not needed since their interpretations are fixed. We consider the required inequalities separately:

- $0 \leq x + y - \&^{\mathbb{N}}(k, x, y) < 2^k$:
  1. By the semantics of $\&^{\mathbb{N}}$, we have $\&^{\mathbb{N}}(k, x, y) \leq x$. Since $y \geq 0$, we get $\&^{\mathbb{N}}(k, x, y) \leq x+y$. Therefore, $x + y - \&^{\mathbb{N}}(k, x, y) \geq 0$.
  2. Next, we show $x + y - \&^{\mathbb{N}}(k, x, y) < 2^k$. Since $0 \leq x, y < 2^k$, we have: $x = \Sigma_{i=0}^{k-1} 2^i \cdot \mathsf{ex}_i(x)$ and $y = \Sigma_{i=0}^{k-1} 2^i \cdot \mathsf{ex}_i(y)$, where for every $z \in \mathbb{N}$, $\mathsf{ex}_i(z) = (z \text{ div } 2^i) \bmod 2$. By the semantics of $\&^{\mathbb{N}}$, we have $\&^{\mathbb{N}}(k, x, y) = \Sigma_{i=0}^{k-1} 2^i \cdot \text{ite}(\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 1, 1, 0)$. Notice that we can also have $y = \Sigma_{i=0}^{k-1} 2^i \cdot \text{ite}(\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 1, 1, 0) + \Sigma_{i=0}^{k-1} 2^i \cdot \text{ite}(\mathsf{ex}_i(x) = 0 \wedge \mathsf{ex}_i(y) = 1, 1, 0)$.
     Now, we clearly have that $x+y-\&^{\mathbb{N}}(k, x, y)$ equals $\Sigma_{i=0}^{k-1} 2^i \cdot \mathsf{ex}_i(x) + \Sigma_{i=0}^{k-1} 2^i \cdot \text{ite}(\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 1, 10) + \Sigma_{i=0}^{k-1} 2^i \cdot \text{ite}(\mathsf{ex}_i(x) = 0 \wedge \mathsf{ex}_i(y) = 1, 1, 0) - \Sigma_{i=0}^{k-1} 2^i \cdot \text{ite}(\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 1, 1, 0)$, which is: $\Sigma_{i=0}^{k-1} 2^i \cdot \mathsf{ex}_i(x) + \Sigma_{i=0}^{k-1} 2^i \cdot \text{ite}(\mathsf{ex}_i(x) = 0 \wedge \mathsf{ex}_i(y) = 1, 1, 0)$. This is equal to $\Sigma_{i=0}^{k-1} 2^i \cdot \text{ite}(\mathsf{ex}_i(x) = 1, 1, 0) + \Sigma_{i=0}^{k-1} 2^i \cdot \text{ite}(\mathsf{ex}_i(x) = 0 \wedge \mathsf{ex}_i(y) = 1, 1, 0)$. Clearly, this must be at most $\Sigma_{i=0}^{k-1} 2^i \cdot \text{ite}(\mathsf{ex}_i(x) = 1, 1, 0) + \Sigma_{i=0}^{k-1} 2^i \cdot \text{ite}(\mathsf{ex}_i(x) = 0, 1, 0)$, which equals $\Sigma_{i=0}^{k-1} 2^i$, which must be strictly smaller than 2^k.

- $0 \leq x + y - 2 \cdot \&^{\mathbb{N}}(k, x, y) < 2^k$:
  1. By the semantics of $\&^{\mathbb{N}}$, we have $\&^{\mathbb{N}}(k, x, y) \leq x$ and $\&^{\mathbb{N}}(k, x, y) \leq y$. Therefore, $x + y \geq 2 \cdot \&^{\mathbb{N}}(k, x, y)$, and so $x + y - 2 \cdot \&^{\mathbb{N}}(k, x, y) \geq 0$.
  2. For the other direction, since $\&^{\mathbb{N}}(k, x, y) \geq 0$ for all k, x, y, we have: $x + y - 2 \cdot \&^{\mathbb{N}}(k, x, y) \leq x + y - \&^{\mathbb{N}}(k, x, y)$. By the inequality item, $x + y - \&^{\mathbb{N}}(k, x, y) < 2^k$. ◀

We now justify the elimination of operator $|$ and $\oplus$, relying on the following equations from [18]:

$$x +^{\mathcal{B}} y \approx (x \,\&\, y) +^{\mathcal{B}} (x \mid y) \qquad (1)$$

$$x \oplus y \approx (x \mid y) -^{\mathcal{B}} (x \,\&\, y) \qquad (2)$$

For $|$, we get get: $x \mid y \approx (x +^{\mathcal{B}} y) -^{\mathcal{B}} (x \,\&\, y)$. Translating the right-hand side of this equation to integer arithmetic with $\&^{\mathbb{N}}$, we get $((x + y) \bmod 2^k - (\&^{\mathbb{N}}(k, x, y))) \bmod 2^k$, where k is the bit-width of x and y. This can be simplified to $((x + y) - (\&^{\mathbb{N}}(k, x, y))) \bmod 2^k$. In order to show an equality to $((x + y) - (\&^{\mathbb{N}}(k, x, y)))$, which is our actual translation, it is left to show that $0 \leq ((x + y) - (\&^{\mathbb{N}}(k, x, y))) < 2^k$, as stated in the first item of Lemma 9.

The justification for the elimination of operator $\oplus$ is similar. Combining (1) and (2), we get that $x \oplus y \approx ((x +^{\mathcal{B}} y) -^{\mathcal{B}} (x \,\&\, y)) -^{\mathcal{B}} (x \,\&\, y)$ and thus $x \oplus y \approx x +^{\mathcal{B}} y -^{\mathcal{B}} (2 \cdot^{\mathcal{B}} x \,\&\, y)$. Translating the right-hand side of this equation to integer arithmetic with $\&^{\mathbb{N}}$, we get $((x + y \bmod 2^k) - (2 \cdot \&^{\mathbb{N}}(k, x, y) \bmod 2^k)) \bmod 2^k$, which can be simplified to $(x + y - 2 \cdot \&^{\mathbb{N}}(k, x, y)) \bmod 2^k$. To prove that this is the same as $(x + y - 2 \cdot \&^{\mathbb{N}}(k, x, y))$, it suffices to prove that $0 \leq (x + y - 2 \cdot \&^{\mathbb{N}}(k, x, y)) < 2^k$, as stated in the second item of Lemma 9.

4.4.3 Correctness of Lemmas

The following lemmas prove the validity of our lemmas bound and $\text{sum}_{\geq}$.

▶ **Lemma 10.** *Suppose* $v \geq 7$. *Then* $x \geq v \Rightarrow v \cdot x + v^2 < \text{pow}_2(x)$.

Proof. We first prove that $x \geq 7 \Rightarrow 2 \cdot x^2 < 2^x$ by induction on x. If $x = 7$ then this trivially holds. Assume that this holds for some x. Then, $2 \cdot (x + 1)^2 = 2 \cdot x^2 + 4 \cdot x + 2$. By the induction hypothesis, $2 \cdot x^2 < 2^x$. Also, $4 \cdot x + 2 < 2 \cdot x^2 < 2^x$ when $x \geq 7$. Hence we get $2 \cdot (x + 1)^2 < 2^{x+1}$. Let $v \geq 7$ and suppose $x \geq v$. Then, $v \cdot x \leq x^2$ and $v^2 \leq x^2$, and so $v \cdot x + v^2 \leq 2 \cdot x^2 < 2^x$. ◀

▶ **Lemma 11.** *If* $k \geq v > 0$ *then* $(\langle x \rangle_v \wedge \langle y \rangle_v) \Rightarrow \&^{\mathbb{N}}(k, x, y) \approx \Sigma_{i=0}^{v-1} \mathsf{ex}_i(x) \cdot \mathsf{ex}_i(y) \cdot 2^i$.

Proof. We always have $\&^{\mathbb{N}}(k, x, y) \approx \Sigma_{i=0}^{k-1} \mathsf{ex}_i(x) \cdot \mathsf{ex}_i(y) \cdot 2^i$. Since $k \geq v$ and $0 \leq x, y < 2^v$, we have that $\mathsf{ex}_i(x) = \mathsf{ex}_i(y) = 0$ for every $v \leq i \leq k - 1$, and thus $\mathsf{ex}_i(x) \cdot \mathsf{ex}_i(y) \cdot 2^i = 0$. ◀

5 Evaluation

We have implemented our approach by extending two tools: the generic SMT solver API smt-switch [21] and the SMT solver cvc5 [1]. In smt-switch, we extended the SMT-LIB parser with support for parsing Σ_{PBV}-formulas, and implemented our rewriter $\mathcal{RW}_{\mathcal{B}}$ (Section 4.1) and the translation Trans (Algorithm 3). In cvc5, we implemented our procedure Solve$_\star$ (Algorithm 5) by extending cvc5's non-linear arithmetic module with two subsolvers to lazily handle pow_2 and $\&^{\mathbb{N}}$. The subsolver for $\&^{\mathbb{N}}$ generalizes the $\&^{\mathbb{N}}$-subsolver in [39], which was only able to reason over $\&^{\mathbb{N}}$ with fixed bit-widths, by allowing reasoning over symbolic bit-widths.

Note that the handling of Σ_{PBV}-formulas and their translation is entirely implemented in smt-switch. Externalizing the Σ_{PBV}-translation pipeline enables the use of our approach in combination with other back-end solvers that support reasoning over pow_2 and $\&^{\mathbb{N}}$ operators.

5.1 Benchmarks

The SMT-LIB standard [2, 4] does not yet define a theory for parametric bit-vectors. Consequently, benchmarks encoding problems over parametric bit-vectors are not yet part of the SMT-LIB benchmark library. Thus, we evaluate our techniques utilizing a wide range of benchmark sets, originating from various sources, as detailed below.

***alive* (200 benchmarks).** Set *alive* consists of verification conditions for compiler optimizations that are generated by Alive [20]. The evaluation of [30] included 180 such conditions. Here, we include 20 additional conditions that were not supported by the implementation of [30]. Among the new conditions, 17 involve multiple bit-widths and 3 include terms of the form to-pbv(k, k), representing a bit-vector of length k whose unsigned integer value is k.

***ic* (180 benchmarks).** Set *ic* consists of benchmarks to verify the correctness of the *invertibility conditions* formalized in Niemetz et al. [27], which utilizes them for quantifier instantiation. They are also instrumental to the local search procedure of [27]. The evaluation of [30] included correctness checks for 160 out of 180 conditions. Here, we add the remaining 20 checks that encode invertibility conditions over concatenation, which were not supported by the implementation of [30].

***rewrite* (1500 benchmarks).** Set *rewrite* consists of the bit-width independent correctness checks of rewrite rule candidates for the theory of fixed-size bit-vectors considered in [30]. They were automatically generated using CVC4SY [36], a Syntax-Guided Synthesis (SyGuS) engine.

***syrew* (1500 benchmarks).** Set *syrew* consists of bit-width independent versions of the equivalence checks of T_{BV}-terms from [31], which were enumerated by the SyGuS engine of cvc5. There, these checks were instantiated for large bit-widths to evaluate the performance of T_{BV}-solvers.

***lemmas* (70 benchmarks).** Set *lemmas*, also originating from [31], consists of 70 refinement lemmas describing properties of arithmetic bit-vector operators for a CEGAR-style procedure for T_{BV} implemented in the SMT solver Bitwuzla [25]. These lemmas were verified to be correct up to bit-width 256 [31] but are required to be correct for arbitrarily large bit-widths. This benchmark set encodes bit-width independent correctness checks of these lemmas. Note that two benchmarks involve multiple bit-widths, and thus cannot be handled by the implementation of [30].

***icfb* (46 benchmarks).** Set *icfb* originates from a local search procedure [24] that generalizes the technique from [27] and defines invertibility conditions (and conditions for a weaker notion of invertibility called consistency) over ternary bit-vectors. These conditions were verified to be correct up to bit-width 65 but are required to hold for any bit-width. This benchmark set consists of bit-width independent correctness checks of the conditions, of which 16 benchmarks involve multiple bit-widths. Note that some conditions cannot be encoded to T_{PBV} due to the occurrence of bit-width dependent functions that require counting, e.g., the counting of leading zeroes, and were thus excluded from this set. Further note that in the original publication [24], one of the invertibility conditions for operator $<_{\mathrm{u}}$ was given incorrectly (the implementation and verification of the rule used the correct definition) [26]. We include both versions for this condition.

Table 5 Configurations considered in our experimental setup.

Feature	BASELINE	EAGER	PBV
Target Theory	$T_{IA}(\text{pow}_2, \&^{\mathbb{N}}, \|^{\mathbb{N}}, \oplus^{\mathbb{N}})$	$T_{IA}(\text{pow}_2, \&^{\mathbb{N}})$	$T_{IA}(\text{pow}_{2\star}, \&^{\mathbb{N}}_{\star})$
Multiple Bit-widths	✗	✓	✓
Lazy pow_2	✗	✗	✓
Lazy $\&^{\mathbb{N}}$	✗	✗	✓
$\|$*-elimination*	✗	✓	✓
$\oplus$*-elimination*	✗	✓	✓
$\gg$ *without* mod	✗	✓	✓
New lemmas for $\&^{\mathbb{N}}$	✗	✓	✓
New lemmas for pow_2	✗	✓	✓
No redundant axioms	✗	✓	✓
$\mathcal{RW}_{\mathcal{B}}$	✗	✓	✓
mod*-reduction*	✗	✓	✓

***mut* (9442 benchmarks).** The majority of the benchmarks in the sets above are unsatisfiable. For a more thorough evaluation on satisfiable benchmarks, we created a benchmark set *mut* by mutating the benchmarks from the other sets. This realistically mimics the introduction of bugs in verification conditions, e.g., by using a bitwise disjunction instead of conjunction. Let (f, g) be a pair of Σ_{PBV}-terms or function symbols with the same arity, and let φ be a Σ_{PBV}-formula in which f occurs. A mutation of φ is obtained from (f, g) by replacing the first occurrence of f by g. We generated mutations for pairs $(f, g) \in \{(\&, |), (+^{\mathcal{B}}, -^{\mathcal{B}}), (-^{\mathcal{B}}, \sim), (\ll, \gg), (\text{to-pbv}(k, 1), \text{to-pbv}(k, 0)), (\text{to-pbv}(k, k), \text{to-pbv}(k, 0))\}$ and their permutations (g, f), and filtered out duplicates.

5.2 Experimental Setup

For our experimental evaluation, we consider three configurations: configuration BASELINE, which corresponds to the best configuration of [30]; configuration PBV, which implements our techniques as described in Section 4; and configuration EAGER, which corresponds to extending configuration BASELINE with our rewriter $\mathcal{RW}_{\mathcal{B}}$ and optimizations as detailed in Table 5, while still handling operators pow_2 and $\&^{\mathbb{N}}$ eagerly. Recall that the implementation of [30] utilized translations that were tailored to each considered benchmark set, which does not allow a faithful comparison to our PBV-configurations. Thus, we reimplemented the best configuration *comb* from [30] as configuration BASELINE in our tool. This not only allows us to evaluate BASELINE on a larger benchmark set, but also ensures a fair comparison since all configurations use the same underlying cvc5 version as well as the same infrastructure for parsing, manipulating and solving formulas.

Table 5 outlines the differences between the configurations. The features considered for each configuration are listed in the first column. *Target Theory* specifies the underlying theory (e.g., whether pow_2 and $\&^{\mathbb{N}}$ are uninterpreted or not). Accordingly, *Lazy* pow_2 and *Lazy* $\&^{\mathbb{N}}$ determine whether these operators are handled lazily or eagerly. For uninterpreted functions, we use quantified axiomatizations, and *No redundant axioms* determines whether these axiomatizations are always added to the input formula, regardless of whether the axiomatized operator occurs in the formula. *Multiple Bit-widths* corresponds to supporting multiple bit-widths in the same formula. $|$*-elimination* and $\oplus$*-elimination* denote the elimination of $|$ and $\oplus$ in the translation (as described in Section 4.2). The usage of an optimization to the translation of $\gg$, which eliminates a mod operation (as shown in Algorithm 3), is determined by $\gg$ *without* mod. *New lemmas for* $\&^{\mathbb{N}}$ and *New lemmas for* pow_2 correspond

to the usage of the new lemmas from Tables 3 and 4. $\mathcal{RW}_{\mathcal{B}}$ determines whether rewriter $\mathcal{RW}_{\mathcal{B}}$ from Section 4.1 is enabled, and mod-*reduction* determines whether some optimizations that eliminate nested mod in arithmetic operators are applied, as mentioned in Section 4.2.

Configurations Baseline and Eager rely on an *eager* translation. They introduce uninterpreted functions that are axiomatized by means of quantified formulas that are added as preamble to each translation. In contrast, configuration Pbv employs a translation to theory $T_{IA}(\text{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$, which does not rely on quantifiers and handles operators pow_2 and $\&^{\mathbb{N}}$ *lazily*. Further, configuration Baseline does not support multiple bit-widths in the same formula. Configuration Eager, on the other hand, incorporates several optimizations over Baseline, including the new lemmas, that are also incorporated in Pbv. Notice that operators $|$ and $\oplus$ are handled natively in configuration Baseline, while configurations Eager and Pbv use elimination rules for them. In Section B, we include an extended set of results with 14 more configurations.

▶ Remark 12. Even though our evaluation includes benchmark sets from the evaluation in [30], the numbers are not directly comparable due to the following reasons. Each benchmark in set *ic* describes an equivalence, which was split into two implications in [30]. Additionally, *conditional inverses* were introduced there, and were incorporated into the benchmarks with the main goal of proving invertibility conditions as correct. Benchmark set *rewrite* was solved in a semi-automated and iterative manner: whenever a rewrite candidate was proven correct, it was included as an axiom in the benchmarks to prove. In this evaluation, we refrain from such interventions, as our goal is to measure how efficient our approach is in a fully automated setting. Further, the translation of [30] utilized quantifier-specific optimizations and included instantiation patterns as well as concrete instantiations for some of the axioms. However, such patterns are often tailored to a specific solving procedure, and concrete instantiations are only sound for unsatisfiable benchmarks (which was the focus of [30]). In configuration Baseline, we do not use any of these optimizations.

We ran our experiments on a cluster of 22 machines equipped with Intel Xeon Gold 6348 CPUs. For each solver and benchmark pair, we used a CPU time limit of 60 seconds and a memory limit of 8GB. Preliminary experiments showed that the difference in results with higher time limits were not significant. Also, no configuration exceeded the memory limit of 8GB during solving.

5.3 Results

Table 6 summarizes the results for each benchmark set described in Section 5.1, over the configurations introduced in Section 5.2, as well as the virtual best solver over these configurations in column VBS.

Overall, configuration Pbv significantly outperforms Baseline and Eager in number of solved instances and runtime. Comparing Pbv against Eager indicates that using a lazily handling pow_2 and $\&^{\mathbb{N}}$ is superior to the quantifier-based eager approach. Comparing Eager against Baseline shows that the optimizations discussed in Section 4 are beneficial for the eager translation as well. On the 1863 instances commonly solved by all three configurations, Eager is almost 4 times faster than Baseline while Pbv is more than 12 times faster than Eager. On the 3060 instances commonly solved by Eager and Pbv, configuration Pbv is more than 18 times faster. Notice that configurations Baseline and Eager do not solve any satisfiable benchmarks, due to the usage of quantified axiomatizations. In fact, the presence of these axiomatizations already prevents both Baseline and Eager from reporting that the formula $\top$ is satisfiable.

Table 6 Overall number of solved benchmarks for each configuration. VBS corresponds to the virtual best solver over all three configurations. Time is the penalized runtime, counting unsolved instances as timeout.

Benchmarks	#	Baseline	Eager	Pbv	VBS
alive	200	71	93	**107**	124
ic	180	43	58	**77**	81
rewrite	2006	658	1221	**1331**	1382
syrew	1500	558	720	**912**	951
lemmas	70	12	14	**23**	23
icfb	46	1	9	**12**	12
mut	9441	669	1084	**4863**	4915
total	13443	2012	3199	**7325**	7488
sat		0	0	**3641**	3641
unsat		2012	3199	**3684**	3847
unique		24	57	**4222**	4303
time [seconds]		709k	634k	**375k**	355k

Our benchmark sets include in total 371 benchmarks that involve multiple bit-widths, which Baseline does not support. Configuration Pbv solves 178 instances, 20 of which are satisfiable. Configuration Eager solves 110 instances, all of which are unsatisfiable.

Simplifying the T_{PBV}-formula via rewriting as described in Section 4.1 is overall beneficial for both the Eager and the Pbv configurations but has a bigger impact on Eager. Without rewriting, Eager solves 474 fewer instances, while Pbv only loses 72 instances.

Proving the correctness of the invertibility conditions from [28] for arbitrary bit-widths (benchmark set *ic*) has previously been attempted with two different approaches: *(i)* the eager T_{PBV} approach described in [30] and *(ii)* a bit-width independent formalization in Coq [12]. Out of the 81 invertibility conditions proven across our three configurations, 11 were not proven by *(i)* and *(ii)*. Of these 11, 8 involve multiple bit-widths which neither *(i)* nor *(ii)* can handle. The remaining 3 conditions involve operators that are not supported by *(ii)* (signed inequalities, multiplication and division).

For benchmarks in sets *lemmas* and *icfb*, we provide the first bit-width independent verification. The lemmas and conditions in these sets were only proven correct for a range of fixed bit-widths [24, 31]. Configuration Pbv was able to confirm that the incorrect condition is indeed incorrect. However, it was unable to prove the corrected condition within the given time limit.

6 Conclusion and Further Research

We have presented a new theory of parametric bit-vectors T_{PBV} and proposed a dedicated, lazy procedure for solving T_{PBV}-formulas. We have shown experimentally that our techniques are a significant improvement over previous techniques in a wide range of benchmarks.

In future work, we plan to explore proof production for parametric bit-vectors, based on the algorithms presented in this paper. This will enable the integration of the proposed solver into proof assistants like Coq [11], Lean [22] or Isabelle/HOL [32], in order to increase automation in these tools for proofs involving parametric bit-vectors.

For our evaluation, we compiled sets of benchmarks for various challenging applications. These benchmarks are, however, relatively small in terms of formula size. It would be interesting to evaluate the scalability of our approach with respect to formula size. However, obtaining appropriate benchmarks for this purpose is currently challenging.[2] Generating and experimenting with such benchmarks is left for future work.

References

1 Haniel Barbosa, Clark W. 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 *TACAS (1)*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.

2 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2023.

3 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.7. Technical report, Department of Computer Science, The University of Iowa, 2025. Available at www.SMT-LIB.org.

4 Clark Barrett, A. Stump, and Cesare Tinelli. The SMT-LIB standard - version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories, Edinburgh, Scotland,(SMT '10)*, 2010.

5 Nikolaj S. Bjørner and Mark C. Pichora. Deciding fixed and non-fixed size bit-vectors. In *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 376–392. Springer, 1998. doi:10.1007/BFB0054184.

6 Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Ziyad Hanna, Zurab Khasidashvili, Amit Palti, and Roberto Sebastiani. Encoding RTL constructs for MathSAT: a preliminary report. *Electronic Notes in Theoretical Computer Science*, 144(2):3–14, 2006. doi:10.1016/J.ENTCS.2005.12.001.

7 Martin Brain, Florian Schanda, and Youcheng Sun. Building better bit-blasting for floating-point problems. In Tomás Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, volume 11427 of *Lecture Notes in Computer Science*, pages 79–98. Springer, 2019. doi:10.1007/978-3-030-17462-0_5.

8 Raik Brinkmann and Rolf Drechsler. RTL-datapath verification using integer linear programming. In *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design*, pages 741–746. IEEE, 2002. doi:10.1109/ASPDAC.2002.995022.

9 Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In *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 I 23*, pages 58–75. Springer, 2017. doi:10.1007/978-3-662-54577-5_4.

10 Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Satisfiability modulo transcendental functions via incremental linearization. In *Automated Deduction–CADE 26: 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6–11, 2017, Proceedings*, pages 95–113. Springer, 2017. doi:10.1007/978-3-319-63046-5_7.

[2] For example, *parameterizing* bit-vector benchmarks from SMT-LIB (which are exclusively over fixed-size bit-vectors) while preserving their semantics is not straightforward.

11 The Coq development team. The coq proof assistant reference manual version 8.9, 2019. URL: https://coq.inria.fr/distrib/current/refman/.

12 Burak Ekici, Arjun Viswanathan, Yoni Zohar, Cesare Tinelli, and Clark W. Barrett. Formal verification of bit-vector invertibility conditions in Coq. In Uli Sattler and Martin Suda, editors, *Frontiers of Combining Systems - 14th International Symposium, FroCoS 2023, Prague, Czech Republic, September 20-22, 2023, Proceedings*, volume 14279 of *Lecture Notes in Computer Science*, pages 41–59. Springer, 2023. doi:10.1007/978-3-031-43369-6_3.

13 Herbert Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

14 Florian Frohn and Jürgen Giesl. Satisfiability modulo exponential integer arithmetic. In *IJCAR (1)*, volume 14739 of *Lecture Notes in Computer Science*, pages 344–365. Springer, 2024. doi:10.1007/978-3-031-63498-7_21.

15 Aarti Gupta and Allan L. Fisher. Parametric circuit representation using inductive boolean functions. In *CAV*, volume 697 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 1993. doi:10.1007/3-540-56922-7_3.

16 Aarti Gupta and Allan L. Fisher. Representation and symbolic manipulation of linearly inductive boolean functions. In *ICCAD*, pages 192–199. IEEE Computer Society / ACM, 1993. doi:10.1109/ICCAD.1993.580055.

17 Fuqi Jia, Rui Han, Pei Huang, Minghao Liu, Feifei Ma, and Jian Zhang. Improving bit-blasting for nonlinear integer constraints. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, pages 14–25, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3597926.3598034.

18 Henry S. Warren Jr. *Hacker's Delight, Second Edition*. Pearson Education, 2013.

19 Hanna Lachnitt, Mathias Fleury, Leni Aniva, Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. IsaRare: Automatic verification of SMT rewrites in Isabelle/HOL. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, volume 14570 of *Lecture Notes in Computer Science*, pages 311–330. Springer, 2024. doi:10.1007/978-3-031-57246-3_17.

20 Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *PLDI*, pages 22–32. ACM, 2015. doi:10.1145/2737924.2737965.

21 Makai Mann, Amalee Wilson, Yoni Zohar, Lindsey Stuntz, Ahmed Irfan, Kristopher Brown, Caleb Donovick, Allison Guman, Cesare Tinelli, and Clark W. Barrett. Smt-Switch: A solver-agnostic C++ API for SMT solving. In *SAT*, volume 12831 of *Lecture Notes in Computer Science*, pages 377–386. Springer, 2021. doi:10.1007/978-3-030-80223-3_26.

22 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-79876-5_37.

23 Alexander Nadel. Bit-vector rewriting with automatic rule generation. 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 663–679. Springer, 2014. doi:10.1007/978-3-319-08867-9_44.

24 Aina Niemetz and Mathias Preiner. Ternary propagation-based local search for more bit-precise reasoning. In *FMCAD*, pages 214–224. IEEE, 2020. doi:10.34727/2020/ISBN.978-3-85448-042-6_29.

25 Aina Niemetz and Mathias Preiner. Bitwuzla. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2023. doi:10.1007/978-3-031-37703-7_1.

26 Aina Niemetz and Mathias Preiner. Correction: Ternary Propagation-Based Local Search for more Bit-Precise Reasoning. `https://bitwuzla.github.io/data/fmcad2020/fmcad2020-correction.pdf`, 2025.

27 Aina Niemetz, Mathias Preiner, and Armin Biere. Propagation based local search for bit-precise reasoning. *Formal Methods Syst. Des.*, 51(3):608–636, 2017. `doi:10.1007/S10703-017-0295-6`.

28 Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. On solving quantified bit-vector constraints using invertibility conditions. *Formal Methods Syst. Des.*, 57(1):87–115, 2021. `doi:10.1007/S10703-020-00359-9`.

29 Aina Niemetz, Mathias Preiner, Andrew Reynolds, Yoni Zohar, Clark W. Barrett, and Cesare Tinelli. Towards bit-width-independent proofs in SMT solvers. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 366–384. Springer, 2019. `doi:10.1007/978-3-030-29436-6_22`.

30 Aina Niemetz, Mathias Preiner, Andrew Reynolds, Yoni Zohar, Clark W. Barrett, and Cesare Tinelli. Towards satisfiability modulo parametric bit-vectors. *J. Autom. Reason.*, 65(7):1001–1025, 2021. `doi:10.1007/S10817-021-09598-9`.

31 Aina Niemetz, Mathias Preiner, and Yoni Zohar. Scalable bit-blasting with abstractions. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 178–200. Springer, 2024. `doi:10.1007/978-3-031-65627-9_9`.

32 Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Science & Business Media, 2002. `doi:10.1007/3-540-45949-9`.

33 Andres Nötzli, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W Barrett, and Cesare Tinelli. Reconstructing fine-grained proofs of rewrites using a domain-specific language. In *FMCAD*, pages 65–74, 2022. `doi:10.34727/2022/ISBN.978-3-85448-053-2_12`.

34 Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark W. Barrett, and Cesare Tinelli. Syntax-guided rewrite rule enumeration for SMT solvers. In *SAT*, volume 11628 of *Lecture Notes in Computer Science*, pages 279–297. Springer, 2019. `doi:10.1007/978-3-030-24258-9_20`.

35 Mark Christopher Pichora. *Automated Reasoning about Hardware Data Types Using Bit-Vectors of Symbolic Lengths*. PhD thesis, University of Toronto, CAN, 2003. AAINQ84686.

36 Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *CAV (2)*, volume 11562 of *Lecture Notes in Computer Science*, pages 74–83. Springer, 2019. `doi:10.1007/978-3-030-25543-5_5`.

37 Cesare Tinelli and Calogero G. Zarba. Combining decision procedures for sorted theories. In Jóse Júlio Alferes and João Leite, editors, *Logics in Artificial Intelligence*, pages 641–653, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. `doi:10.1007/978-3-540-30227-8_53`.

38 Zhihong Zeng, Priyank Kalla, and Maciej Ciesielski. LPSAT: a unified approach to RTL satisfiability. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 398–402. IEEE, 2001. `doi:10.1109/DATE.2001.915055`.

39 Yoni Zohar, Ahmed Irfan, Makai Mann, Aina Niemetz, Andres Nötzli, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. Bit-precise reasoning via int-blasting. In *VMCAI*, volume 13182 of *Lecture Notes in Computer Science*, pages 496–518. Springer, 2022. `doi:10.1007/978-3-030-94583-1_24`.

A Rewrite Rules of the Parametric Bit-Vector Rewriter $\mathcal{RW}_{\mathcal{B}}$

The following table lists the rewrite rules implemented in our rewriter $\mathcal{RW}_{\mathcal{B}}$. We denote with k_x the bitwidth of x, use a lexicographic ordering $<_{lex}$ between terms, and normalize commutative operations ($+^{\mathcal{B}}$, &, etc.) according to this ordering. For example, if $t_1 <_{lex} t_2$, then $t_1 +^{\mathcal{B}} t_2$ is rewritten to itself and $t_2 +^{\mathcal{B}} t_1$ is rewritten to $t_1 +^{\mathcal{B}} t_2$.

Rule Name	Term	Rewritten Term
bv-concat-extract-merge	(concat (extract k (+ j 1) s) (extract j i s))	(extract k i s)
bv-extract-extract	(extract l k (extract j i x))	(extract (+ i l) (+ i k) x)
bv-extract-whole	(extract k_x 0 x)	x
bv-add-zero	(bvadd x 0)	x
bv-reverse-extract-and	(bvand (extract j i x) (extract j i y))	(extract j i (bvand x y))
bv-xor-simplify-2	(bvxor x (bvnot x))	(bvnot 0)
bv-or-zero	(bvor x (int_to_pbv k_x 0))	x
bv-mul-one	(bvmul x (int_to_pbv k_x 1))	x
bv-mul-zero	(bvmul x (int_to_pbv k_x 0))	(int_to_pbv k_x 0)
bv-zero-extend-eliminate	(zero-extend 0 x)	x
bv-sign-extend-eliminate	(sign-extend 0 x)	x
bv-not-neq	(= x (bvnot x))	false
bv-neg-sub	(bvneg (bvsub x y))	(bvsub y x)
bv-neg-idemp	(bvneg (bvneg x))	x
bv-ugt-eliminate	(bvugt x y)	(bvult y x)
bv-uge-eliminate	(bvuge x y)	(bvule y x)
bv-sgt-eliminate	(bvsgt x y)	(bvslt y x)
bv-sge-eliminate	(bvsge x y)	(bvsle y x)
bv-shl-by-const-0	(bvshl x (int_to_pbv k_x 0))	x
bv-shl-by-const-2	(bvshl x (int_to_pbv k_x k_x))	(int_to_pbv k_x 0)
bv-lshr-by-const-0	(bvlshr x (int_to_pbv k_x 0))	x
bv-lshr-by-const-2	(bvlshr x (int_to_pbv k_x k_x))	(int_to_pbv k_x 0)
bv-ashr-by-const-0	(bvashr x (int_to_pbv k_x 0))	x
bv-bitwise-idemp-2	(bvor x x)	x
bv-or-one	(bvor x (bvnot x))	(bvnot (int_to_pbv k_x 0))
bv-xor-duplicate	(bvxor x x)	(int_to_pbv k_x 0)
bv-xor-zero	(bvxor x (int_to_pbv k_x 0))	x
bv-bitwise-not-or	(bvor x (bvnot x))	(bvnot (int_to_pbv k_x 0))
bv-xor-not	(bvxor (bvnot x) (bvnot y))	(bvxor x y)
bv-not-idemp	(bvnot (bvnot x))	x
bv-ult-zero-1	(bvult (int_to_pbv k_x 0) x)	(not (= x (int_to_pbv k_x 0)))
bv-ult-zero-2	(bvult x (int_to_pbv k_x 0))	false
bv-ult-self	(bvult x x)	false
bv-lt-self	(bvslt x x)	false
bv-ule-self	(bvule x x)	true
bv-ule-zero	(bvule x (int_to_pbv k_x 0))	(= x (int_to_pbv k_x 0))
bv-zero-ule	(bvule (int_to_pbv k_x 0) x)	true
bv-sle-self	(bvsle x x)	true
bv-ule-max	(bvule x (bvnot x))	true
bv-udiv-zero	(bvudiv x (int_to_pbv k_x 0))	(bvnot (int_to_pbv k_x 0))
bv-udiv-one	(bvudiv x (int_to_pbv k_x 1))	x
bv-urem-one	(bvurem x (int_to_pbv k_x 1))	(int_to_pbv k_x 0)
bv-urem-self	(bvurem x x)	(int_to_pbv k_x 0)
bv-shl-zero	(bvshl (int_to_pbv k_x 0) x)	(int_to_pbv k_x 0)
bv-lshr-zero	(bvlshr (int_to_pbv k_x 0) x)	(int_to_pbv k_x 0)
bv-ashr-zero	(bvashr (int_to_pbv k_x 0) x)	(int_to_pbv k_x 0)
bv-ult-one	(bvult x (int_to_pbv k_x 1))	(= x (int_to_pbv k_x 0))

B More Detailed Evaluation

In addition to the experimental evaluation presented above, we have performed an in-depth ablation study of the various features listed in Table 5. In this section, we describe the results for 14 configurations we evaluated in addition to the three described in Table 5. These additional configurations are defined as described in Tables 7 and 8.

Table 7 Intermediate configurations with $T_1 = T_{IA}(\text{pow}_2, \&^{\mathbb{N}}, |^{\mathbb{N}}, \oplus^{\mathbb{N}})$.

Feature	OR-E	XOR-E	SH-M-E	ALL-E	POW2++	PIAND++
Target Theory	T_1					
Multiple Bit-widths	✗	✗	✗	✗	✗	✗
Lazy pow_2	✗	✗	✗	✗	✗	✗
Lazy $\&^{\mathbb{N}}$	✗	✗	✗	✗	✗	✗
$\|$*-elimination*	✓	✗	✗	✓	✗	✗
$\oplus$*-elimination*	✗	✓	✗	✓	✗	✗
$\gg$ *without* mod	✗	✗	✓	✓	✗	✗
New lemmas for $\&^{\mathbb{N}}$	✗	✗	✗	✗	✗	✓
New lemmas for pow_2	✗	✗	✗	✗	✓	✗
No redundant axioms	✗	✗	✗	✗	✗	✗
$\mathcal{RW}_{\mathcal{B}}$	✗	✗	✗	✗	✗	✗
mod*-reduction*	✗	✗	✗	✗	✗	✗

Table 8 Intermediate configurations. $T_3 = T_{IA}(\text{pow}_{2_\star}, \&^{\mathbb{N}})$, that is, pow_2 has a fixed interpretation, while $\&^{\mathbb{N}}$ is freely interpreted. $T_4 = T_{IA}(\text{pow}_2, \&^{\mathbb{N}}_\star)$, that is, $\&^{\mathbb{N}}$ has a fixed interpretation, while pow_2 is freely interpreted. $T_5 = T_{IA}(\text{pow}_2, \&^{\mathbb{N}})$. $T_2 = T_{IA}(\text{pow}_{2_\star}, \&^{\mathbb{N}}_\star)$.

Feature	POW2-L	PIAND-L	EAGER$^{\mathcal{R}-}$	EAGER$^{\mathcal{R}}$	EAGER^{-}	PBV$^{\mathcal{R}}$	PBV$^{\mathcal{R}-}$	PBV^{-}
Target Theory	T_3	T_4	T_5	T_5	T_5	T_2	T_2	T_2
Multiple Bit-widths	✗	✗	✓	✓	✓	✓	✓	✓
Lazy pow_2	✓	✗	✗	✗	✗	✓	✓	✓
Lazy $\&^{\mathbb{N}}$	✗	✓	✗	✗	✗	✓	✓	✓
$\|$*-elimination*	✗	✗	✓	✓	✓	✓	✓	✓
$\oplus$*-elimination*	✗	✗	✓	✓	✓	✓	✓	✓
$\gg$ *without* mod	✗	✗	✓	✓	✓	✓	✓	✓
New lemmas for $\&^{\mathbb{N}}$	✗	✓	✓	✓	✓	✓	✓	✓
New lemmas for pow_2	✓	✗	✓	✓	✓	✓	✓	✓
No redundant axioms	✗	✗	✓	✓	✓	✓	✓	✓
$\mathcal{RW}_{\mathcal{B}}$	✗	✗	✗	✗	✓	✗	✗	✓
mod*-reduction*	✗	✗	✗	✓	✗	✓	✗	✗

Configuration OR-E is obtained from BASELINE by turning on the elimination of $|$, and similarly for XOR-E. The right-most 6 configurations of Table 8 differ by turning on or off the bit-vector rewriter $\mathcal{RW}_{\mathcal{B}}$, as well as the arithmetic optimizations that eliminate mod operators. In particular, EAGER$^{\mathcal{R}-}$ is obtained from EAGER by turning both of these off.

The results of the additional configurations are shown in Tables 9 and 10. Table 9 focuses on configurations that are based on BASELINE. Note that the elimination of mod in $\gg$ (configuration SH-M-E) has a greater effect than the elimination of $|$ and $\oplus$, since our benchmarks include more occurrences of $\gg$. Table 10 includes configurations that evaluate the effect of the bit-vector rewriter $\mathcal{RW}_{\mathcal{B}}$. Overall, enabling the rewriter improves performance, especially for the eager configurations.

Table 9 Overall number of solved benchmarks for each configuration. VBS corresponds to the virtual best solver over all three configurations.

Benchmarks	#	OR-E	XOR-E	SH-M-E	ALL-E	POW2++	PIAND++	VBS
alive	200	71	71	71	71	71	71	72
ic	180	44	44	48	49	43	43	50
rewrite	2006	692	653	724	758	669	633	809
syrew	1500	561	556	604	603	570	551	625
lemmas	70	12	12	13	13	12	12	13
icfb	46	1	1	1	1	1	1	1
mut	9441	692	663	774	794	669	634	843
total	13443	2071	2000	2235	2289	2033	1945	2413
sat		0	0	0	0	0	0	0
unsat		2071	2000	2235	2289	2033	1955	2413

Table 10 Overall number of solved benchmarks for each configuration. VBS corresponds to the virtual best solver over all three configurations.

Benchmarks	#	POW2-L	PIAND-L	EAGER$^{\not\leftarrow}$	EAGER$^{\not\leftarrow}$	EAGER^{-}	PBV$^{\not\leftarrow}$	PBV$^{\not\leftarrow}$	PBV^{-}	VBS
alive	200	63	74	87	95	85	105	97	100	125
ic	180	55	45	58	59	59	75	75	77	83
rewrite	2006	681	797	732	925	1068	1296	1148	1189	1381
syrew	1500	580	593	608	721	606	911	796	798	956
lemmas	70	17	13	14	14	14	23	23	23	25
icfb	46	1	1	9	9	9	11	11	12	12
mut	9441	732	733	885	902	1088	4832	4840	4881	5138
total	13443	2129	2256	2393	2725	2929	7253	6990	7080	7720
sat		0	0	0	0	0	3652	3657	3651	3841
unsat		2129	2256	2393	2725	2929	3601	3333	3429	3879

Semi-Algebraic Proof Systems for QBF

Olaf Beyersdorff
Friedrich Schiller University Jena, Germany

Ilario Bonacina
UPC Universitat Politècnica de Catalunya, Barcelona, Spain

Kaspar Kasche
Friedrich Schiller University Jena, Germany

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

Luc Nicolas Spachmann
Friedrich Schiller University Jena, Germany

Abstract

We introduce new semi-algebraic proof systems for Quantified Boolean Formulas (QBF) analogous to the propositional systems Nullstellensatz, Sherali-Adams and Sum-of-Squares. We transfer to this setting techniques both from the QBF literature (strategy extraction) and from propositional proof complexity (size-degree relations and pseudo-expectation). We obtain a number of strong QBF lower bounds and separations between these systems, even when disregarding propositional hardness.

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

Keywords and phrases QBF, Proof Complexity, Sums-of-Squares, Nullstellensatz, Sherali-Adams, Semi-Algebraic Proof Systems

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.5

Funding *Olaf Beyersdorff*: Carl-Zeiss Foundation and DFG grant BE 4209/3-1.
Ilario Bonacina: This author was funded by the AEI with the grant number PID2022-138506NB-C22.
Kaspar Kasche: Carl-Zeiss Foundation.
Meena Mahajan: Supported in part by the J. C. Bose Fellowship of SERB, ANRF.

Acknowledgements The authors would like to thank the Oberwolfach Research Institute for Mathematics: the idea for this work started during the Oberwolfach workshop 2413 "Proof Complexity and Beyond", and the Schloss Dagstuhl – Leibniz Center for Informatics: part of this work has been done during the Dagstuhl Seminar 24421 "SAT and Interactions".

1 Introduction

Two key results in algebraic and semi-algebraic geometry are the Nullstellensatz and the Positivstellensatz. The first can be seen as an algebraic identity certifying that a set of polynomial equations is unsatisfiable while the second as an algebraic identity certifying that a system of polynomial *inequalities* is unsatisfiable. In other words, both the Nullstellensatz and the Positivstellensatz naturally give rise to proof systems and in recent years intense research was performed on the proof complexity of such systems. In particular, the proof system *Sum-of-Squares*, a special case of Positivstellensatz, starting from [7], has received a lot of attention for its connection with algorithms based on hierarchies of SDP relaxations (see for instance [45, 47]). For similar reasons, the even more restrictive *Sherali-Adams* proof system has been investigated, named after its connection with the Sherali-Adams hierarchy of linear programming since its original definition (see for instance [36, 50]).


© Olaf Beyersdorff, Ilario Bonacina, Kaspar Kasche, Meena Mahajan, and Luc Nicolas Spachmann;
licensed under Creative Commons License CC-BY 4.0



In this work, we devise a simple and natural way to extend the proof systems Nullstellensatz (NS), Sum-of-Squares (SOS) and Sherali-Adams (SA) from the context of propositional existentially quantified variables to existentially and universally quantified variables. In other words, we show how to define proof systems for quantified Boolean formulas (QBF) inspired by the propositional proof systems above.

The study of propositional and QBF proof systems is motivated both by theoretical reasons and also by connections to SAT and QBF solving [10, 22, 28]. While for SAT-solvers conflict-driven clause learning (CDCL) is the ruling paradigm, which by the seminal work of [1, 48] is essentially equivalent to the propositional proof system *Resolution* (Res), there are several competing approaches in QBF, with CDCL-based [53] and *expansion*-based solving [43] among the main paradigms. To model the strength of QBF-solvers several (often incomparable) QBF proof systems have been introduced and analysed [14, 39, 43]. Of most relevance to this work is QU-Resolution (QU-Res) [44, 52]. QU-Res adds to propositional Resolution the ∀-reduction rule that allows to eliminate universal variables from clauses. In [16] it has been shown that this approach of augmenting a propositional proof system by a ∀-reduction rule taking care of universal quantifiers also works for other common inference-based proof systems such as various Frege systems [16, 32], *Cutting Planes* [19, 33] – a proof system modelling geometric reasoning related to Chvátal-Gomory cuts – and *Polynomial Calculus* [21, 31], modelling algebraic reasoning related to Gröbner bases computations.

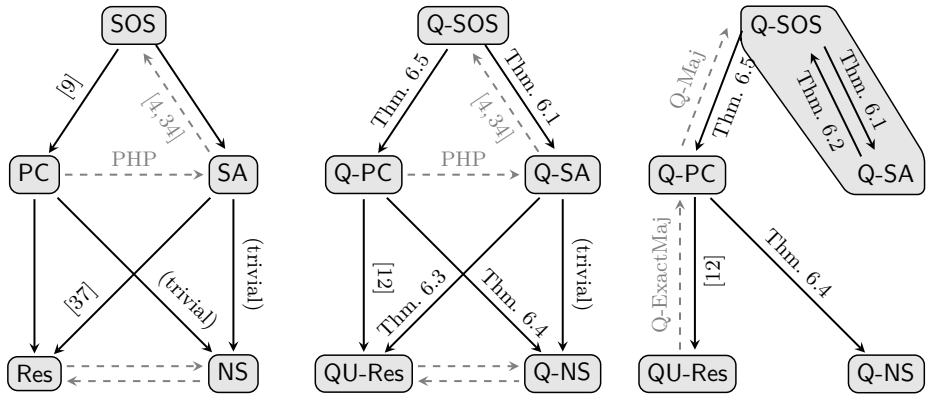
A common feature of all the propositional proof systems above is that they are *inference*-based, unlike the static systems NS/SA/SOS where a proof is just an algebraic identity of some specific form depending on the system at hand. This has posed quite some problems to adapt such systems to the QBF setting, and it was not clear at all whether an approach similar to a ∀-reduction rule was even viable. Recently there has been a suggestion to define QBF analogues of NS based on ∀-expansion [29], but these differ considerably from our approach here and the ∀-reduction paradigm discussed above.

We show that an approach similar to a ∀-reduction rule *does* allow to **define QBF versions of** NS/SA/SOS, which we call Q-NS/Q-SA/Q-SOS respectively. We argue that our definitions are quite natural: they add to the algebraic equations of NS/SA/SOS simple polynomials that strongly resemble ∀-reduction and meet the same technical condition on variable dependence.

We begin the systematic study of these QBF proof systems in terms of **lower and upper bounds, strategy extraction, and simulations**. Concerning the latter, Figure 1a recalls the relations between propositional NS/SA/SOS and further propositional proof systems such as Resolution (Res) and Polynomial Calculus (PC). In Figure 1b, we depict our results on the new QBF systems Q-NS/Q-SA/Q-SOS and how they relate to QU-Res and Q-PC. The figures are virtually identical: what changes is that proofs of the simulations, although mimicking those in the propositional setting, require extra care. Figure 1c depicts the simulation order when we factor out the propositional complexity and consider *genuine* QBF hardness stemming from quantifier alternations – a framework that has become standard in QBF proof complexity (cf. [20, 30] for background). We call the "genuine" size measure qsize, which only counts monomial size in the new ∀-reduction polynomials. Lower bounds on qsize are tighter and trivially imply lower bounds on the traditional size measure that counts all monomials. Hence, lower bounds and separations in qsize are harder to obtain. In fact, Q-SA and Q-SOS become equivalent w.r.t. qsize while they are separated w.r.t. size.

The fact that the systems we define fit so nicely into the lattice of QBF proof systems using the ∀-reduction approach suggests that the definitions we give are natural analogues of ∀-reduction in this context. The analogy with the ∀-reduction rule of QU-Res gets even

clearer when using the language of weighted clauses and Resolution to describe NS and SA [25]. For simplicity (and length constraints) we describe Q-NS and Q-SA using the usual algebraic language instead of the language of weighted clauses.



(a) Propositional systems.

(b) QBF systems p-simulations w.r.t. size.

(c) QBF systems p-simulations w.r.t. qsize.

Figure 1 Simulations and separations between algebraic proof systems in the propositional and the QBF setting. By $P \to Q$ we indicate that proof system P polynomially simulates Q, while $P \dashrightarrow Q$ means that the proof system P does not polynomially simulate Q. We omit polynomial (non-)simulations implied by those displayed.

For lower and upper bounds we develop and adapt techniques that originate both from propositional and QBF proof complexity.

Regarding the **transfer of propositional techniques**, we show how to lift common techniques for SOS from the propositional setting to Q-SOS: we establish a **size-degree relation** analogous to the propositional one [2], and show how to adapt the notion of **pseudo-expectation**, the prime lower-bound method for semi-algebraic systems [40] to the QBF setting. Both adaptations require interesting modifications and do not just replicate the propositional techniques (see Lemma 3.11 and Definition 5.1 with the discussion thereafter). We use pseudo-expectations to show an exponential lower bound for Q-SOS for the Equality QBFs [11] with respect to the tighter qsize measure (Theorem 5.3).

Regarding **QBF techniques**, we develop **strategy extraction** for Q-SOS. Strategy extraction has become the predominant technique to analyse QBF proof systems (see [12, 16, 17] for instance) and is also of tremendous practical importance for QBF solving and verification [5, 22, 39]. Specifically, we show that Q-SOS allows strategy extraction by polynomial threshold functions and develop a new score game interpretation. Interestingly, this score game allows to characterise genuine proof size in Q-SOS and Q-NS (Theorem 3.6). We also use it to elegantly show completeness of the new systems and a linear upper bound for the Q-Majority QBFs, which are known to be hard for Q-PC [21], yielding the separation depicted in Figure 1c.

Structure of the paper. Section 2 contains preliminaries and notation. Section 3 defines our new semi-algebraic QBF systems and shows their soundness and completeness together with the size-degree relation for Q-SOS. Section 4 contains the strategy extraction for Q-SOS and some consequences. In Section 5 we develop the lower bound technique of pseudo-expectations for Q-SOS and show an exponential lower bound. Section 6 we compare the QBF systems via p-simulations. We conclude in Section 7 with some open problems.

2 Preliminaries

QBF preliminaries. We consider Quantified Boolean Formulas (QBF) of the form $\mathcal{Q}.\varphi$, where φ is a CNF formula and $\mathcal{Q}$ is the quantifier prefix. Both the variables of φ and the variables of $\mathcal{Q}$ range over a set of Boolean variables V. Let $\text{vars}_\forall(\mathcal{Q})$ (resp. $\text{vars}_\exists(\mathcal{Q})$) be the set of universally (resp. existentially) quantified variables in $\mathcal{Q}$.

The evaluation of a QBF formula $\mathcal{Q}.\varphi$ can be seen as a game (the *evaluation* game) between two players: the existential $\exists$-player and the universal $\forall$-player, where the $\exists$-player's goal is to satisfy the formula φ and the $\forall$-player's goal is to falsify it. The players take turns according to the order of the quantifiers in $\mathcal{Q}$. We call this game the evaluation game to distinguish it from a new game, the *score* game, which we introduce in Section 3.1.

A very well-studied QBF proof system is QU-Res [6, 44, 52], which can be seen as a natural extension of the propositional proof system Resolution [24, 49] to the QBF setting.

The QBF proof system QU-Res refutes a false QBF $\mathcal{Q}.\varphi$ inferring the empty clause $\bot$ from the clauses in φ using the resolution rule $\frac{C\vee v \qquad D\vee\neg v}{C\vee D}$, but also using a $\forall$-*reduction* rule $\frac{C\vee u}{C}$, where all the variables in C must be on the left of u in $\mathcal{Q}$. The *size* of a QU-Res refutation π ($\text{size}(\pi)$) is the number of applications of rules in π, while *Q-size* ($\text{qsize}(\pi)$) is the number of applications of the $\forall$-reduction rule.

Algebraic proof systems. Given a set of Boolean variables V, let $\overline{V}$ be the set of new formal variables $\overline{v}$ for $v \in V$. We consider polynomials with rational coefficients and variables in $V \cup \overline{V}$, i.e. polynomials in the ring $\mathbb{Q}[V \cup \overline{V}]$. Given a polynomial p and an assignment α of its variables, we denote with $p|_\alpha$ the evaluation of p in α.

In this work we encode clauses and CNF formulas into polynomials using the so-called *twin-variables encoding*. A clause $C = \bigvee_{v\in P} v \vee \bigvee_{v\in N} \neg v$ is encoded as the set of polynomials

$$\text{enc}(C) = \left\{ \prod_{v\in P} \overline{v} \prod_{v\in N} v \right\} \cup \{v^2 - v,\ v + \overline{v} - 1 : v \in P \cup N\} \ .$$

A CNF $\varphi = \bigwedge_{j=1}^m C_j$ is encoded as a set of polynomials $\text{enc}(\varphi) = \bigcup_{j=1}^m \text{enc}(C_j)$. The formula φ is satisfiable if and only if the set of polynomial equalities $\{p = 0 : p \in \text{enc}(\varphi)\}$ is satisfiable.

▶ Fact 2.1. Given a polynomial $r \in \mathbb{Q}[V \cup \overline{V}]$, if r evaluates to 0 over every Boolean assignment satisfying φ (and setting $v + \overline{v}$ to 1), then r is in the ideal generated by $\text{enc}(\varphi)$, i.e. there are polynomials q_p such that $r = \sum_{p\in\text{enc}(\varphi)} q_p p$.

A refutation of an unsatisfiable CNF φ in variables V in the system Nullstellensatz, NS, (resp. Sherali-Adams, SA, or Sum-of-Squares, SOS) is an algebraic identity π of the form

$$\sum_{p\in\text{enc}(\varphi)} q_p p + q + 1 = 0 \ , \tag{1}$$

where all the polynomials q_p, q are in $\mathbb{Q}[V \cup \overline{V}]$, and for NS q is identically 0 (resp. for SA q is a polynomial with non-negative coefficients, and for SOS $q = \sum_{s\in S} s^2$, that is q is a sum of squares). The *size* of the NS/SA/SOS refutation π, $\text{size}(\pi)$, is the number of monomials (counted with repetition) in q_p and q. The *degree* of π, $\deg(\pi)$, is the maximum degree of any of the polynomials $q_p p$, and q.

▶ Remark 2.2 (On variations of NS, SA and SOS). The proof system NS has been considered also for polynomials over arbitrary fields [27]. In this paper we focus only on polynomials with rational coefficients. The proof systems NS, SA and SOS have been also studied using

a different encoding of CNF formulas: the encoding $\mathrm{enc}'(C)$, which is the same as $\mathrm{enc}(C)$ but with each $\bar{v}$ variable substituted by $1 - v$. The systems NS, SA and SOS under the enc' encoding are exponentially weaker than the corresponding system under the encoding enc [38]. In this paper we focus only on polynomials using the encoding enc. The proof system SOS has also been studied recently on Boolean variables representing the Boolean values as ± 1 instead of $0/1$ [51], i.e. using instead of the polynomials $v^2 - v$ the polynomials $v^2 - 1$. In this paper we focus only on polynomials using $0/1$-valued variables.

Another well-studied algebraic propositional proof system is *Polynomial Calculus* (PC) [31]. A Polynomial Calculus (over $\mathbb{Q}$) refutation of the set of polynomials $\mathrm{enc}(\varphi)$, for an unsatisfiable CNF formula φ, is a sequence of polynomials showing that 1 can be derived from $\mathrm{enc}(\varphi)$ using the inference rules $\frac{p \quad q}{p+q}$ for polynomials p, q, and $\frac{p}{vp}$ where v is a variable or $v \in \mathbb{Q}$. The *degree* and (monomial) *size* of the refutation are respectively the largest degree of a polynomial in it and the number of monomials in it (counted with multiplicity).

In [16, 21], the authors showed how to extend the proof system PC to the QBF context. This resulted in QBF proof system *Q-Polynomial Calculus* (Q-PC). Q-PC refutes a QBF $\mathcal{Q}.\varphi$ analogously to the system PC, i.e. showing that the polynomial 1 can be derived from the polynomials in $\mathrm{enc}(\varphi)$ using the inference rules $\frac{p \quad q}{p+q}$ for polynomials p, q, and $\frac{p}{vp}$ where v is a variable or $v \in \mathbb{Q}$, but also using a $\forall$-*reduction* rule $\frac{p}{p|_{u=b}}$, for $b \in \{0, 1\}$ where all the variables in p distinct from u must be left of u in $\mathcal{Q}$. The size of a Q-PC refutation is the number of monomials in the refutation (counted with repetition), while the qsize is the number of monomials in the polynomials involved in the $\forall$-reduction steps (again counted with repetition). The reason to study qsize and not just size is to factor out the propositional hardness of the principles and focus on genuine QBF hardness (cf. [20, 30]).

3 Algebraic systems for QBFs

We introduce new QBF proof systems inspired by propositional NS/SA/SOS. We call them Q-NS/Q-SA/Q-SOS. As their propositional counterparts they are static proof systems: a refutation of a false QBF $\mathcal{Q}.\varphi$ over variables V in Q-Nullstellensatz, Q-NS, (resp. Q-Sherali-Adams, Q-SA, and Q-Sum-of-Squares, Q-SOS) is an algebraic identity π of the form

$$\sum_{p \in \mathrm{enc}(\varphi)} q_p p + \sum_{u \in \mathrm{vars}_\forall(\mathcal{Q})} q_u(1 - 2u) + q + 1 = 0 \ , \tag{2}$$

where all the polynomials q_p, q_u q are in $\mathbb{Q}[V \cup \overline{V}]$, the variables in q_u are all quantified before u in $\mathcal{Q}$ (i.e. on the left of u), and for Q-NS q is identically 0 (resp. for Q-SA q is a polynomial with non-negative coefficients, and for Q-SOS $q = \sum_{s \in S} s^2$, that is q is a sum of squares). We call the expression in eq. (2) a Q-NS-*refutation* of $\mathcal{Q}.\varphi$ (resp. Q-SA-refutation/Q-SOS-refutation).

▶ **Definition 3.1** (size, degree, qsize and $\mathrm{qdeg}_\exists$). *The* size *of a* Q-NS/Q-SA/Q-SOS *refutation* π *(*$\mathrm{size}(\pi)$*) is the number of monomials (counted with repetition) in* q_p, q_u *and* q. *The* degree *of* π *(*$\deg(\pi)$*) is the maximum degree of any of the polynomials* $q_p p$, $q_u(1-2u)$, *and* q.

The Q-size *of* π ($\mathrm{qsize}(\pi)$) *is defined analogously to the* size *but accounts only for the monomials in the polynomials* q_u. *The existential Q-degree of* π ($\mathrm{qdeg}_\exists(\pi)$) *is the maximum existential degree of any* q_u, *where the existential degree is the highest number of existentially quantified variables in any monomial.*

The definitions of size and degree for Q-NS/Q-SA/Q-SOS are completely analogous to the definitions in the propositional setting, while Q-size and Q-degree factor out propositional hardness and therefore give measures more appropriate to study principles where the hardness

stems from quantification. The definition of Q-size also aligns with genuine QBF hardness measures defined in [20] and analysed e.g. in [13,21] for QU-Resolution and Q-PC, where only universal reduction steps are counted. In a sense, the polynomial q_u in (2) can be understood as a universal reduction step on u. In particular, on QBFs without universal variables, Q-NS/Q-SA/Q-SOS are equivalent to their propositional counterparts NS/SA/SOS.

Any lower bound on Q-size immediately implies the same lower bound on size. The reason to consider the *existential* Q-degree is a connection between Q-size and existential Q-degree similar to the inequality between size and width in resolution [13] (see Section 3.3).

As a first result we prove that Q-NS, Q-SA, Q-SOS are sound QBF proof systems.

▶ **Theorem 3.2** (soundness). *If there exists a* Q-NS- *or* Q-SA- *or* Q-SOS*-refutation of $\mathcal{Q}.\varphi$, then $\mathcal{Q}.\varphi$ is false.*

Proof. Suppose, for a contradiction, that $\mathcal{Q}.\varphi$ is a true QBF, so the ∃-player has a winning strategy σ, but at the same time there is a refutation of $\mathcal{Q}.\varphi$ of the form as in eq. (2) where all the variables in polynomials q_u are on the left of u and q is identically zero (Q-NS), or a polynomial with non-negative coefficients (Q-SA), or a sum of squares (Q-SOS).

For every strategy τ of the ∀-player, the game proceeds following the strategies σ and τ and constructs a total Boolean assignment $\alpha_{\sigma,\tau}$ satisfying the matrix φ. That is $\sum_{p\in \mathrm{enc}(\varphi)} q_p p$ evaluates to 0 under every assignment $\alpha_{\sigma,\tau}$. For a universal variable u, we write $\tau_{<u}$ for the part of τ on variables to the left of u and $\tau_{\geqslant u}$ for the rest of τ. Taking a uniform probability distribution over all universal strategies τ, for every universal variable u it holds that:

$$\mathbb{E}_\tau\left[q_u(1-2u)\right] \overset{(\star)}{=} \mathbb{E}_{\tau_{<u}}\left[q_u\,\mathbb{E}_{\tau_{\geqslant u}}\left[1-2u\right]\right] = \mathbb{E}_{\tau_{<u}}\left[q_u\cdot 0\right] = 0\ ,$$

where in the equality $(\star)$ we used the fact that all the polynomials q_u only depend on variables on the left of u. Hence, evaluating both sides of eq. (2) on $\alpha_{\sigma,\tau}$ and taking $\mathbb{E}_\tau$, the LHS equals $\mathbb{E}_\tau[q|_{\alpha_{\sigma,\tau}} + 1]$, which is always at least 1, while the RHS is 0. Contradiction. ◀

▶ Remark 3.3 (Q-NS over arbitrary fields). The definition of Q-NS from eq. (2) can be trivially adapted from polynomials over $\mathbb{Q}$ to arbitrary fields of characteristic different from 2. In characteristic 2 it gives an unsound system since all the terms $(1-2u)$ are identically 1. Indeed, in characteristic 2 every formula with at least one universal variable could be "refuted" by setting all $q_p = 0$ and a single $q_u = 1$.

▶ Remark 3.4 (unary vs binary coefficients). Unlike in the propositional setting where the unary versions of NS/SA/SOS give rise to non-trivial (and interesting) proof systems [41], in the QBF setting imposing unary (i.e. ± 1) coefficients in Q-NS/Q-SA/Q-SOS refutations seems to give rise to very weak systems. For instance, unary Q-SOS cannot even efficiently refute a false QBF formula as simple as $\forall u_1 \forall u_2 \cdots \forall u_n . \bigvee_{i=1}^n u_i$. We omit the argument as it is similar to the one used to prove Theorem 3.2 above.

3.1 Completeness via a score game

To show completeness of Q-NS/Q-SA/Q-SOS, we introduce a new *score* game. We call it *score* game to distinguish it from the *evaluation* game used for the QBF semantics (cf. Sec. 2).

The *score* game, as the evaluation game, is played between a universal and an existential player on a QBF $\mathcal{Q}.\varphi$, building a total Boolean assignment. As in the evaluation game, the players take turns according to the quantifier prefix $\mathcal{Q}$ and the existential player can freely decide on the value of existential variables. For the universal variables the score game differs from the usual evaluation game: the universal player gives a preference for the universal

variable u in the form of a number $s_u \in \mathbb{Q}$. Then, the existential player sets u to $b \in \{0,1\}$ and the universal player scores $s_u(2b-1)$ points. There are two variants of this game that differ in the winning condition:

variant 1 the universal player wins if φ is falsified or the total score is strictly positive;

variant 2 the universal player wins if φ is falsified or the total score equals 1.

Clearly every winning strategy of VARIANT 2 is also a winning strategy of VARIANT 1. The intuition behind the universal preferences is that the sign of s_u encodes the preferred assignment (if the preferred assignment is $u = 0$ then the universal player sets $s_u > 0$, and $s_u < 0$ for $u = 1$) and the absolute value encodes the magnitude of this preferred choice. If the existential player follows the choice, the universal player loses $|s_u|$ points; otherwise he gains the same amount.

Our interest in the score game is that the universal winning strategies can be transformed into Q-NS/Q-SOS refutations.

▶ **Proposition 3.5.** *A QBF $\mathcal{Q}.\varphi$ is false if and only if the universal player has a winning strategy in the score game for $\mathcal{Q}.\varphi$ (in either variant).*

Proof. Let the QBF $\mathcal{Q}.\varphi$ be false. Then there exists a winning strategy $\tau = (\tau_u)_{u \in \mathrm{vars}_\forall(\mathcal{Q})}$ for the universal player in the evaluation game. In the score game, on the universal variable u the universal player plays depending on the total score S up to this point and τ. The preferred choice of the universal player is τ_u and he assigns to u score

$$s_u = \begin{cases} 0 & \text{if } S = 1\ , \\ (1-2\tau_u)(1-S) & \text{otherwise}\ . \end{cases}$$

If in each step of the game the universal player gets always his preference τ_u, then the matrix φ is falsified (since τ is a winning strategy in the evaluation game). Otherwise, let u^* be the first variable where the universal player does not get his preference and S^* the total score before deciding the value for u. Since, by assumption the universal player does not get his preference but the value $1 - \tau_{u^*}$ instead, then, after setting u^*, the total score is

$$S^* + s_{u^*}(2(1-\tau_{u^*}) - 1) = 1\ . \tag{3}$$

At this moment the universal player has essentially just won since he can set all following scores to 0 and the final total score of the game will be 1.

For the other direction, if the universal player has a winning strategy in the score game on $\mathcal{Q}.\varphi$ then he wins also against the case when the existential player makes the scores negative in each moment of the game. In this case the resulting assignment must falsify the matrix φ since the universal player is using a winning strategy. As such, this strategy is also a winning strategy for the universal player in the usual evaluation game and the QBF $\mathcal{Q}.\varphi$ is false. ◀

We require the universal strategy for each universal variable u to be expressed as a polynomial in all variables to the left of u in the quantifier prefix. The *size* of a universal strategy in the score game is then the sum of the number of monomials in all the s_u.

▶ **Theorem 3.6.** *Let π be a shortest* Q-SOS *(resp.* Q-NS*) refutation of $\mathcal{Q}.\varphi$ with respect to its Q-size. Then* $\mathrm{qsize}(\pi)$ *equals the size of the shortest universal winning strategy in the score game in* VARIANT *1 (resp.* VARIANT *2) on $\mathcal{Q}.\varphi$.*

Proof. Let $U = \mathrm{vars}_\forall(\mathcal{Q})$, let S be the size of the shortest universal winning strategy in the score game on $\mathcal{Q}.\varphi$, and let π be the LHS of a shortest Q-SOS (resp. Q-NS) refutation written as

$$\sum_{p\in\mathrm{enc}(\varphi)} q_p p + \sum_{u\in\mathrm{vars}_\forall(\mathcal{Q})} q_u(1-2u) + q = -1 \ . \tag{4}$$

To show that $\mathrm{qsize}(\pi) \geqslant S$ consider the universal strategy setting $s_u = q_u$. This is a winning strategy in the score game, i.e. for every total assignment α the universal player wins the score game. Indeed, if α falsifies φ then the universal player wins automatically (in both versions of the game). Assume then that α satisfies φ, that is for every $p \in \mathrm{enc}(\varphi)$, $p|_\alpha = 0$. Therefore, π evaluated at α is the same as $\sum_{u\in U} q_u(1-2u) + q$ evaluated at α. The expression $\sum_{u\in U} q_u(1-2u)$ evaluates under α to -1 if π is a Q-NS refutation or to $\leqslant -1 < 0$ if π is a Q-SOS refutation. Therefore the total score when playing the game given the total assignment α is

$$\sum_{u\in U} s_u(2u-1) = \left(\sum_{u\in U} q_u(2u-1)\right)\Big|_\alpha = -\left(\sum_{u\in U} q_u(1-2u)\right)\Big|_\alpha$$

and this latter sum equals 1 if π is a Q-NS refutation or it is > 0 if π is a Q-SOS refutation. In other words the universal player in such cases wins using the scores.

To prove $\mathrm{qsize}(\pi) \leqslant S$, we consider the cases where π is a Q-NS or Q-SOS refutation.

Case 1: π is a Q-NS refutation. Let $(s_u)_{u\in U}$ be a shortest universal winning strategy for the score game in VARIANT 2 on $\mathcal{Q}.\varphi$ and let q_u be the polynomial computing s_u as a function of the variables left of u in $\mathcal{Q}$. In particular, on all Boolean assignments α satisfying the matrix φ, the universal player wins because the total score is 1, i.e. $\left(\sum_{u\in U} q_u(1-2u)\right)\big|_\alpha = -1$ and therefore $\sum_{u\in U} q_u(1-2u) + 1$ is in the ideal generated by the polynomials in $\mathrm{enc}(\varphi)$ (this follows from Fact 2.1). This gives a Q-NS refutation of $\mathcal{Q}.\varphi$ with a Q-size of at most S.

Case 2: π is a Q-SOS refutation. Let $(s_u)_{u\in U}$ be a shortest universal winning strategy for the score game in VARIANT 2 on $\mathcal{Q}.\varphi$ and let q_u be the polynomial computing s_u as a function of the variables left of u in $\mathcal{Q}$. For an assignment α, let $\mathrm{score}(\alpha) = \sum_{u\in U} s_u\ (2u-1)|_\alpha$. For every Boolean assignment α satisfying the matrix φ, since $(s_u)_u$ is a winning strategy, we have $\mathrm{score}(\alpha) > 0$. That is for $c = \frac{1}{2}\min_{\alpha\models\varphi}\mathrm{score}(\alpha)$ we have

$$\sum_{u\in U} \frac{s_u}{c}(1-2u) = -\frac{\mathrm{score}(\alpha)}{c} < -1 \ . \tag{5}$$

In this way, for every α satisfying φ, $1 - \frac{\mathrm{score}(\alpha)}{c} < 0$. Let $q = -\sum_{\alpha\models\varphi}\left(1 - \frac{\mathrm{score}(\alpha)}{c}\right)\chi_\alpha(\boldsymbol{v})$, where $\chi_\alpha(\boldsymbol{v})$ is the monomial which evaluates to 1 if the variables $\boldsymbol{v}$ are set according to α, and 0 on any other Boolean assignment. Modulo the polynomials $v^2 - v$, q is a sum of squares, hence to conclude it is enough to notice that the polynomial $\sum_{u\in U}\frac{1}{c}q_u(1-2u) + q + 1$ evaluates to 0 on every assignment satisfying $\mathrm{enc}(\varphi)$, hence it belongs to the ideal generated by the polynomials in $\mathrm{enc}(\varphi)$ (this follows from Fact 2.1). This gives a Q-SOS refutation of $\mathcal{Q}.\varphi$ having the same qsize as the strategy $(s_u)_u$. ◀

▶ **Corollary 3.7** (completeness). Q-NS/Q-SA/Q-SOS *are complete.*

Proof. Proposition 3.5 and Theorem 3.6 immediately imply the completeness of Q-NS. As Q-NS refutations are special cases of Q-SA and Q-SOS refutations their completeness also follows. ◀

3.2 Upper bounds in Q-SOS via the score game

Due to Theorem 3.6, the score game can be used to obtain bounds on the qsize of Q-SOS refutations. The advantage is being able to argue directly on countermodels (without reference to the syntactic representation of the matrix). To that end, we use QBFs $Q\text{-}C_n$ from [16], which are defined via their countermodel computed by a family of circuits C_n.

▶ **Definition 3.8** ($Q\text{-}C_n$ [16]). *Let n be an integer and C_n be a circuit with inputs $x_1, \dots, x_n$ and a single output. We define*

$$Q\text{-}C_n = \exists x_1 \cdots \exists x_n \forall u \exists t_1 \cdots \exists t_m . \big(u \leftrightarrow C_n(x_1, \dots, x_n)\big) \ ,$$

where the additional variables t_i are used for a Tseitin-encoding of the circuit C_n into CNF (the i^{th} node in C_n is represented by variable t_i, and if, e.g., node i is an $\wedge$-gate between nodes j and k, then we have clauses encoding $t_i \leftrightarrow (t_j \wedge t_k)$, and similarly for $\vee$ and $\neg$ gates).

For the $Q\text{-}C_n$ formulas, the only countermodel sets u to $C_n(x_1, ..., x_n)$. Here, we are specifically interested in choosing circuits C_n that compute $\mathsf{Majority}_n$. A circuit calculating $\mathsf{Majority}_n$ evaluates to true, if and only if at least half of the n input variables are set to true. We show that $\mathsf{Q\text{-}Majority}_n$ has short Q-SOS refutations in the qsize measure.

▶ **Proposition 3.9.** *$Q\text{-}\mathsf{Majority}_n$ has Q-SOS refutations of linear* qsize.

Proof. In the score game for $\mathsf{Q\text{-}Majority}_n$, set $s_u = -x_1 - \cdots - x_n + \frac{n}{2} - \frac{1}{4}$. The choice of the constant $\frac{1}{4}$ is somewhat arbitrary, but should be between 0 and $\frac{1}{2}$. As there is only a single universal variable, this defines a complete strategy for the universal player. We show that s_u is a winning strategy. For an arbitrary assignment α, $s_u|_\alpha < 0$ if and only if at least half of the existential x_i variables are set to 1, otherwise $s_u|_\alpha > 0$. As such, the total score of the score game on α is $\big(-x_1 - \cdots - x_n + \frac{n}{2} - \frac{1}{4}\big)(2u-1)\big|_\alpha$. This is negative only if either $u = 1$ and at least half of the x_i equal 1 or $u = 0$ and less then half of the x_i equal 1, i.e. if the matrix is satisfied, the score is positive.

This strategy has size $n + 1$ and degree 1, hence, by Theorem 3.6, there exists a Q-SOS refutation of $\mathsf{Q\text{-}Majority}_n$ with a qsize linear in n. ◀

It is known that $\mathsf{Q\text{-}Majority}_n$ requires Q-PC refutations of exponential qsize [21], therefore the previous result yields the exponential separation between Q-PC and Q-SOS in Figure 1.

3.3 From existential Q-degree to Q-size

In various proof systems, strong enough lower bounds on the degree/width of proofs immediately imply non-trivial lower bounds on proof size. This happens for instance in propositional proof systems such as Resolution [8], Polynomial Calculus [31], Sherali-Adams and Sum-of-Squares [2]; and in QBF proof systems as well, for instance in QU-Resolution [13], and Q-PC [21]. It turns out that a very similar statement holds in Q-SOS between the existential Q-degree ($\mathrm{qdeg}_\exists(\cdot)$, see Definition 3.1) and Q-size ($\mathrm{qsize}(\cdot)$, see Definition 3.1).

▶ **Theorem 3.10.** *Let $Q.\varphi$ be a false QBF with n variables that has a Q-SOS refutation of* qsize *s. Then it has a Q-SOS refutation of* $\mathrm{qdeg}_\exists$ $O(\sqrt{n \log s})$.

The argument is similar to the proof of the analogous size-width inequality for Resolution from [8]. The main difference is the proof of the lemma below showing how to combine a proof of $\mathrm{qdeg}_\exists$ $k-1$ of $\mathcal{Q}.\varphi|_{x=1}$ and a proof of $\mathrm{qdeg}_\exists$ k of $\mathcal{Q}.\varphi|_{x=0}$ into a proof of $\mathrm{qdeg}_\exists$ k of $\mathcal{Q}.\varphi$. This is done using the score game from the previous section.

▶ **Lemma 3.11.** *Let $Q.\varphi$ be a false QBF and $x \in \mathrm{vars}_\exists(\mathcal{Q})$. If there is a* Q-SOS *refutation π_1 of $Q.\varphi|_{x=1}$ with $\mathrm{qdeg}_\exists(\pi_1) \leqslant k-1$, and a* Q-SOS *refutation π_0 of $Q.\varphi|_{x=0}$ with $\mathrm{qdeg}_\exists(\pi_0) \leqslant k$, then there is a* Q-SOS *refutation π of $\mathcal{Q}.\varphi$ with $\mathrm{qdeg}_\exists(\pi) \leqslant k$.*

Proof. We consider the equivalent representation of the proofs as universal strategies in the score game. Let q_1 be the final score in π_1 and q_0 be the final score in π_0. Let c be the maximum of $|q_0|+1$ over all assignments, and d be the smallest *positive* value that q_1 can take (or 1 if q_1 does not take positive values). For each $u \in \mathrm{vars}_\forall(\mathcal{Q})$, let $q_u = x \cdot q_u^1 + \frac{d}{c} q_u^0$, where q_u^0 and q_u^1 are the polynomials for u in π_0 and π_1. Combining these q_u yields a strategy π with final score $q = x \cdot q_1 + \frac{d}{c} q_0$ and $\mathrm{qdeg}_\exists(\pi) \leqslant k$.

We still need to argue that π is a universal winning strategy, i.e. that on every assignment α satisfying φ we have $q(\alpha) > 0$. If α sets $x = 0$, then it satisfies $\varphi|_{x=0}$, so $q_0(\alpha) > 0$ due to the correctness of π_0. But if $x = 0$ then $q = 0 \cdot q_1 + \frac{d}{c} q_0 > 0$. If α sets $x = 1$, then it satisfies $\varphi|_{x=1}$, so $q_1(\alpha) > 0$ due to the correctness of π_1. By the definitions of c and d, we have $c > -q_0(\alpha)$ and $d \leqslant q_1(\alpha)$. This means that $\frac{d}{c} q_0 > -d \geqslant -q_1(\alpha)$ and $q = 1 \cdot q_1 + \frac{d}{c} q_0 > 0$. ◀

▶ **Lemma 3.12.** *Let $d, n, b \in \mathbb{N}^{\geqslant 0}$ and $Q.\varphi$ be a false QBF. Let π be a* Q-SOS *refutation of $Q.\varphi$ so that its q_u polynomials contain, in total, fewer than $(1 - \frac{d}{2n})^{-b}$ monomials of existential degree $> d$. Then there is a* Q-SOS *refutation π' of $Q.\varphi$ with $\mathrm{qdeg}_\exists(\pi') \leqslant d + b$.*

The proof of this lemma is virtually identical to the proof of [8, Theorem 3.5]. Informally, the proof is by an inductive argument on n and b, considering the *high-degree* monomials to be the ones with $\mathrm{qdeg}_\exists$ at least d. By a counting argument there will be a literal x appearing in at least a $\frac{d}{2n}$ fraction of them. Restricting by $x = 0$ and $x = 1$, we have that the first restriction eliminates at least $\frac{d}{2n}$ of the high-degree monomials, while the second eliminates one variable. Then using the inductive hypothesis and Lemma 3.11 concludes the argument.

Given the two lemmas above it is immediate to prove Theorem 3.10.

Proof of Theorem 3.10. Set $b = d = \sqrt{2n \log s}$ and observe that $s < (1 - \frac{d}{2n})^{-b}$. The number of high-degree monomials in the refutation is smaller than its total number of monomials s, so we can apply Lemma 3.12 and get a refutation of $\mathrm{qdeg}_\exists(b + d) \in O(\sqrt{n \log s})$. ◀

4 Lower bounds via strategy extraction in the evaluation game

In QBF proof systems, strategy extraction is a welcome and ubiquitous feature. Informally, given a refutation of a false QBF, strategy extraction allows to represent in some computational model a winning strategy of the universal player in the *evaluation* game. Different QBF proof systems give rise to strategy extraction in different computational models. For example, from QU-Res refutations we get *unified decision lists* [13] and from Frege+∀-reduction refutations we get NC_1 circuits [16] .

In this section, we show that Q-SOS/Q-SA/Q-NS also admit strategy extraction, using *polynomial threshold functions* (PTF) as the computational model (Theorem 4.3). We use this fact to prove a lower bound in Q-SOS (Corollary 4.4) and a p-simulation of Q-SOS by Q-TC_0-Frege (Corollary 4.5).

▶ **Definition 4.1** (polynomial threshold function)**.** *A Boolean function $f : \{0,1\}^n \to \{\pm 1\}$ is computed by a* polynomial threshold function (PTF)*, if there exists some n-variate polynomial $p \in \mathbb{Q}[\boldsymbol{x}]$ such that $f(\boldsymbol{x}) = \mathrm{sign}(p(\boldsymbol{x}))$. The* size *of the PTF is the number of monomials in p and the* degree *of the PTF is the degree of p.*

▶ Remark 4.2 (On the PTF degree of parity). It is well known that $f(\boldsymbol{x}) = x_1 \oplus x_2 \oplus \cdots \oplus x_n$ cannot be computed by PTFs of degree less than n. We recall briefly the argument (see also for instance [46]). Let $f(\boldsymbol{x}) = \operatorname{sign}(p(\boldsymbol{x}))$ with p a n-variate polynomial of degree d. Since f is symmetric, there exists a symmetric n variate polynomial p' such that $f(\boldsymbol{x}) = \operatorname{sign}(p'(\boldsymbol{x}))$ and $\deg(p') \leqslant \deg(p) = d$. Since p' is symmetric, there exists a univariate polynomial p'' such that $p''(x_1+\cdots+x_n) = p'(\boldsymbol{x})$ and $\deg(p'') = \deg(p')$. In other words, $f(\boldsymbol{x}) = \operatorname{sign}(p''(x_1+\cdots+x_n))$. To conclude it is enough to notice that on $0, 1, \ldots, n$ the polynomial p'' must be alternating signs and hence it must have at least n real roots; therefore $d \geqslant \deg(p'') \geqslant n$.

Given a Q-SOS/Q-SA/Q-NS refutation $\sum_{p\in\operatorname{enc}(\varphi)} q_p p + \sum_{u\in\operatorname{vars}_\forall(\mathcal{Q})} q_u(1-2u) + q + 1 = 0$ of a false QBF $\mathcal{Q}.\varphi$, we claim that for every universal variable u, $\operatorname{sign}(q_u)$ computes a strategy for $1-2u$.[1] This can then be easily transformed into a strategy for u through a linear output transformation mapping -1 to 1 and 1 to 0. Per definition, for every universal variable u, $\operatorname{sign}(q_u)$ is a PTF. The size of the extracted strategy is the sum of the sizes of the PTFs and, as such, the Q-size of the refutation. Analogously, the degree of the extracted strategy, i.e. the maximum degree of the PTFs, equals the total Q-degree of the refutation.

▶ **Theorem 4.3.** *Let $\sum_{p\in\operatorname{enc}(\varphi)} q_p p + \sum_{u\in\operatorname{vars}_\forall(\mathcal{Q})} q_u(1-2u) + q + 1 = 0$ be a* Q-SOS/ Q-SA/ Q-NS *proof of a false QBF $\mathcal{Q}.\varphi$. Then the universal strategy that maps each universal variable $u \in \operatorname{vars}_\forall(\mathcal{Q})$ to $\frac{1-\operatorname{sign}(q_u)}{2}$ is a countermodel (i.e. falsifies φ).*

Proof. The syntactic restriction of countermodels, i.e. that each variable only depends on variables left of it in the quantifier prefix $\mathcal{Q}$, holds by definition of q_u.

For every universal variable u played according to the strategy, we have $q_u(1-2u) = q_u \operatorname{sign}(q_u) \geqslant 0$. As such, $\sum_{u\in\operatorname{vars}_\forall(\mathcal{Q})} q_u(1-2u) + q + 1 \geqslant 1$. Hence, $\sum_{p\in\operatorname{enc}(\varphi)} q_p p \leqslant -1$, which is only possible if the matrix φ is not satisfied (otherwise $q_p p = 0$ for all $p \in \operatorname{enc}(\varphi)$). ◀

To exemplify the strategy extraction technique for Q-SOS, we use Remark 4.2 and Theorem 4.3 to prove that the Parity formulas [18]

$$\mathsf{Parity}_n = \exists x_1 \cdots \exists x_n \forall u \exists t_1 \cdots \exists t_n.\ (t_1 \leftrightarrow x_1) \wedge (u \nleftrightarrow t_n) \wedge \bigwedge_{i=2}^{n} (t_i \leftrightarrow t_{i-1} \oplus x_i).$$

are exponentially hard for Q-SOS. Notice that the only winning strategy for the universal player is to set $u = x_1 \oplus \cdots \oplus x_n$.

▶ **Corollary 4.4.** *Every* Q-SOS *refutation of Parity_n requires Q-size* $\exp(\Omega(n))$.

Proof. Let $d := \operatorname{qdeg}_\exists(\pi)$ and apply strategy extraction (Theorem 4.3) to get a PTF of degree d that computes $u = \bigoplus_{i=1}^{n} x_i$. By Remark 4.2, its degree is at least n, so $d \geqslant n$. Let $s := \operatorname{qsize}(\pi)$ and apply Theorem 3.10 to obtain $n \leqslant d = O(\sqrt{n \log s})$ and therefore $s = \exp(\Omega(n))$. Theorem 3.10 can be applied here, because Parity_n only has a single universal variable and, as such, its existential Q-degree equals its total Q-degree. ◀

As a second consequence of Theorem 4.3, strategy extraction can also be used to embed Q-SOS into more powerful systems, in this case Q-TC_0-Frege. Q-TC_0-Frege is the TC_0-Frege system with an added universal reduction rule. The Q-size of a Q-TC_0-Frege refutation is the sum of the number of symbols of all lines involved in a $\forall$-reduction step.

▶ **Corollary 4.5.** *Q-TC_0-Frege p-simulates* Q-SOS *in the* qsize *measure.*

[1] We use the convention that $\operatorname{sign}(0) = +1$.

5 Lower bounds via Q-pseudo-expectation

In the propositional setting, the notion of pseudo-expectation is the standard tool to obtain degree lower bounds for SOS, see for instance [40]. Thanks to the size-degree relation for SOS [2], degree lower bounds also give size lower bounds.

Inspired by the propositional notion of pseudo-expectation, we give a notion of pseudo-expectation for Q-SOS and use it to prove lower bounds on the $\mathrm{qdeg}_\exists$ of Q-SOS refutations. In particular, for a false QBF $\mathcal{Q}.\varphi$ and a Q-SOS expression π of the form

$$\sum_{p\in\mathrm{enc}(\varphi)} q_p p + \sum_{u\in\mathrm{vars}_\forall(\mathcal{Q})} q_u(1-2u) + q + 1 \ , \tag{6}$$

where all the variables in q_u are on the left of u in $\mathcal{Q}$ and q is a sum of squares, we consider *witnesses* that $\pi \neq 0$, i.e. that π *is not* a Q-SOS refutation of $\mathcal{Q}.\varphi$. In analogy to the propositional case, we call the witnesses we construct *Q-pseudo-expectations*.

▶ **Definition 5.1** (Q-pseudo-expectation in Q-SOS). *Given $\mathcal{Q}.\varphi$ and a* Q-SOS *expression π as in eq.* (6)*, a* Q-pseudo-expectation *for $\mathcal{Q}.\varphi$ and π is a linear function $\tilde{\mathbb{E}}\colon \mathbb{Q}[V \cup \overline{V}] \to \mathbb{R}$ such that:*

1. $\tilde{\mathbb{E}}[1] = 1$*;*
2. $\tilde{\mathbb{E}}[q + \sum_{p\in\mathrm{enc}(\varphi)} q_p p] \geqslant 0$*;*
3. $\tilde{\mathbb{E}}[\sum_{u\in\mathrm{vars}_\forall(\mathcal{Q})} q_u(1-2u)] \geqslant 0$.

In the propositional context, a single pseudo-expectation for SOS typically targets a fixed degree d and has properties similar to 1.–3. above but for arbitrary q_p and sum-of-squares q such that the degree of $q_p p$ and q are at most d. In this way, a single pseudo-expectation rules out the possibility of *any* small-degree SOS refutation. There are exceptions to this general approach, for instance the pseudo-expectations used in [42] that are targeting all SOS proofs over a certain set of monomials, *but* we are not aware of degree lower bounds in SOS proved by constructing a family of pseudo-expectations each tailored to a specific set of polynomials (i.e. as in Definition 5.1 but without the condition in item 3.). In the QBF context, this is what we do. To rule out small $\mathrm{qdeg}_\exists$ Q-SOS refutations we use a family of pseudo-expectations, each targeting *one* possible candidate Q-SOS proof, i.e. an expression as in eq. (6). We formalise this approach in Theorem 5.2 and exemplify it in Theorem 5.3.

▶ **Theorem 5.2.** *Given a QBF $\mathcal{Q}.\varphi$, if for every* Q-SOS *expression π as in eq.* (6) *with $\mathrm{qdeg}_\exists(\pi) < d$ there is a pseudo-expectation for $\mathcal{Q}.\varphi$ and π, then every* Q-SOS *refutation of $\mathcal{Q}.\varphi$ has* $\mathrm{qdeg}_\exists$ *at least d.*

Proof. If there was a Q-SOS refutation $\pi = 0$ of $\mathcal{Q}.\varphi$ with $\mathrm{qdeg}_\exists(\pi) < d$, then taking the pseudo-expectation $\tilde{\mathbb{E}}$ for $\mathcal{Q}.\varphi$ and π we get $\tilde{\mathbb{E}}[\pi] = \tilde{\mathbb{E}}[0] = 0$. Notice that necessarily $\tilde{\mathbb{E}}[0] = 0$, by linearity and the identity $\tilde{\mathbb{E}}[0+0] = \tilde{\mathbb{E}}[0]$. On the other hand, again by linearity, and the properties of $\tilde{\mathbb{E}}$, we get $\tilde{\mathbb{E}}[\pi] \geqslant 1$. ◀

We apply the Q-pseudo-expectation technique to show a $\mathrm{qdeg}_\exists$ lower bound on Q-SOS refutations of the $\mathsf{Equality}_n$ formulas [11] where

$$\mathsf{Equality}_n = \exists x_1 \cdots x_n \forall u_1 \cdots u_n \exists t_1 \cdots t_n . \bigwedge_{i=1}^{n} (t_i \to (x_i \nleftrightarrow u_i)) \wedge \bigvee_{i=1}^{n} t_i \ .$$

For the $\mathrm{qdeg}_\exists$ lower bound we only use the fact that every assignment satisfying the matrix sets $u_i \nleftrightarrow x_i$ for some i.

$\mathsf{Equality}_n$ is a quite simple QBF. Hardness for Q-SOS might suggest that the system is weak. $\mathsf{Equality}_n$ is also hard for QU-Res and Q-PC [11], but easy in Q-depth-d Frege [11], so the hardness appears to only stem from the expressiveness of the objects used. Similarly, Q-SOS on depth-d arithmetic circuits (instead of polynomials) would shortly prove $\mathsf{Equality}_n$, but if polynomials are represented explicitly as sums of monomials, $\mathsf{Equality}_n$ becomes hard.

▶ **Theorem 5.3.** *Every* Q-SOS *refutation* π *of* $\mathsf{Equality}_n$ *has* $\mathrm{qdeg}_\exists(\pi) \geqslant n$ *and* $\mathrm{qsize}(\pi) \geqslant \exp(\Omega(n))$.

Proof. Let $\mathcal{Q}.\varphi$ be the QBF encoding of $\mathsf{Equality}_n$. First notice that, thanks to Theorem 3.10, it is enough to prove the $\mathrm{qdeg}_\exists$ lower bound. (We comment that to do this, the strategy extraction technique from Section 4 would not work here.)

Assume, towards a contradiction, that there is a Q-SOS refutation of $\mathcal{Q}.\varphi$ of $\mathrm{qdeg}_\exists(\pi) < n$:

$$\sum_{p\in\mathrm{enc}(\varphi)} q_p p + \sum_{u\in\mathrm{vars}_\forall(\mathcal{Q})} q_u(1-2u) + q + 1 = 0 \ . \tag{7}$$

Let π the LHS of eq. (7). We construct a Q-pseudo-expectation for $\mathcal{Q}.\varphi$ and π. This, by Theorem 5.2, implies the wanted contradiction.

Given $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \in \{0,1\}^n$, let $\boldsymbol{x} \mapsto \boldsymbol{\alpha}$ be the Boolean assignment setting x_i to α_i for each $i \in [n]$. We define analogously $\boldsymbol{u} \mapsto \boldsymbol{\alpha}$ and $\boldsymbol{t} \mapsto \boldsymbol{\alpha}$. Given $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \{0,1\}^n$, let $\boldsymbol{\alpha} \oplus \boldsymbol{\beta}$ be the vector whose ith entry is the sum $\alpha_i + \beta_i \pmod 2$.

Let $h = \sum_{u\in\mathrm{vars}_\forall(\mathcal{Q})} q_u(1-2u)$. We have that

$$\sum_{\boldsymbol{\alpha},\boldsymbol{\beta}\in\{0,1\}^n} h|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha},\ \boldsymbol{u}\mapsto\boldsymbol{\beta}} = 0 \ . \tag{8}$$

Let $\boldsymbol{\gamma} \in \{0,1\}^n$ be the assignment maximizing $\sum_{\boldsymbol{\alpha}\in\{0,1\}^n} h|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha},\ \boldsymbol{u}\mapsto\boldsymbol{\gamma}}$. By eq. (8),

$$\sum_{\boldsymbol{\alpha}\in\{0,1\}^n} h|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha},\ \boldsymbol{u}\mapsto\boldsymbol{\gamma}} \geqslant 0 \ . \tag{9}$$

We define our candidate Q-pseudo-expectation as

$$\tilde{\mathbb{E}}(p) = 2^{-n} \sum_{\boldsymbol{\alpha}\in\{0,1\}^n} p|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha},\ \boldsymbol{u}\mapsto\boldsymbol{\gamma},\ \boldsymbol{t}\mapsto\boldsymbol{\alpha}\oplus\boldsymbol{\gamma}} \ ,$$

and we prove that $\tilde{\mathbb{E}}$ satisfies all the conditions of the definition of Q-pseudo-expectation. The definition of $\tilde{\mathbb{E}}$ immediately implies that $\tilde{\mathbb{E}}[1] = 1$, and $\tilde{\mathbb{E}}[h] \geqslant 0$ follows from eq. (9). Let $g = q + \sum_{p\in\mathrm{enc}(\varphi)} q_p p$. To conclude the argument we need to prove that $\tilde{\mathbb{E}}[g] \geqslant 0$.

Since $g = -h - 1$ and by construction h cannot contain t_i variables, the polynomial g contains only x_i and u_i variables. That is,

$$\tilde{\mathbb{E}}[g] = 2^{-n} \sum_{\boldsymbol{\alpha}\in\{0,1\}^n} g|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha},\ \boldsymbol{u}\mapsto\boldsymbol{\gamma}} \overset{(\star)}{=} 2^{-n} \sum_{\boldsymbol{\alpha}\in\{0,1\}^n} g|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}\oplus\boldsymbol{\gamma},\ \boldsymbol{u}\mapsto\boldsymbol{\gamma}} \ ,$$

where the last equality $(\star)$ follows as $\boldsymbol{\alpha} \oplus \boldsymbol{\gamma}$ ranges over all $\{0,1\}^n$ just in a different order.

Let g' be the polynomial such that for every $\boldsymbol{\alpha} \in \{0,1\}^n$, $g'|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}} = g|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}\oplus\boldsymbol{\gamma},\ \boldsymbol{u}\mapsto\boldsymbol{\gamma}}$. The polynomial g' is constructed from $g|_{\boldsymbol{u}\mapsto\boldsymbol{\gamma}}$ replacing every occurrence of x_i by $1 - x_i$ (resp. x_i) if $\gamma_i = 1$ (resp. $\gamma_i = 0$). Crucially, g' has degree at most $n-1$, since $\deg(g') \leqslant \deg_\exists(g) = \deg_\exists(-h-1) \leqslant \mathrm{qdeg}_\exists(\pi) \leqslant n-1$. That is, for each monomial m in g', there is some variable x_i not appearing in it, and m gets the same value on every pair $\boldsymbol{\alpha}, \boldsymbol{\alpha}'$ where $\boldsymbol{\alpha} \in \{0,1\}^n$ and

$\boldsymbol{\alpha}'$ is identical to $\boldsymbol{\alpha}$ except in position i. This implies $\sum_{\boldsymbol{\alpha}\in\{0,1\}^n}(-1)^{|\boldsymbol{\alpha}|}\ m|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}} = 0$, where $|\boldsymbol{\alpha}|$ denotes the 1-norm of $\boldsymbol{\alpha}$, the sum of all 1s in it. Summing over all monomials in g', we get

$$\begin{aligned}
0 &= \sum_{\boldsymbol{\alpha}\in\{0,1\}^n} (-1)^{|\boldsymbol{\alpha}|}\ g'|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}} \\
&= \sum_{\boldsymbol{\alpha}\in\{0,1\}^n} g'|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}} + \sum_{\boldsymbol{\alpha}\in\{0,1\}^n} ((-1)^{|\boldsymbol{\alpha}|}-1)\ g'|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}} \\
&= \sum_{\boldsymbol{\alpha}\in\{0,1\}^n} g|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}\oplus\boldsymbol{\gamma},\ \boldsymbol{u}\mapsto\boldsymbol{\gamma}} + \sum_{\boldsymbol{\alpha}\in\{0,1\}^n} ((-1)^{|\boldsymbol{\alpha}|}-1)\ g'|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}} \\
&= 2^n\tilde{\mathbb{E}}[g] + \sum_{\boldsymbol{\alpha}\in\{0,1\}^n} ((-1)^{|\boldsymbol{\alpha}|}-1)\ g'|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}}\ .
\end{aligned}$$

Hence, to conclude that $\tilde{\mathbb{E}}[g] \geqslant 0$ it is enough to show that

$$\sum_{\boldsymbol{\alpha}\in\{0,1\}^n} ((-1)^{|\boldsymbol{\alpha}|}-1)\ g'|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}} \leqslant 0\ . \tag{10}$$

For $\boldsymbol{\alpha}$s such that $|\boldsymbol{\alpha}|$ is even, the coefficient in front of $g'|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}}$ is 0, while for $\boldsymbol{\alpha}$s such that $|\boldsymbol{\alpha}|$ is odd, the coefficient in front of $g'|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}}$ is -2. That is, to prove eq. (10), it suffices to show that for $\boldsymbol{\alpha}$s such that $|\boldsymbol{\alpha}|$ is odd, $g'|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}} \geqslant 0$. By construction $g'|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}} = g|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}\oplus\boldsymbol{\gamma},\ \boldsymbol{u}\mapsto\boldsymbol{\gamma}}$, and since there are no t_i variables in g, $g|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}\oplus\boldsymbol{\gamma},\ \boldsymbol{u}\mapsto\boldsymbol{\gamma}} = g|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}\oplus\boldsymbol{\gamma},\ \boldsymbol{u}\mapsto\boldsymbol{\gamma},\boldsymbol{t}\mapsto\boldsymbol{\alpha}}$.

Now, if $|\boldsymbol{\alpha}|$ is odd, in particular $\boldsymbol{\alpha} \neq \mathbf{0}$ and $\boldsymbol{\alpha}\oplus\boldsymbol{\gamma} \neq \boldsymbol{\gamma}$. It is easy to check that the assignment $\boldsymbol{x}\mapsto\boldsymbol{\alpha}\oplus\boldsymbol{\gamma},\ \boldsymbol{u}\mapsto\boldsymbol{\gamma},\boldsymbol{t}\mapsto\boldsymbol{\alpha}$ sets to 0 (i.e. satisfies) all the polynomials $p \in \mathrm{enc}(\varphi)$. Since q is always non-negative on every assignment, we can conclude that $g|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}\oplus\boldsymbol{\gamma},\ \boldsymbol{u}\mapsto\boldsymbol{\gamma},\boldsymbol{t}\mapsto\boldsymbol{\alpha}} \geqslant 0$ and therefore $g'|_{\boldsymbol{x}\mapsto\boldsymbol{\alpha}} \geqslant 0$. ◀

6 Simulations

We now investigate how the algebraic QBF systems relate to each other and to other known QBF proof systems such as QU-Resolution and Q-PC and show the p-simulations of Figure 1.

▶ **Theorem 6.1.** Q-SOS *p-simulates* Q-SA *w.r.t. the* size *and* qsize *measures.*

Proof. It is well known that (degree $2d$) SOS p-simulates (degree d) SA (see for instance [40, Lemma 3.63]). The same argument works without change in the QBF setting: every variable v in a positive monomial can be substituted by v^2 summing a suitable multiple of $v^2 - v$. In this way, every positive monomial $\frac{a}{b}m$ with $a, b \in \mathbb{N}$ can be converted into $\frac{s_1^2+s_2^2+s_3^2+s_4^2}{b^2}m^2$ where s_1, s_2, s_3, s_4 are four integers that sum up to ab (they exists by Lagrange's Four Squares Theorem). This converts $\frac{a}{b}m$ into a sum of at most four squares with rational coefficients. ◀

In the argument above, notice that the only increment in degree is in the propositional part, hence, different from the propositional case, Q-SOS with $\mathrm{qdeg}_\exists\ d$ p-simulates Q-SA with $\mathrm{qdeg}_\exists\ d$. In the qsize measure, the converse also holds.

▶ **Theorem 6.2.** Q-SA *p-simulates* Q-SOS *w.r.t. the* qsize *measure.*

Proof. Every polynomial q that is non-negative on the Boolean assignments can be written as a (possibly exponentially large) sum of the form $\sum_\alpha q|_\alpha\, \chi_\alpha(\boldsymbol{v})$, where $\chi_\alpha(\boldsymbol{v})$ is a monomial that evaluates to 1 when the variables are set according to the Boolean assignment α and on any other Boolean assignment it is 0. In other words, every Q-SOS refutation can be written as a possibly exponentially larger Q-SA refutation. The exponential blow-up appears in the propositional part which is not accounted for in qsize. ◀

▶ **Theorem 6.3.** Q-SA *p-simulates* QU-*Res w.r.t. both* size *and* qsize.

Proof. (sketch) The argument in [37, Proposition 1 and Corollary 2] showing that degree d SA p-simulates width d Resolution can be easily adapted to the QBF setting. An alternative proof could be using the characterization of SA as *weighted* Resolution [25, Theorem 5.7]. Weighted Resolution is a proof system handling weighted clauses, i.e. pairs (C, w) with C a clause and $w \in \mathbb{Z}$ (or $\mathbb{Q}$). Adding to weighted Resolution the rule $\frac{(C \vee u, 2w)}{(C, w)}$, where all the variables in C appear left of u in the quantifier prefix will result in a system p-equivalent to Q-SA. QU-Res is then p-equivalent to weighted Resolution augmented with the rule above where all weights in the proofs are non-negative. ◀

▶ **Theorem 6.4.** Q-PC *p-simulates* Q-NS *w.r.t. the* size *and* qsize *measures.*

Proof. Let $\mathcal{Q}.\varphi$ be a false QBF with n variables and m clauses and π be a Q-NS refutation of $\mathcal{Q}.\varphi$ of the form

$$\sum_{p \in \mathrm{enc}(\varphi)} q_p p + \sum_{u \in \mathrm{vars}_\forall(\mathcal{Q})} q_u(1-2u) + 1 = 0 \ , \tag{11}$$

Lines in a Q-PC proof can be multiplied by arbitrary polynomials. Hence, we can obtain in Q-PC the sum $\sum_{p \in \mathrm{enc}(\varphi)} q_p p$ from the polynomials in $\mathrm{enc}(\varphi)$ in a polynomial number of steps. Let $\mathrm{vars}_\forall(\mathcal{Q})$ be $u_1, u_2, \ldots, u_n$. Due to the symbolic equality in eq. (11), this sum equals $-1 - \sum_{i=1}^{n} q_{u_i}(1-2u_i)$. We then use the $\forall$-reduction on u_n, then u_{n-1} etc. In the first step, restricting by $u_n = 1$ and $u_n = 0$, we get respectively

$$-1 - \sum_{i=1}^{n-1} q_{u_i}(1-2u_i) - q_{u_n} \text{ and } -1 - \sum_{i=1}^{n-1} q_{u_i}(1-2u_i) + q_{u_n} \ .$$

Adding them, we get $-1 - \sum_{i=1}^{n-1} q_{u_i}(1-2u_i)$. We repeat this process until we get rid of all universal variables and only the -1 remains. It is clear from the argument that this simulation only increases the size and qsize linearly. ◀

▶ **Theorem 6.5.** Q-SOS *p-simulates* Q-PC *w.r.t. the* size *and* qsize *measures.*

Proof. (sketch) The argument in [9, Lemma 3.1] showing that degree $2d$ SOS p-simulates degree d PC adapts easily to the QBF setting. The idea is that given a Q-PC derivation $p_1, \ldots, p_s$ we derive an algebraic expression for $-p_i^2$ which eventually for $i = s$ will give a Q-SOS refutation of $\mathcal{Q}.\varphi$. This is done inductively on i and is based on the following algebraic identities:

- sum rule (from p and q deduce $ap + bq$ with $a, b \in \mathbb{Q}$):
  $-(ap+bq)^2 = -2a^2p^2 - 2b^2q^2 + (ap - bq)^2$;
- product rule (from p deduce xp):
  $-(xp)^2 = -p^2 + (p - xp)^2 + 2p^2(x - x^2)$;
- $\forall$-reduction (from $p + qu$ deduce p):
  $-p^2 = -2(p+qu)^2 + (p+q)^2 - (q^2 + 2pq)(1-2u) + 2q^2(u^2 - u)$;
- $\forall$-reduction (from $p + qu$ deduce $p + q$):
  $-(p+q)^2 = -2(p+qu)^2 + p^2 - (q^2 + 2pq)(1-2u) + 2q^2(u^2 - u)$. ◀

▶ **Corollary 6.6.** Q-SOS *p-simulates* Q-NS *and is exponentially stronger, w.r.t. the* size *and* qsize *measures.*

Proof. The simulation follows from Theorem 6.5 and Theorem 6.4. The separation follows from Proposition 3.9 and Theorem 6.4 for the qsize measure, and from propositional separations [4, 34] for the size measure. ◀

7 Conclusion

In this work we defined semi-algebraic proof systems for QBF and initiated their proof complexity investigation. While our results already reveal an interesting picture in terms of simulations and lower and upper bounds, a number of questions remain that appear to be of interest for further research.

In the propositional setting Res and NS are incomparable proof systems. Are also Q-NS and QU-Res incomparable w.r.t. the qsize measure?

In Section 4 we showed how to express strategy extraction for Q-SOS using polynomial threshold functions. Although this suffices for lower bounds, it appears interesting to determine the correct computational model *characterizing* strategy extraction for Q-SOS and Q-NS in the same sense as the tight characterisations for QU-Res [13], Q-PC [21], and QBF Frege systems [16].

Finally, it would be interesting to determine the relationship of our new semi-algebraic QBF systems to the static expansion-based algebraic systems suggested in [29], which might turn out to be incomparable in strength.

References

1 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.

2 Albert Atserias and Tuomas Hakoniemi. Size-Degree Trade-Offs for Sums-of-Squares and Positivstellensatz Proofs. In *34th Computational Complexity Conference (CCC 2019)*, volume 137, pages 24:1–24:20, 2019. doi:10.4230/LIPICS.CCC.2019.24.

3 Albert Atserias, Massimo Lauria, and Jakob Nordström. Narrow proofs may be maximally long. In *29th Conference on Computational Complexity (CCC 2014)*, pages 286–297, 2014. doi:10.1109/CCC.2014.36.

4 Albert Atserias, Massimo Lauria, and Jakob Nordström. Narrow proofs may be maximally long. *ACM Trans. Comput. Logic*, 17(3):19:1–19:30, 2016. A preliminary version of this work appeared as [3]. doi:10.1145/2898435.

5 Valeriy Balabanov and Jie-Hong R. Jiang. Unified QBF certification and its applications. *Form. Methods Syst. Des.*, 41(1):45–65, 2012. doi:10.1007/s10703-012-0152-6.

6 Valeriy Balabanov, Magdalena Widl, and Jie-Hong R. Jiang. QBF resolution systems and their proof complexities. In *Proc. Theory and Applications of Satisfiability Testing (SAT)*, pages 154–169, 2014. doi:10.1007/978-3-319-09284-3_12.

7 Boaz Barak, Fernando G.S.L. Brandao, Aram W. Harrow, Jonathan Kelner, David Steurer, and Yuan Zhou. Hypercontractivity, sum-of-squares proofs, and their applications. In *44th Annual ACM Symposium on Theory of Computing (STOC 2012)*, pages 307–326, 2012. doi:10.1145/2213977.2214006.

8 Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow – Resolution made simple. *J. ACM*, 48(2):149–169, 2001. doi:10.1145/375827.375835.

9 Christoph Berkholz. The Relation between Polynomial Calculus, Sherali-Adams, and Sum-of-Squares Proofs. In *35th Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 96, pages 11:1–11:14, 2018. doi:10.4230/LIPIcs.STACS.2018.11.

10 Olaf Beyersdorff. Proof complexity of quantified Boolean logic – a survey. In Marco Benini, Olaf Beyersdorff, Michael Rathjen, and Peter Schuster, editors, *Mathematics for Computation (M4C)*, pages 353–391. World Scientific, Singapore, 2022.

11 Olaf Beyersdorff, Joshua Blinkhorn, and Luke Hinde. Size, cost, and capacity: A semantic technique for hard random QBFs. In *Proc. Conference on Innovations in Theoretical Computer Science (ITCS'18)*, pages 9:1–9:18, 2018. doi:10.4230/LIPICS.ITCS.2018.9.

12 Olaf Beyersdorff, Joshua Blinkhorn, and Luke Hinde. Size, cost, and capacity: A semantic technique for hard random QBFs. *Logical Methods in Computer Science*, 15(1), 2019. A preliminary version of this work appeared as [11]. doi:10.23638/LMCS-15(1:13)2019.

13 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.

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

15 Olaf Beyersdorff, Ilario Bonacina, and Leroy Chew. Lower bounds: From circuits to QBF proof systems. In *Proc. ACM Conference on Innovations in Theoretical Computer Science (ITCS'16)*, pages 249–260, 2016. doi:10.1145/2840728.2840740.

16 Olaf Beyersdorff, Ilario Bonacina, Leroy Chew, and Jan Pich. Frege systems for quantified Boolean logic. *J. ACM*, 67(2):9:1–9:36, 2020. Preliminary versions of this work appeared as [15] and [23]. doi:10.1145/3381881.

17 Olaf Beyersdorff, Leroy Chew, and Mikolás Janota. New resolution-based QBF calculi and their proof complexity. *ACM Transactions on Computation Theory*, 11(4):26:1–26:42, 2019. doi:10.1145/3352155.

18 Olaf Beyersdorff, Leroy Chew, and Mikoláš Janota. Proof complexity of resolution-based QBF calculi. In *Proc. Symposium on Theoretical Aspects of Computer Science (STACS'15)*, pages 76–89. LIPIcs, 2015. doi:10.4230/LIPIcs.STACS.2015.76.

19 Olaf Beyersdorff, Leroy Chew, Meena Mahajan, and Anil Shukla. Understanding cutting planes for QBFs. *Inf. Comput.*, 262:141–161, 2018. doi:10.1016/j.ic.2018.08.002.

20 Olaf Beyersdorff, Luke Hinde, and Ján Pich. Reasons for hardness in QBF proof systems. *ACM Transactions on Computation Theory*, 12(2):10:1–10:27, 2020. doi:10.1145/3378665.

21 Olaf Beyersdorff, Tim Hoffmann, Kaspar Kasche, and Luc Nicolas Spachmann. Polynomial calculus for quantified boolean logic: Lower bounds through circuits and degree. In *49th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 306, pages 27:1–27:15, 2024. doi:10.4230/LIPICS.MFCS.2024.27.

22 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, 2nd edition*, Frontiers in Artificial Intelligence and Applications. IOS press, 2021. doi:10.3233/FAIA201015.

23 Olaf Beyersdorff and Ján Pich. Understanding Gentzen and Frege systems for QBF. In *Proc. ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2016. doi:10.1145/2933575.2933597.

24 A. Blake. *Canonical expressions in boolean algebra*. PhD thesis, University of Chicago, 1937.

25 Ilario Bonacina, Maria Luisa Bonet, and Jordi Levy. Weighted, circular and semi-algebraic proofs. *J. Artif. Intell. Res.*, 79:447–482, 2024. A preliminary version of this work appeared as [26]. doi:10.1613/JAIR.1.15075.

26 Maria Luisa Bonet and Jordi Levy. Equivalence between systems stronger than resolution. In *23rd International Conference on Theory and Applications of Satisfiability Testing (SAT 2020)*, volume 12178, pages 166–181, 2020. doi:10.1007/978-3-030-51825-7_13.

27 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. doi:10.1006/jcss.2000.1726.

28 Sam Buss and Jakob Nordström. Proof complexity and SAT solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications, pages 233–350. IOS Press, 2021. doi:10.3233/FAIA200990.

29 Sravanthi Chede, Leroy Chew, Balesh Kumar, and Anil Shukla. Understanding Nullstellensatz for QBFs. *Electron. Colloquium Comput. Complex.*, TR23-129, 2023. URL: https:TR//eccc.weizmann.ac.il/report/2023/129/.

30 Hubie Chen. Proof complexity modulo the polynomial hierarchy: Understanding alternation as a source of hardness. *ACM Transactions on Computation Theory*, 9(3):15:1–15:20, 2017. doi:10.1145/3087534.

31 Matthew Clegg, Jeff Edmonds, and Russell Impagliazzo. Using the Groebner basis algorithm to find proofs of unsatisfiability. In *28th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 174–183, 1996. doi:10.1145/237814.237860.

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

33 William Cook, Collette R. Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987. doi:10.1016/0166-218X(87)90039-4.

34 Stefan S. Dantchev, Nicola Galesi, Abdul Ghani, and Barnaby Martin. Proof complexity and the binary encoding of combinatorial principles. *SIAM J. Comput.*, 53(3):764–802, 2024. A preliminary version appeared as [35]. doi:10.1137/20M134784X.

35 Stefan S. Dantchev, Abdul Ghani, and Barnaby Martin. Sherali-adams and the binary encoding of combinatorial principles. In *14th Latin American Symposium on Theoretical Informatics (LATIN 2020)*, volume 12118, pages 336–347, 2020. doi:10.1007/978-3-030-61792-9_27.

36 Stefan S. Dantchev and Barnaby Martin. Rank complexity gap for lovász-schrijver and sherali-adams proof systems. *Comput. Complex.*, 22(1):191–213, 2013. doi:10.1007/S00037-012-0049-1.

37 Stefan S. Dantchev, Barnaby Martin, and Mark Nicholas Charles Rhodes. Tight rank lower bounds for the Sherali-Adams proof system. *Theor. Comput. Sci.*, 410(21-23):2054–2063, 2009. doi:10.1016/J.TCS.2009.01.002.

38 Susanna F. de Rezende, Massimo Lauria, Jakob Nordström, and Dmitry Sokolov. The power of negative reasoning. In *36th Computational Complexity Conference (CCC)*, volume 200, pages 40:1–40:24, 2021. doi:10.4230/LIPICS.CCC.2021.40.

39 Uwe Egly, Florian Lonsing, and Magdalena Widl. Long-distance resolution: Proof generation and strategy extraction in search-based QBF solving. In *Proc. Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 291–308, 2013. doi:10.1007/978-3-642-45221-5_21.

40 Noah Fleming, Pravesh Kothari, and Toniann Pitassi. Semialgebraic proofs and efficient algorithm design. *Found. Trends Theor. Comput. Sci.*, 14(1-2):1–221, 2019. doi:10.1561/0400000086.

41 Mika Göös, Alexandros Hollender, Siddhartha Jain, Gilbert Maystre, William Pires, Robert Robere, and Ran Tao. Separations in proof complexity and TFNP. *J. ACM*, 71(4):26:1–26:45, 2024. doi:10.1145/3663758.

42 Tuomas Hakoniemi. Feasible interpolation for polynomial calculus and sums-of-squares. In *47th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 168, pages 63:1–63:14, 2020. doi:10.4230/LIPICS.ICALP.2020.63.

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

44 Hans Kleine Büning, Marek Karpinski, and Andreas Flögel. Resolution for quantified Boolean formulas. *Inf. Comput.*, 117(1):12–18, 1995. doi:10.1006/INCO.1995.1025.

45 Jean B. Lasserre. Global optimization with polynomials and the problem of moments. *SIAM J. Optim.*, 11(3):796–817, 2001. doi:10.1137/S1052623400366802.

46 Marvin Minsky and Seymour Papert. *Perceptrons – An introduction to computational geometry*. MIT Press, 1987. doi:10.7551/mitpress/11301.001.0001.

47 Ryan O'Donnell and Yuan Zhou. Approximability and proof complexity. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1537–1556. SIAM, 2013. doi:10.1137/1.9781611973105.111.

48 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.

49 John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965. doi:10.1145/321250.321253.

50 Hanif D. Sherali and Warren P. Adams. A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. *SIAM J. Discret. Math.*, 3(3):411–430, 1990. doi:10.1137/0403036.

51 Dmitry Sokolov. (semi)algebraic proofs over ±1 variables. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 78–90, 2020. doi:10.1145/3357713.3384288.

52 Allen Van Gelder. Contributions to the theory of practical quantified Boolean formula solving. In *Proc. Principles and Practice of Constraint Programming (CP)*, pages 647–663, 2012. doi:10.1007/978-3-642-33558-7_47.

53 Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified Boolean satisfiability solver. In *Proc. IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 442–449, 2002. doi:10.1145/774572.774637.

An Algebraic Approach to MaxCSP

Ilario Bonacina
UPC Universitat Politècnica de Catalunya, Barcelona, Spain

Jordi Levy
IIIA, CSIC, Bellatera, Spain

Abstract

Recently, there have been some attempts to base SAT and MaxSAT solvers on calculi beyond Resolution, even trying to solve SAT using MaxSAT proof systems. One of these directions was to perform MaxSAT sound inferences using polynomials over finite fields, extending the proof system Polynomial Calculus, which is known to be more powerful than Resolution.

In this work, we extend the use of the Polynomial Calculus for optimization, showing its completeness over many-valued variables. This allows using a more direct and efficient encoding of CSP problems (e.g., k-Coloring) and solving the maximization version of the problem on such encoding (e.g., Max-k-Coloring).

2012 ACM Subject Classification Theory of computation → Proof complexity; Computing methodologies → Algebraic algorithms; Mathematics of computing → Solvers

Keywords and phrases MaxCSP, Polynomial Calculus, MaxSAT

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.6

Funding This work was supported by the grant numbers PID2022-138506NB-C21, and PID2022-138506NB-C22 funded by AEI.

1 Introduction

The *Constraint Satisfaction Problem* (CSP) is the problem of, given a set of constraints on variables, finding an assignment satisfying all of them. SAT can be seen as the restriction of this problem to Boolean, *i.e.,* 0/1-valued, variables and constraints given as clauses. MaxCSP and MaxSAT are their respective optimization versions, where we find assignments maximizing the number of satisfied restrictions. CSP could be tackled via translations to SAT, see for instance, [21]. MaxCSP could be tackled via translations to MaxSAT, but this kind of reduction is more subtle since we have to preserve not only satisfiability, but also the number of violated constraints. As a result, in some problems, it may be more efficient to try to solve the original problem using CSP techniques directly, instead of these translations.

In this paper, we consider the natural adaptation of the notions of MaxCSP and MaxSAT to sets of polynomials and we show how to use an algebraic framework to tackle optimization problems on sets of polynomials over *multi-valued* variables. Sets of polynomials over many-valued variables can be used to encode CSP and MaxCSP more directly, avoiding the translation to SAT.

Given a set of polynomials P, we want to determine the maximum number of polynomials in P that can be simultaneously evaluated to zero under a common assignment of the variables. Unlike SAT and MaxSAT that rely on Boolean encodings, using sets of polynomials allows to use a direct multi-valued encoding of MaxCSP problems that are more expressive, concise, and closer in nature to possible underlying algebraic properties of the original problem at hand.

The algebraic framework we use is an adaptation of *Polynomial Calculus* (PC) introduced in [9]. PC is the proof system underlying the reasoning power of algorithms to compute Gröbner bases [7, 6, 12], and, from a theoretical point of view, PC is well-studied in connection


© Ilario Bonacina and Jordi Levy;
licensed under Creative Commons License CC-BY 4.0

with big open questions in Complexity Theory. From a practical point of view, the interest in PC stems from the fact that it is stronger and more versatile than Resolution, the proof system underlying the Conflict-Driven-Clause-Learning paradigm of state-of-the-art SAT-solvers [20, 1]. Moreover, algorithms computing Gröbner bases are useful in practice in the context of *coloring of graphs* [18, 10, 11] and *verification of multiplier circuits* [16, 14, 15, 13]. The previous applications use polynomials over *multi-valued* or *Fourier* encoding, *i.e.,* using variables such that $x^\kappa = 1$. Unlike the case of PC over *Boolean* variables, PC over multi-valued variables is not so well understood. Some well-known principles, such as Tseitin formulas have short (polynomial size) proofs in PC over ± 1-variables, while they do not have short proofs over PC with Boolean variables. Recently, it has also been shown that there are unsatisfiable formulas having polynomial size refutation in PC over Boolean variables, while requiring exponential size in PC over ± 1-valued [19].

Recently, Bonacina, Bonet and Levy [2, 3] introduced an algebraic calculus (wPC_κ) for certifying the optimum value in optimization problems on polynomials over the finite field $\mathbb{F}_\kappa$. This is a generalization of PC to the context of optimization and not only satisfiability. The authors proved the completeness of the calculus in the case of polynomials *over Boolean variables*.

In this work, we consider polynomials over non-Boolean encodings and therefore we give a theoretical ground for extending the applicability of the algorithms using algebraic reasoning and many-valued variables from the context of CSPs to MaxCSPs. For the sake of clarity, we omit the possible generalization to the case of weights and the presence of hard constraints, for instance coming from a Fourier encoding (i.e. using polynomials $x^\ell - 1$ to limit the possible values taken), although all the content of this paper will adapt easily to those cases.

The main contribution of this work is the completeness of the calculus wPC_κ over arbitrary sets of polynomials in $\mathbb{F}_\kappa$ (Theorem 5.1). This argument loosely follows the structure of the completeness of wPC_κ over polynomials forced to take Boolean values in [3] but requires several non-trivial adaptations. This gives an algorithmic procedure that is efficient in some cases, see Section 6.

As a proof of concept, we also show how to apply the calculus wPC_3 to certify Max-3-colorability of graphs exploiting the use of non-Boolean variables (see Section 4).

Structure of the Paper

Section 2 contains all the necessary preliminaries. Section 3 gives the definition of the algebraic framework for optimization and the definition of wPC_κ. Section 4 exemplifies the algebraic framework to Max-coloring. Section 5 contains the completeness of the calculus wPC_κ (Theorem 5.1). This is the main technical contribution of this work. Section 6 gives an application of the algorithmic procedure giving the completeness of wPC_κ. Section 7 give respectively some technical and more general final remarks.

2 Preliminaries

For a natural number $n \in \mathbb{N}$, let $[n] = \{1, \ldots, n\}$. Capital letters typically denote sets and multi-sets, for instance, X usually denotes a set of variables. With p, q, f we typically denote polynomials. κ will always denote the number of elements of a generic finite field.

Basic Algebra Notations

Informally, a *field* $\mathbb{F}$ is a set equipped with addition and multiplication that behave similarly as they would do in the rationals or real numbers.

For instance, $\mathbb{F}_3$ is the finite field with 3 elements $\{0, 1, -1\}$. The addition is modulo 3, that is $1 + 1 = -1$ and $-1 - 1 = 1$, the multiplication is the usual one, and for each $a \in \mathbb{F}_3$, $a^3 = a$. For κ a power of a prime, $\mathbb{F}_\kappa$ is the (unique up-to-isomorphism) finite field with κ elements. For every element $a \in \mathbb{F}_\kappa$ it holds that $a^\kappa = a$.

Given a set of variables X and a field $\mathbb{F}$, $\mathbb{F}[X]$ is the ring of polynomials with coefficients in $\mathbb{F}$ and variables in X.

A *assignment* on the variables X is a map $\alpha : X \to \mathbb{F}$. The *evaluation* of a polynomial $p \in \mathbb{F}[X]$ on α is $p(\alpha)$ and it is the element of $\mathbb{F}$ given by substituting each variable x in p with $\alpha(x)$ and then simplifying the resulting expression using the sum and multiplication rules of $\mathbb{F}$. An assignment α *satisfies* a polynomial p if p vanishes on α, i.e., $p(\alpha) = 0$.

MaxSAT on Polynomials

Similarly to the usual MaxSAT, in the optimization problem on polynomials, we are interested in *maximizing* the number of polynomials set to zero on a common assignment. For a multi-set of polynomials $P \subseteq \mathbb{F}[X]$ and an assignment α let

$$\mathrm{cost}(P, \alpha) = |\{p \in P \ : \ p(\alpha) \neq 0\}| \ ,$$

i.e., $\mathrm{cost}(P, \alpha)$ is the number of polynomials in P non vanishing in α (the polynomials are counted with the respective multiplicity in P). The optimization problem on polynomials becomes *minimizing* the quantity $\mathrm{cost}(P, \alpha)$ over all possible assignments α:

$$\mathrm{cost}(P) = \min_\alpha \mathrm{cost}(P, \alpha) \ .$$

In this work we are interested in formal systems that can be used to certify lower bounds on $\mathrm{cost}(P)$, *i.e.,* $\mathrm{cost}(P) \geqslant s$ for some natural number s. To do so we use the notion of *strongly sound* inference rules introduced in the context of MaxSAT Resolution in [4, 5], adapted here from the context of clauses to generic polynomials.

▶ **Definition 2.1** (strongly sound rule). *Let $p_1, \ldots, p_m, q_1, \ldots, q_\ell$ be polynomials in $\mathbb{F}[X]$ and $\dfrac{p_1, \ldots, p_m}{q_1, \ldots, q_\ell}$ be an inference rule. The rule is* strongly sound *if for every assignment $\alpha : X \to \mathbb{F}$,*

$$\mathrm{cost}(\{p_1, \ldots, p_m\}, \alpha) = \mathrm{cost}(\{q_1, \ldots, q_\ell\}, \alpha) \ .$$

The interest in strongly sound rules lies in the fact they can be used to build inference systems for MaxSAT or, in our case, inference systems for optimization on polynomials. This is based on the fact that a strongly sound rule $\dfrac{p_1, \ldots, p_m}{q_1, \ldots, q_\ell}$ on a multi-set of polynomials P with $p_1, \ldots, p_m \in P$ can be used as a *substitution rule* giving the multi-set

$$P' = P \setminus \{p_1, \ldots, p_m\} \cup \{q_1, \ldots, q_\ell\} \ ,$$

where $\mathrm{cost}(P) = \mathrm{cost}(P')$. Hopefully, P' is simpler, and by repeating the process we would like to arrive to a P' which trivially certifies that $\mathrm{cost}(P') \geqslant s$, for instance, since there are s-many constant non-zero polynomials in P'. To certify that $\mathrm{cost}(P) = s$ we additionally need to prove that the remaining non-constant polynomials in P' have a common zero.

Examples of such systems are wPC_3 (Definition 3.1) or, working with clauses instead of polynomials, MaxSAT-Resolution [17, 4, 5].

In this work, we focus on polynomials in $\mathbb{F}[X]$ and actually we are only interested in how they behave *after* being evaluated, therefore it makes sense to consider them modulo the following equivalence relation.

▶ **Definition 2.2** ($\equiv$)**.** *Given $p, q \in \mathbb{F}[X]$, $p \equiv q$ iff for every assignment $\alpha : X \to \mathbb{F}$, $p(\alpha) = q(\alpha)$.*

For every polynomial $p \in \mathbb{F}_\kappa[X]$, it holds that $p^\kappa \equiv p$ and, if $p_1 \equiv q_1$ and $p_2 \equiv q_2$, then $p_1 p_2 \equiv q_1 q_2$. This gives the following strongly sound substitution rule:

$$\frac{p}{q} \quad (\text{EQUIVALENCE RULE}, \ \equiv) \ , \tag{1}$$

where $p \equiv q$. By the Nullstellensatz, it is possible to check efficiently whether $p \equiv q$ since this is equivalent to the polynomial $p - q$ being in the ideal generated by $\{x^\kappa - x \ : x \in X\}$. This latter condition is efficiently checkable "multilinearizing" the polynomial $p - q$, that is substituting each occurrence of variables $x^{a(\kappa-1)+b}$ with x^b, where $b \in \{0, \ldots, \kappa - 1\}$, and then checking that after this process all the terms cancel-out and the polynomial is identically 0.

3 Polynomial Calculus for MaxSAT

Bonacina, Bonet and Levy in [3] introduced *Polynomial Calculus for* MaxSAT (wPC_κ) as a way to use a system stronger than MaxSAT-Resolution to solve MaxSAT. For simplicity, we omit the treatment of weighted polynomials and sets of polynomials as "hard" constraints. We also omit the case of negative weights and therefore what we call wPC_κ is $\mathsf{wPC}_{\mathbb{N},\mathbb{F}_\kappa}$ in [3].

▶ **Definition 3.1** (wPC_κ, [3])**.** *Given a multi-set of polynomials $P \subset \mathbb{F}_\kappa[X]$, a wPC_κ derivation of a polynomial $p \in \mathbb{F}_\kappa[X]$ is a sequence of multi-sets $L_0, \ldots, L_t$ such that*

1. *$L_0 = P$, $p \in L_t$, and*
2. *for each $i > 0$, L_i is the result of the application of a SUM / PROD / $\equiv$ rule as a substitution rule on L_{i-1}, where the SUM and PROD rules are the following:*

$$\frac{p \quad q}{p+q \quad pq \quad p((p+q)^{\kappa-1} - 1)} \ (\textit{SUM}) \qquad \frac{p}{pq \quad p(q^{\kappa-1} - 1)} \ (\textit{PROD}) \tag{2}$$

The size *of a wPC_κ derivation is the total number of symbols in it.*

The system wPC_κ can be used to certify that for a multi-set of polynomials $P \subset \mathbb{F}_\kappa[X]$, $\text{cost}(P) \geqslant s$. Indeed, a wPC_κ derivation from P of s copies of the constant polynomial 1 implies that $\text{cost}(P) \geqslant s$. Or, equivalently, a wPC_κ derivation of s non-zero constant polynomials, since by the PROD rule from any non-zero constant we can derive the polynomial 1 multiplying by its inverse (for an example of application, see last inference step in Example 3.3). In wPC_κ it is also possible to certify that $\text{cost}(P) = s$: if on top of the s-many polynomials identically 1 we give an assignment which is a common zero of the polynomials obtained alongside the 1s.

To clarify the rules for wPC_κ, we give an equivalent, but more explicit, set of rules for $\kappa = 3$:

$$\frac{p \quad q}{p+q \quad pq \quad pq(q-p)} \ (\text{SUM}) \qquad \frac{p}{pq \quad p(q^2-1)} \ (\text{PROD}) \tag{3}$$

where $p, q \in \mathbb{F}_3[X]$. The rule PROD in (3) is strongly sound essentially because for any assignment α exactly one among $q(\alpha)$ and $q(\alpha)^2 - 1$ is zero. The rule SUM in (3) is strongly sound by case analysis: the less trivial case to check is when we have an assignment α such that $\mathrm{cost}(\{p, q\}, \alpha) = 2$, i.e. $p(\alpha) \in \{1, -1\}$ and $q(\alpha) \in \{1, -1\}$. In this case, the cost of the conclusions of the rule is also 2 because $p(\alpha)q(\alpha) \neq 0$ and, over the assignments αs we are considering, exactly one among $p(\alpha) + q(\alpha)$ and $q(\alpha) - p(\alpha)$ is zero.

▶ Remark 3.2 (On the coefficients of the polynomials). Polynomial Calculus on polynomials with coefficients in a generic ring (for instance $\mathbb{Z}$) was also investigated in [8]. Both the article [3] and ours only consider the MaxSAT/MaxCSP adaptation of Polynomial Calculus over finite fields. The reason is that the rules SUM and PROD do not immediately adapt to arbitrary finite rings such as $\mathbb{Z}/_{4\mathbb{Z}}$ (the set of integers with arithmetic modulo 4) or fields of characteristic 0. In particular, the SUM and PROD rules on polynomials with coefficients in $\mathbb{Z}/_{4\mathbb{Z}}$ are *not* strongly sound. This seems to be related to the presence of 0-divisors, e.g. $2 \cdot 2 = 0$ in $\mathbb{Z}/_{4\mathbb{Z}}$. Further investigation is needed to define a suitable version of Polynomial Calculus for MaxSAT or MaxCSP over infinite fields or generic rings.

To illustrate the calculus wPC_3 and the use of the rules from (3) and (1) we give a couple of examples.

▶ **Example 3.3.** Consider the set of polynomials $P = \{x,\ -y,\ -x + y - 1\} \subset \mathbb{F}_3[x, y]$. The following wPC_3 derivation shows that $\mathrm{cost}(P) = 1$ (we use the $\equiv$-rule implicitly):

$$\dfrac{\dfrac{\dfrac{x \quad -y \quad -x+y-1}{-y \quad y-1 \quad -x^2+xy-x \quad xy^2+xy}\,\text{SUM}}{-1 \quad -y^2+y \quad 0 \quad -x^2+xy-x \quad xy^2+xy}\,\text{SUM}}{1 \quad 0 \quad -y^2+y \quad 0 \quad -x^2+xy-x \quad xy^2+xy}\,\text{PROD}$$

The first SUM is between x and $-x + y - 1$. The second SUM is between $-y$ and $y - 1$. The polynomial $xy^2 + xy$ in the first SUM comes from the fact that

$$x \cdot (-x + y - 1) \cdot ((-x + y - 1) - x) \equiv xy^2 + xy\ .$$

The 0 in the second SUM comes from the fact that $(-y) \cdot (y - 1) \cdot ((y - 1) - (-y)) \equiv 0$.

This shows that $\mathrm{cost}(P) \geqslant 1$. To conclude that $\mathrm{cost}(P) = 1$, it is enough to show that the assignment $x = 0$, $y = 0$ is a zero of all polynomials of the next to last multi-set except the polynomial -1. The last step to transform this non-zero constant polynomial into 1 may be omitted. ⌟

▶ **Example 3.4.** Consider $P = \{x^2+1\} \subset \mathbb{F}_3[x]$. Since $(x^2+1)^2 = x^4+2x^2+1 \equiv 3x^2+1 \equiv 1$, the following wPC_3 derivation shows that $\mathrm{cost}(P) = 1$:

$$\dfrac{\dfrac{x^2+1}{(x^2+1)\cdot(x^2+1) \quad (x^2+1)\cdot((x^2+1)^2-1)}\,\text{PROD}}{1 \quad 0}\,\equiv$$

That is, in particular, the polynomial x^2+1 doesn't have roots in $\mathbb{F}_3$ (as expected). Proposition 5.5 generalizes this example. ⌟

▶ **Example 3.5.** Consider $P = \{-\prod_{i=1}^{n} x_i,\ \prod_{i=1}^{n} x_i - 1\} \subset \mathbb{F}_3[x_1, \ldots, x_n]$. Clearly, P is unsatisfiable and a single application of the SUM rule gives immediately the polynomial 1. On the other hand, encoding the multi-valued variables x_i with Boolean indicator variables will result in polynomials with an exponential (in n) number of terms. For instance, when we encode the value of x_i as $y_{i,0} + y_{i,1} - y_{i,-1}$, and we add the additional polynomial constraints $y_{i,0} + y_{i,1} + y_{i,-1} - 1$ and $y_{i,j}^2 - y_{i,j}$, the Boolean analogue of the set P becomes $\{-\prod_{i=1}^{n}(y_{i,0} + y_{i,1} - y_{i,-1}),\ \prod_{i=1}^{n}(y_{i,0} + y_{i,1} - y_{i,-1}) - 1\}$.

Bonacina, Bonet and Levy in [3] proved that the rules in (2) are strongly sound. This is just a case analysis on the values of $p(\alpha)$, $q(\alpha)$ on an assignment α and uses the fact that $p(\alpha) + q(\alpha) \neq 0$ if and only if $(p(\alpha) + q(\alpha))^{\kappa-1} = 1$. As a consequence, the system wPC_κ is *sound*.

▶ **Theorem 3.6** (soundness [3])**.** *Let $P \subseteq \mathbb{F}_\kappa[X]$ be a multi-set of polynomials. If there exists a wPC_κ derivation of s copies of the polynomial 1 from P then $\mathrm{cost}(P) \geqslant s$.*

A converse of Theorem 3.6 (i.e., that the calculus wPC_κ is also complete) is also true, *under the additional assumption* that the initial polynomials P are on Boolean 0/1-valued variables [3].

In Section 5 (Theorem 5.1), we complete and extend the completeness result in [3] showing a converse of Theorem 3.6 for an arbitrary κ *without any additional assumption*. Therefore allowing the use of wPC_κ beyond multi-sets of polynomials over Boolean variables. As Example 3.5 shows, using Boolean or multi-valued variables might result in completely different complexities of the problem at hand.

4 A possible application: Max-3-Coloring

A calculus for optimization on polynomials on arbitrary variables allows using direct encoding of MaxCSPs. Moreover, the algebraic language offers the possibility to concisely encode many natural constraints used in CSP.

One notable example is the Vertex Coloring Problem (VCP), which we analyze in detail in this section. Given an undirected graph $G = (V, E)$, the *Vertex Coloring Problem* (VCP) is the problem of finding a labeling of the vertices of G such that no vertices along an edge share the same label. Deciding if a given graph is 3-colorable, i.e. colorable with a set of 3 labels, is one of the classical NP-complete problems.

Max-3-Coloring, the problem of maximizing the number of edges that connect 3-colored vertexes with distinct labels, is a more difficult problem. Max-2-Coloring is exactly MaxCUT, which is NP-complete, whereas 2-Coloring is in P.

To show that a given graph G is not say 3-colorable we can associate to the VCP on G a CNF formula which is satisfiable iff G is 3-colorable. Alternatively, we can associate the VCP on G to a set of polynomials with a common solution iff G is 3-colorable. To encode the VCP algebraically, we need to express restrictions of the form $x \neq y$. There are several ways of doing this. Using polynomials with coefficients in $\mathbb{F}_3$, it can be encoded as $x^2 + xy + y^2 - 1$. This polynomial evaluates to zero when $x \neq y$.

This immediately generalizes to expressing the *nogood* $\langle v_1, \ldots, v_\ell \rangle$ for variables $\langle x_1, \ldots, x_\ell \rangle$, i.e. the fact that at least for one index i, $x_i \neq v_i$. This can be done using the polynomial $\prod_{i=1}^{\ell}(x_i^2 + v_i\, x_i + v_i^2 - 1)$ which evaluates to zero exactly when there is an index i such that $x_i \neq v_i$.

Alternatively, for instance, the constraint $x \neq y$ for 3-coloring can be encoded using Fourier variables in $\mathbb{F}_4$ as $\{x^3-1, y^3-1, x^2+xy+y^2-1\}$, where the first two polynomials are hard constraints and the third one is a soft constraint. Both the encoding in $\mathbb{F}_3$ and the encoding in $\mathbb{F}_4$ with Fourier variables generalize easily to ℓ-coloring. In this work, for simplicity, we only consider the encoding with variables in $\mathbb{F}_3$.

▶ **Definition 4.1** ($\mathsf{VC}_3(G)$)**.** *Given an undirected graph $G = (V, E)$, consider one variable x_v, for each $v \in V$, and the set of polynomials $\mathsf{VC}_3(G) \subset \mathbb{F}_3[\{x_v \ : \ v \in V\}]$ given by*

$$\mathsf{VC}_3(G) = \{x_v^2 + x_v x_w + x_w^2 - 1 \ : \ \{v, w\} \in E\} \ .$$

▶ **Proposition 4.2.** *The polynomials in $\mathsf{VC}_3(G)$ have a common zero iff G is 3-colorable. Moreover, $\mathrm{cost}(\mathsf{VC}_3(G))$ equals the minimum number of edges connecting two vertexes with the same color in an optimal coloring.*

Proof. It is immediate to see that $x_v^2 + x_v x_w + x_w^2 - 1 = 0$ if and only if $x_v \neq x_w$. This could be done either by analyzing all cases or, alternatively, noticing that if $x_v \neq x_w$, then

$$0 = \frac{x_v - x_w}{x_v - x_w} - 1 = \frac{x_v^3 - x_w^3}{x_v - x_w} - 1 = x_v^2 + x_v x_w + x_w^2 - 1 \ .$$

If $x_v = x_w$, then $x_v^2 + x_v x_w + x_w^2 - 1 = 3x_v^2 - 1 = -1$. ◀

In the following, as an example, we consider the 3-colorability of the complete graph $G = K_4$, *i.e.,* the graph with 4 vertexes x, y, z, t, and 6 edges connecting all pairs of vertexes. For $v, w \in \{x, y, z, t\}$, let $p_{vw} = v^2 + v\,w + w^2 - 1$, that is

$$\mathsf{VC}_3(K_4) = \{p_{xy}, p_{xz}, p_{xt}, p_{yz}, p_{yt}, p_{zt}\} \ . \tag{4}$$

In all optimal 3-colorings of K_4, one edge connects two vertexes with the same color, i.e. we must have that $\mathrm{cost}(\mathsf{VC}_3(K_4)) = 1$.

In Fig. 1 we show a wPC$_3$ derivation certifying that $\mathrm{cost}(\mathsf{VC}_3(K_4)) \geqslant 1$. An alternative derivation is in Fig. 2.

Every time that a polynomial is used in one inference, it is removed and cannot be used in other inferences. Therefore, every polynomial has a unique outgoing arrow.

In the wPC$_3$ derivation in Fig. 1, we obtain the polynomial s'_7 which is

$$\begin{aligned} s'_7 &= s'_1 + s'_2 + s'_3 + o'_{4,2} + o'_{5,2} \\ &= x^2y^2 - x^2yz - xy^2z - xyz^2 + y^2z^2 - x^2yt - xy^2t \\ &\quad + x^2zt - y^2zt + xz^2t - yz^2t - xyt^2 + y^2t^2 + xzt^2 \\ &\quad - yzt^2 + x^2 + xy + xz + yz + z^2 + xt + yt \\ &\quad + zt + t^2 + 1 \ . \end{aligned}$$

To complete the proof and certify that $\mathrm{cost}(\mathsf{VC}_3(K_4)) \geqslant 1$, we need to prove that s'_7 has no roots. This can be proved efficiently in wPC$_3$ following the general structure of the wPC$_3$ derivation in Proposition 5.5.

To prove that $\mathrm{cost}(\mathsf{VC}_3(K_4)) = 1$, we need to show that there exists an assignment that simultaneously evaluates to zero all the other polynomials. Obviously, this is not trivial.

An alternative wPC$_3$ derivation for the set of polynomials in (4) is in Fig. 2. In this case, we obtain the polynomial s_5, which directly proves $\mathrm{cost}(\mathsf{VC}_3(K_4)) \geqslant 1$ since $s_5 \equiv -1$. In this second proof, we use the $R_x\,(p)$ operator defined in (5) and the saturation strategy from the completeness theorem.

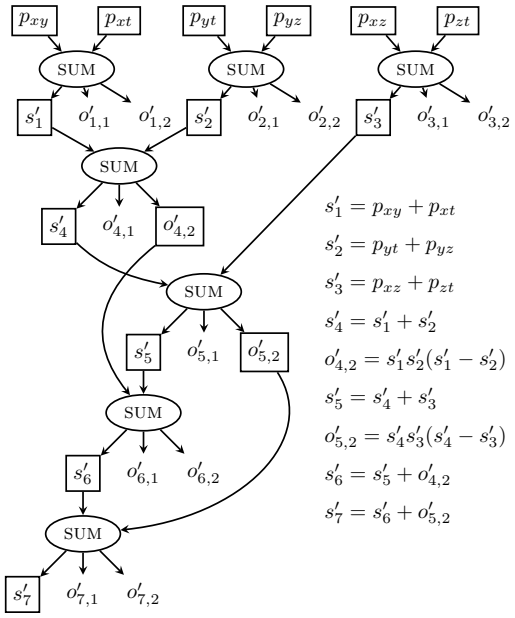


Figure 1 A proof of the non-3-colorability of K_4.

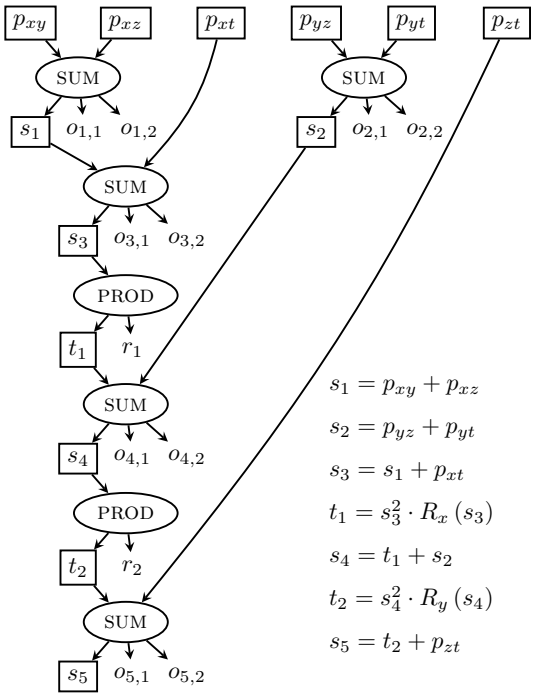


Figure 2 Alternative proof of the non-3-colorability of K_4.

In the following section, we prove a completeness result ensuring that, instead of just an unsolvable polynomial, we can get $\mathrm{cost}(\mathsf{VC}_3(K_n))$ copies of the polynomial 1 and a set of simultaneously solvable polynomials.

5 Completeness

In this section, we prove the converse of Theorem 3.6.

▶ **Theorem 5.1** (completeness). *Let $P \subset \mathbb{F}_\kappa[X]$ be a multi-set of polynomials. If $\mathrm{cost}(P) = s$, then there exists a wPC_κ derivation of a multi-set containing s copies of the polynomial 1 from P.*

The proof of Theorem 5.1 follows the structure of [3, Theorem 4.1] which was ultimately inspired by [5]. Informally, the argument goes by the following algorithmic procedure. Take an ordering on the variables $X = \{x_1, \ldots, x_n\}$, then on the polynomials *depending* on x_1 try to infer as many non-trivial polynomials without x_1 as possible using the SUM and PROD rules. This would be, intuitively, the notion of *saturation* (Definition 5.6). Then repeat the process one variable at a time. At the end of the process, we will have obtained s-many copies of the polynomial 1 together with several other polynomials. The process will have preserved the cost along the way, while the remaining polynomials are satisfiable, by the structural properties of the saturation. To make this very high-level structure of the argument formal, first, we need to formalize the notion of when a polynomial *depends* on a variable x.

▶ **Definition 5.2** (dependence). *A polynomial $p \in \mathbb{F}_\kappa[X]$* does not depend *on a variable $x \in X$ if there exists a polynomial $q \in \mathbb{F}_\kappa[X]$ not containing x such that $p \equiv q$.*

For instance, the polynomial $x^3 - x - 1 \in \mathbb{F}_3[x]$ does not depend on x, since $x^3 - x - 1 \equiv -1$ over $\mathbb{F}_3$.

The following proposition gives an equivalent characterization for when a polynomial does not depend on a variable.

For $p \in \mathbb{F}_\kappa[X]$, we use the notation $p_{x \mapsto a}$, where $x \in X$, to denote the polynomial p where each occurrence of x has been substituted with $a \in \mathbb{F}_\kappa$, and we use the notation

$$R_x(p) = \prod_{a \in \mathbb{F}_\kappa} p_{x \mapsto a} \ .$$

In particular, for $p \in \mathbb{F}_3[X]$ and $x \in X$,

$$R_x(p) = p_{x \mapsto 0} \cdot p_{x \mapsto 1} \cdot p_{x \mapsto -1} \ . \tag{5}$$

▶ **Proposition 5.3.** *Let $p \in \mathbb{F}_\kappa[X]$ be a polynomial and $x \in X$ a variable. Then the following are equivalent*

1. *p does not depend on x,*
2. *$p \equiv R_x(p)$.*

Proof. Clearly item 2 implies item 1, since p is equivalent to the polynomial $R_x(p)$ that does not contain x.

Conversely, if p does not depend on x, then there exists q not containing the variable x such that $p \equiv q$. Then, restricting by $x \mapsto a$ for all $a \in \mathbb{F}_\kappa$, we get $p_{x \mapsto a} \equiv q_{x \mapsto a} \equiv q$. Which gives $p \equiv p_{x \mapsto a}$ for each $a \in \mathbb{F}_\kappa$. Hence

$$R_x(p) = \prod_{a \in \mathbb{F}_\kappa} p_{x \mapsto a} \equiv p^\kappa \equiv p \ . \qquad \blacktriangleleft$$

The polynomials $R_x(p)$ play a central role in our argument for Theorem 5.1. In particular, it is possible to derive $R_x(p)$ from p using the rules of wPC_3.

▶ **Lemma 5.4.** *From any polynomial $p \in \mathbb{F}_\kappa[X]$, it is possible to derive $R_x(p)$ using* PROD *and the equivalence rule.*

Proof. The derivation is the following:

$$\dfrac{\dfrac{p}{p^{\kappa-1} \cdot R_x(p) \quad p\big((p^{\kappa-2} \cdot R_x(p))^{\kappa-1} - 1\big)}\ \text{PROD}}{R_x(p) \qquad p\big((p^{\kappa-2} \cdot R_x(p))^{\kappa-1} - 1\big)}\ \equiv \tag{6}$$

In the last equivalence, we used the fact that $p^{\kappa-1} \cdot R_x(p) \equiv R_x(p)$. To see this, we just need to show that for every assignment $\alpha : X \to \mathbb{F}_\kappa$,

$$p^{\kappa-1}(\alpha) \cdot R_x(p)(\alpha) \equiv R_x(p)(\alpha)\ . \tag{7}$$

Let $b = \alpha(x)$. Then, the equivalence in (7) follows from the following equalities:

$$\begin{aligned} p^{\kappa-1}(\alpha) \cdot R_x(p)(\alpha) &= p^{\kappa-1}(\alpha) \cdot \prod_{a \in \mathbb{F}_\kappa} p_{x \mapsto a}(\alpha) \\ &= p^{\kappa}(\alpha) \cdot \prod_{a \in \mathbb{F}_\kappa \setminus \{b\}} p_{x \mapsto a}(\alpha) \\ &\equiv p(\alpha) \cdot \prod_{a \in \mathbb{F}_\kappa \setminus \{b\}} p_{x \mapsto a}(\alpha) \\ &= R_x(p)(\alpha)\ . \end{aligned}$$

◀

Towards proving Theorem 5.1, first consider a generalization of Example 3.4.

▶ **Proposition 5.5.** *If $p \in \mathbb{F}_\kappa[X]$ is a polynomial that never vanishes, i.e.* $\mathrm{cost}(\{p\}) = 1$*, then there is a* wPC_κ *derivation of* 1 *from* $\{p\}$*.*

Proof. Let the variables in X be $x_1, \ldots, x_n$. A possible derivation is to consider $p_0 = p$ and, for each $i = 0, \ldots, n-1$, consider

$$p_{i+1} = R_{x_{i+1}}(p_i)\ . \tag{8}$$

By induction on i, we prove that p_i never vanishes and p_i does not depend on $x_1, \ldots, x_i$. By assumption, this holds for $i = 0$. Suppose then p_i never vanishes and p_i does not depend on $x_1, \ldots, x_i$. We have two cases.

If p_i does not depend on x_{i+1}, then, by Proposition 5.3, $p_{i+1} \equiv p_i$ and the inductive hypothesis is implied.

If p_i does depend on x_{i+1}, then p_{i+1}, by construction does not depend on x_{i+1}. We need to show that p_{i+1} never vanishes. Suppose towards a contradiction it vanishes in α. Evaluating (8) in α we get:

$$0 = p_{i+1}(\alpha) = R_{x_{i+1}}(p_i)(\alpha) = \prod_{a \in \mathbb{F}_\kappa} p_{i, x_{i+1} \mapsto a}(\alpha) = \prod_{a \in \mathbb{F}_\kappa} p_i(\alpha_a)$$

where α_a is the assignment α modified such that $x_{i+1} \mapsto a$. A contradiction immediately arises from the fact that p_i never vanishes, while the previous chain of equalities implies that $p_i(\alpha_a) = 0$ for some $a \in \mathbb{F}_\kappa$.

The sequence of polynomials $p_0, \ldots, p_n$ gives a backbone of a wPC_κ derivation: indeed, by Lemma 5.4, each p_i is derivable from p_{i-1} and p_n is equivalent to 1. ◀

The argument in Proposition 5.5, give rise to the notion of *saturated* set of polynomials, and the extension of the previous argument to saturated sets is Lemma 5.9, the main lemma needed to prove Theorem 5.1.

▶ **Definition 5.6** (x-saturated). *A set of polynomials $S \subseteq \mathbb{F}_\kappa[X]$ is* x-saturated *if for all ℓ-tuple $p_1, \ldots, p_\ell \in S$ depending on x and for all ℓ-tuple $a_1, \ldots, a_\ell \in \mathbb{F}_\kappa$,*

$$R_x\left(\sum_{i=1}^{\ell} a_i p_i\right) \equiv 0 \ .$$

Some combinations of polynomials in the definition above are trivially zero or are redundant. For instance, $R_x(p) \equiv 0$ if and only if $R_x(-p) \equiv 0$.

The notion of x-saturated set of polynomials (Definition 5.6) is more general than what is actually needed to prove Lemma 5.9 and therefore Theorem 5.1. It is enough to check that

$$\begin{aligned} R_x(p_1) &\equiv 0 \\ R_x(\omega p_1 + p_2) &\equiv 0 \\ R_x(\omega p_1 + \omega p_2 + p_3) &\equiv 0 \\ &\vdots \\ R_x(\omega p_1 + \cdots + \omega p_{\kappa-2} + p_{\kappa-1}) &\equiv 0 \\ R_x(p_1 + \cdots + p_\kappa) &\equiv 0 \end{aligned}$$

for every $\omega \in \mathbb{F}_\kappa \setminus \{0\}$ and polynomials $p_j \in \mathbb{F}_\kappa[X]$ depending on x.

First, we show that for any set of polynomials $P \subset \mathbb{F}_\kappa[X]$ it is possible to derive in wPC_κ an x-saturated set S.

▶ **Lemma 5.7.** *For polynomials $p_1, \ldots, p_\ell \in \mathbb{F}_\kappa[X]$, coefficients $a_1, \ldots, a_\ell \in \mathbb{F}_\kappa$ and $x \in X$, there is a* wPC_κ *derivation from P of a multi-set containing $R_x\left(\sum_{i=1}^{\ell} a_i p_i\right)$.*

Proof. To derive in wPC_κ the polynomial $R_x\left(\sum_{i=1}^{\ell} a_i p_i\right)$ we first use the PROD and SUM rules to derive $\sum_{i=1}^{\ell} a_i p_i$ and then apply the construction in Lemma 5.4 to $\sum_{i=1}^{\ell} a_i p_i$ deriving $R_x\left(\sum_{i=1}^{\ell} a_i p_i\right)$. ◀

▶ **Lemma 5.8.** *For any multi-set of polynomials $P \subseteq \mathbb{F}_\kappa[X]$ and $x \in X$, there is a derivation from P in* wPC_κ *of an x-saturated multi-set of polynomials S.*

The proof of this lemma is similar to analoguos lemmas in [5, 3].

Proof. We start from P and whenever we have polynomials $p_1, \ldots, p_\ell \in P$ depending on x and coefficients $a_1, \ldots, a_\ell \in \mathbb{F}_\kappa$ such that $R_x\left(\sum_{i=1}^{\ell} a_i p_i\right) \not\equiv 0$ we apply the construction from Lemma 5.7 and get a new multi-set of polynomials P' with the property that for every α, $\mathrm{cost}(P, \alpha) = \mathrm{cost}(P', \alpha)$ but in P' we have at least one more polynomial q without x and $q \not\equiv 0$. This polynomial q is $R_x\left(\sum_{i=1}^{\ell} a_i p_i\right)$. For a set of polynomials Q, let $\sigma(Q) = \sum_\alpha \mathrm{cost}(Q_x, \alpha)$, where Q_x is the multi-set of polynomials in Q depending on x. It is always the case that $\sigma(Q) \geqslant 0$.

From the fact that $q \not\equiv 0$, that is $\sigma(\{q\}) > 0$, and the fact that wPC_κ rules preserve the cost, we obtain that $\sigma(P') < \sigma(P)$. We exhaustively apply this construction and in at most $\sigma(P)$ many steps we must reach a set of polynomials S which is x-saturated. ◀

The main property of saturated sets of polynomials is that they behave very nicely w.r.t. assignments. The following is the main technical lemma of the article.

▶ **Lemma 5.9.** *If $S \subset \mathbb{F}_\kappa[X]$ is an x-saturated multi-set of polynomials, then every assignment $\alpha : X \to \mathbb{F}_\kappa$ can be modified in x to an assignment satisfying all the polynomials in S depending on x.*

Proof. Suppose towards a contradiction, there is an assignment α and polynomials f_b for $b \in \mathbb{F}_\kappa$ in S depending on x such that $f_b(\alpha_b) \neq 0$ for every $b \in \mathbb{F}_\kappa$, where α_b is the assignment mapping x to b and every other value as in α. For shortness let $f_{b,j} = f_b(\alpha_j)$, that is, in particular $f_{b,b} \neq 0$. We just need to prove that for each $b \neq b'$, $f_{b,b'} = 0$. This will imply a contradiction in the following way: Since S is saturated,

$$R_x\left(\sum_{b\in\mathbb{F}_\kappa} f_b\right) = \prod_{j\in\mathbb{F}_\kappa} (\sum_{b\in\mathbb{F}_\kappa} f_b)_{x\mapsto j} \equiv 0 \ ,$$

and applying α we get

$$0 \overset{(\star)}{=} R_x\left(\sum_{b\in\mathbb{F}_\kappa} f_b\right)(\alpha) = \prod_{j\in\mathbb{F}_\kappa} (\sum_{b\in\mathbb{F}_\kappa} f_{b,j}) \overset{(\star\star)}{=} \prod_{j\in\mathbb{F}_\kappa} f_{j,j} \neq 0 \ ,$$

since $f_{j,j} \neq 0$ for any $j \in \mathbb{F}_\kappa$. The equality $(\star)$ holds since S is saturated and applying α, while the equality $(\star\star)$ holds if we manage to prove $f_{b,j} = 0$ for any $j \neq b$. The rest of the argument is to prove that for each $b \neq b'$, $f_{b,b'} = 0$. This will be a consequence of S being x-saturated. We consider all the polynomials of the form $\omega f_{b_1} + \omega f_{b_2} + \cdots + \omega f_{b_{t-1}} + f_{b_t}$, for $1 \leqslant t < \kappa$ and $\omega \in \mathbb{F}_\kappa \setminus \{0\}$, and proceed by induction on t.

For $t = 1$, since the saturation process stopped in S we must have that, for every $b \in \mathbb{F}_\kappa$,

$$R_x\left(f_b\right) = \prod_{j\in\mathbb{F}_\kappa} (f_b)_{x\mapsto j} \equiv 0 \ ,$$

and, since $(f_b)_{x\mapsto j}(\alpha) = f_b(\alpha_j) =: f_{b,j}$, we get

$$\prod_{j\in\mathbb{F}_\kappa} f_{b,j} = 0 \ . \tag{9}$$

In turn, this implies

$$\prod_{j\in\mathbb{F}_\kappa\setminus\{b\}} f_{b,j} = 0 \ , \tag{10}$$

since by assumption $f_{b,b} \neq 0$.

For $t = 2$, again, since S is saturated, for any b, b' distinct elements of $\mathbb{F}_\kappa$ and any $\omega \in \mathbb{F}_\kappa$, $\prod_{j\in\mathbb{F}_\kappa} (\omega f_b + f_{b'})_{x\to j} \equiv 0$.

Evaluating in α we get

$$\begin{aligned} 0 &= \prod_{j\in\mathbb{F}_\kappa} (\omega f_b + f_{b'})_{x\mapsto j}(\alpha) \\ &= \prod_{j\in\mathbb{F}_\kappa} (\omega f_b(\alpha_j) + f_{b'}(\alpha_j)) \\ &= \prod_{j\in\mathbb{F}_\kappa} (\omega f_{b,j} + f_{b',j}) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{S \subseteq \mathbb{F}_\kappa} \omega^{|S|} \prod_{j \in S} f_{b,j} \prod_{j \notin S} f_{b',j} \\
&= \sum_{\ell=0}^{\kappa} \omega^\ell \sum_{S \subseteq \binom{\mathbb{F}_\kappa}{\ell}} \prod_{j \in S} f_{b,j} \prod_{j \notin S} f_{b',j} \\
&\overset{(\star)}{=} \sum_{\ell=1}^{\kappa-1} \omega^\ell \sum_{S \subseteq \binom{\mathbb{F}_\kappa}{\ell}} \prod_{j \in S} f_{b,j} \prod_{j \notin S} f_{b',j} \ ,
\end{aligned}
$$

where with $\binom{\mathbb{F}_\kappa}{\ell}$ we denoted the set of all subsets of $\mathbb{F}_\kappa$ of size ℓ. The equality in $(\star)$ is due to (9).

Let $A_\ell = \sum_{S \subseteq \binom{\mathbb{F}_\kappa}{\ell}} \prod_{j \in S} f_{b,j} \prod_{j \notin S} f_{b',j}$. By the previous equality, we get that

$$
\begin{pmatrix}
\omega_1 & \omega_1^2 & \cdots & \omega_1^{\kappa-1} \\
\omega_2 & \omega_2^2 & \cdots & \omega_2^{\kappa-1} \\
\vdots & \vdots & \ddots & \vdots \\
\omega_{\kappa-1} & \omega_{\kappa-1}^2 & \cdots & \omega_{\kappa-1}^{\kappa-1}
\end{pmatrix}
\cdot
\begin{pmatrix}
A_1 \\ A_2 \\ \vdots \\ A_{\kappa-1}
\end{pmatrix}
=
\begin{pmatrix}
0 \\ 0 \\ \vdots \\ 0
\end{pmatrix}
$$

where $\mathbb{F}_\kappa \setminus \{0\} = \{\omega_1, \ldots, \omega_{\kappa-1}\}$. The Vandermonde matrix above is invertible, therefore the unique solution is $A_\ell = 0$ for every $\ell \in \{1, \ldots, \kappa-1\}$.

For any $3 \leqslant t \leqslant \kappa - 1$, in a similar way, starting from

$$
\prod_{j \in \mathbb{F}_\kappa} (\omega f_{b_1} + \omega f_{b_2} + \cdots + \omega f_{b_{t-1}} + f_{b_t})_{x \to j} \equiv 0
$$

we obtain that for every $\ell \in \{1, \kappa - 1\}$

$$
\sum_{\substack{S_1, \ldots, S_t \\ \text{disjoint decomposition of } \mathbb{F}_\kappa \\ |S_1| + \cdots + |S_{t-1}| = \ell}} \prod_{m=1}^{t} \prod_{j \in S_m} f_{b_m,j} = 0 \ , \tag{11}
$$

where $S_1, \ldots, S_t$ being a disjoint decomposition of $\mathbb{F}_\kappa$ means that $\bigcup_{i=1}^t S_i = \mathbb{F}_\kappa$ and the S_is are disjoint, but possibly empty. Since the equality (11) holds for every t and ℓ then, by induction, we also have that the equality in (11) holds for non-empty $S_1, \ldots, S_t$, that is we have

$$
\sum_{\substack{S_1, \ldots, S_t \\ \text{partition of } \mathbb{F}_\kappa \\ |S_1| + \cdots + |S_{t-1}| = \ell}} \prod_{m=1}^{t} \prod_{j \in S_m} f_{b_m,j} = 0 \ , \tag{12}
$$

For $t = 2$ and $\ell = 1$ this means

$$
\begin{aligned}
0 \quad &= \quad \sum_{S \subseteq \binom{\mathbb{F}_\kappa}{1}} \prod_{j \in S} f_{b',j} \prod_{j \notin S} f_{b,j} \\
&= \quad \sum_{r \in \mathbb{F}_\kappa} f_{b',r} \prod_{j \in \mathbb{F}_\kappa \setminus \{r\}} f_{b,j} \\
&\overset{\text{(due to (10))}}{=} \sum_{r \in \mathbb{F}_\kappa \setminus \{b\}} f_{b',r} \prod_{j \in \mathbb{F}_\kappa \setminus \{r\}} f_{b,j} \\
&= \quad f_{b,b} \sum_{r \in \mathbb{F}_\kappa \setminus \{b\}} f_{b',r} \prod_{j \in \mathbb{F}_\kappa \setminus \{r,b\}} f_{b,j} \ .
\end{aligned}
$$

Multiplying by $\prod_{j\in\mathbb{F}_\kappa\setminus\{b,b'\}} f_{b,j}$ and using (10) we get $0 = f_{b,b}f_{b',b'}\prod_{j\in\mathbb{F}_\kappa\setminus\{b,b'\}} f_{b,j}^2$, which implies

$$\prod_{j\in\mathbb{F}_\kappa\setminus\{b,b'\}} f_{b,j} = 0\ . \tag{13}$$

For $t = 3$ the product in (13) is a single term $f_{b,j} = 0$ for $j \neq b$ arbitrary, hence concluding the argument. For an arbitrary t, we use an inductive argument with $t = \ell + 1$ and $\ell = 1, \ldots, \kappa - 2$. For simplicity we show how to go from $\ell = 1$, $t = 2$ to $\ell = 2$, $t = 3$. We have the following

$$\begin{aligned}
0 &= \sum_{\substack{S_1,S_2,S_3\\ \text{partition of } \mathbb{F}_\kappa\\ |S_1|+|S_2|=2}} \prod_{m=1}^{3}\prod_{j\in S_m} f_{b_m,j}\\
&= \sum_{\substack{\ell_1,\ell_2\in\mathbb{F}_\kappa\\ \ell_1\neq\ell_2}} f_{b_1,\ell_1}f_{b_2,\ell_2}\prod_{j\in\mathbb{F}_\kappa\setminus\{\ell_1,\ell_2\}} f_{b_3,j}\\
&\overset{\text{(due to (13))}}{=} \sum_{\substack{\ell_1,\ell_2\in\mathbb{F}_\kappa\setminus\{b_3\}\\ \ell_1\neq\ell_2}} f_{b_1,\ell_1}f_{b_2,\ell_2}\prod_{j\in\mathbb{F}_\kappa\setminus\{\ell_1,\ell_2\}} f_{b_3,j}\\
&= f_{b_3,b_3}\sum_{\substack{\ell_1,\ell_2\in\mathbb{F}_\kappa\setminus\{b_3\}\\ \ell_1\neq\ell_2}} f_{b_1,\ell_1}f_{b_2,\ell_2}\prod_{j\in\mathbb{F}_\kappa\setminus\{\ell_1,\ell_2,b_3\}} f_{b_3,j}\ .
\end{aligned}$$

Multiplying by $\prod_{j\in\mathbb{F}_\kappa\setminus\{b_1,b_2,b_3\}} f_{b_3,j}$ and using (13) we get

$$0 = f_{b_1,b_1}f_{b_2,b_2}f_{b_3,b_3}\prod_{j\in\mathbb{F}_\kappa\setminus\{b_1,b_2,b_3\}} f_{b_3,j}^2\ ,$$

which implies

$$\prod_{j\in\mathbb{F}_\kappa\setminus\{b_1,b_2,b_3\}} f_{b_3,j} = 0\ . \tag{14}$$

Repeating this argument inductively, when $t = \kappa - 1$ and $\ell = \kappa - 2$, we get that the analogue of the product in (14) becomes a single term $f_{b_t,j} = 0$ for $j \neq b_t$. This concludes the argument for an arbitrary κ. ◀

▶ Remark 5.10. The argument for Lemma 5.9 also adapts to sets of polynomials with coefficients in $\mathbb{F}_\kappa$ *if* they are forced to take non-zero values, i.e. we consider the sets of polynomials containing $x^{\kappa-1} - 1$ for each variable x as hard constraints.

Using Lemma 5.8 and Lemma 5.9 it is then immediate to prove Theorem 5.1 using an inductive argument. This is very similar to the argument in [5, Theorem 10] and [3, Lemma 4.5].

Proof of Theorem 5.1. Choose some ordering on the set of variables $X = \{x_1, \ldots, x_n\}$. From the initial set of polynomial P using Lemma 5.8 derive using the wPC_κ rules a multi-set P_1 x_1-saturated. Let R_1 be the part of P_1 not depending on x_1 and S_1 be the part of P_1 depending on x_1. The next step is to saturate R_1 w.r.t. x_2 to get a P_2. As before let S_2 and R_2 be the sets of polynomials in P_2 depending on x_2 and not depening on x_2 (resp.).

Repeat the process on R_2 saturating it w.r.t. x_3. In an inductive fashion keep doing this on all the variables one by one. Notice that that S_i depends on x_i but it does not depend on $x_1, ..., x_{i-1}$.

At the end of the process, we get s-many constant and non-zero polynomials together with the set of polynomials $S_1 \cup S_2 \cup \cdots \cup S_n$. With some trivial application of the PROD rule, we can assume that we have s-many copies of the polynomial 1.

To conclude the argument, since the cost is always preserved along the process, it is enough to show that the remaining polynomials are satisfiable. Informally, this holds because the saturated sets we produce are over fewer and fewer variables and therefore we can use Lemma 5.9 inductively on the saturated sets we produced from the last to the first one. Starting from any assignment, by Lemma 5.9 we can modify it in x_n to satisfy S_n. In general, given an assignment α_i satisfying S_i, by Lemma 5.9, we can modify it in the variable x_{i-1} obtaining an assignment α_{i-1} satisfying S_{i-1}. Since all the S_j for $j \geqslant i$ do not depend on x_{i-1} the assignment α_{i-1} continues to satisfy $S_i \cup \cdots \cup S_n$. ◀

The argument for Theorem 5.1 gives a saturation-based proof-construction strategy. Lemma 5.8 gives a possible way to obtain a saturated set, but any other heuristics to obtain a saturated set would be equally good. We exemplify the saturation strategy in Section 6.

▶ Remark 5.11 (On the completeness). To prove Theorem 5.1 we used very little of the actual form of the SUM and PROD rules. We indeed only used the fact that there are strongly sound inference rules that given as premise a polynomial p allow to infer $p\,q$ (together with other polynomials), for an arbitrary q, and that given as premises p and q allow to infer $p + q$ (together with other polynomials). In other words, Theorem 5.1 will hold for any set of strongly sound rules satisfying the properties above.

6 Another example: Flow in graphs

We exemplify the effectiveness of the saturation process on a simple constraint satisfaction problem. Given a connected graph $G = (V, E)$, an order on its vertices, and a function $b : V \to \mathbb{Z}$, find a flow for every edge such that $b(v)$ is the sum of the flows of all adjacent edges, i.e. find an assignment $f : E \to \mathbb{Z}$ such that for all $v \in V$

$$b(v) = \sum_{\substack{u \in N(v) \\ u < v}} f(\{u, v\}) - \sum_{\substack{u \in N(v) \\ u > v}} f(\{u, v\}) \ . \tag{15}$$

The problem can be solved using linear programming and it has a solution if and only if $\sum_{v \in V} b(v) = 0$. However, we are interested in possible refutations when the problem has no solution. In this case, we can pick any prime κ not dividing $\sum_{v \in V} b(v)$ and refute the set of eq. (15) modulo κ, that is show that the set of polynomials, for $v \in V$,

$$\sum_{\substack{u \in N(v) \\ u < v}} x_{u,v} - \sum_{\substack{u \in N(v) \\ u > v}} x_{v,u} - (b(v) \mod \kappa) \tag{16}$$

has no solution in $\mathbb{F}_\kappa$. We show how the saturation process in wPC$_\kappa$, independently of the order we decide to saturate variables, always easily gives a short refutation of (16).

At the beginning choose arbitrarily a variable to saturate, let it be $x_{u,v}$. There are only two polynomials in (16) depending on $x_{u,v}$, one with positive coefficient 1, one with negative coefficient -1. It is easy to see (see Lemma 6.1 below), that to saturate w.r.t. $x_{u,v}$, the only option is to apply the SUM rule to the two (linear) polynomials containing $x_{u,v}$, and

the obtained set is saturated for $x_{u,v}$. In this set, there is only one polynomial without $x_{u,v}$ which is linear. We keep saturating for all the variables one by one. It is easy to see that at any given moment the set to saturate w.r.t. the current variable x only has linear polynomials and only two of them depending on x, in one x appears with coefficient 1, in the other with coefficient -1. Again it is easy to see that the only option to saturate w.r.t. x is to apply the SUM rule to the two linear polynomials depending on x and the obtained set is saturated. Repeating this process for all variables gives the refutation. Since G is connected this will also show that the set of polynomials in (16) is minimally unsatisfiable.

▶ **Lemma 6.1.** *Let $p = \ell q$ be a polynomial in $\mathbb{F}_\kappa[X]$ with ℓ a linear polynomial depending on a variable x. Then $R_x(p) \equiv 0$.*

Proof. The polynomial p has the form $p = (cx + \ell')q$ with ℓ' linear not containing the variable x and $c \neq 0$. We have that $R_x(p) = \prod_{j \in \mathbb{F}_\kappa} (cj + \ell'|_{x \to j})q|_{x \to j}$. To show that $R_x(p) \equiv 0$ it is enough to show that for every assignment $\alpha : X \to \mathbb{F}_\kappa$, $R_x(p)(\alpha) = 0$:

$$R_x(p)(\alpha) = \prod_{j \in \mathbb{F}_\kappa} (cj + \ell'|_{x \to j}(\alpha))q|_{x \to j}(\alpha) = \prod_{j \in \mathbb{F}_\kappa} (cj + \ell'(\alpha))q|_{x \to j}(\alpha)$$

which is set to 0 by the factor corresponding to $j = -\frac{\ell'(\alpha)}{c}$. ◀

7 Conclusions

We described a calculus for MaxCSP based on polynomials and variables ranging over finite fields $\mathbb{F}_\kappa$. This provided a theoretical ground for further investigations of the efficiency of algebraic methods in the context of MaxCSPs.

The calculus is proved complete, and from this proof, we derived a *saturation*-based proof-construction strategy. We exemplified the constructions on a coloring principle, the Max-3-Coloring Problem and on a Flow principle.

References

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

2 Ilario Bonacina, Maria Luisa Bonet, and Jordi Levy. Polynomial calculus for MaxSAT. In *Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2023. doi:10.4230/LIPIcs.SAT.2023.5.

3 Ilario Bonacina, Maria Luisa Bonet, and Jordi Levy. Polynomial calculus for optimization. *Artificial Intelligence*, 337:104208, 2024. A preliminary version of this work appeared as [2]. doi:10.1016/j.artint.2024.104208.

4 Maria Luisa Bonet, Jordi Levy, and Felip Manyà. A complete calculus for Max-SAT. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing, (SAT)*, volume 4121 of *LNCS*, pages 240–251. Springer, 2006. doi:10.1007/11814948_24.

5 Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8-9):606–618, 2007. doi:10.1016/j.artint.2007.03.001.

6 Michael Brickenstein and Alexander Dreyer. PolyBoRi: A framework for Groebner-basis computations with Boolean polynomials. *Journal of Symbolic Computation*, 44(9):1326–1345, 2009. Effective Methods in Algebraic Geometry. doi:10.1016/j.jsc.2008.02.017.

7 B. Buchberger. A theoretical basis for the reduction of polynomials to canonical forms. *ACM SIGSAM Bulletin*, 10(3):19–29, August 1976. doi:10.1145/1088216.1088219.

8 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. doi:10.1006/JCSS.2000.1726.

9 Matthew Clegg, Jeff Edmonds, and Russell Impagliazzo. Using the Groebner basis algorithm to find proofs of unsatisfiability. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 174–183, 1996. doi:10.1145/237814.237860.

10 Jesús A. De Loera, Jon Lee, Peter N. Malkin, and Susan Margulies. Computing infeasibility certificates for combinatorial problems through Hilbert's Nullstellensatz. *Journal of Symbolic Computation*, 46(11):1260–1283, 2011. doi:10.1016/J.JSC.2011.08.007.

11 Jesús A De Loera, Susan Margulies, Michael Pernpeintner, Eric Riedl, David Rolnick, Gwen Spencer, Despina Stasi, and Jon Swenson. Graph-coloring ideals: Nullstellensatz certificates, Gröbner bases for chordal graphs, and hardness of Gröbner bases. In *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 133–140, 2015. doi:10.1145/2755996.2756639.

12 Jean Charles Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 75–83, 2002. doi:10.1145/780506.780516.

13 Daniela Kaufmann, Paul Beame, Armin Biere, and Jakob Nordström. Adding dual variables to algebraic reasoning for gate-level multiplier verification. In *Proceedings of the 25th Design, Automation and Test in Europe Conference (DATE)*, pages 1431–1436, 2022. doi:10.23919/DATE54114.2022.9774587.

14 Daniela Kaufmann and Armin Biere. Nullstellensatz-proofs for multiplier verification. In *Proceedings of the 22nd International Workshop on Computer Algebra in Scientific Computing (CASC)*, pages 368–389, 2020. doi:10.1007/978-3-030-60026-6_21.

15 Daniela Kaufmann, Armin Biere, and Manuel Kauers. Verifying large multipliers by combining SAT and computer algebra. In *Proceedings of the 19th Conference on Formal Methods in Computer Aided Design (FMCAD)*, pages 28–36, 2019. doi:10.23919/FMCAD.2019.8894250.

16 Daniela Kaufmann, Armin Biere, and Manuel Kauers. From DRUP to PAC and back. In *Proceedigns of the 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 654–657, 2020. doi:10.23919/DATE48585.2020.9116276.

17 Javier Larrosa and Federico Heras. Resolution in Max-SAT and its relation to local consistency in weighted CSPs. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 193–198, 2005. URL: http://ijcai.org/Proceedings/05/Papers/0360.pdf.

18 Jesús A. De Loera, Jon Lee, Susan Margulies, and Shmuel Onn. Expressing combinatorial problems by systems of polynomial equations and Hilbert's Nullstellensatz. *Combinatorics, Probability and Computing*, 18(4):551–582, 2009. doi:10.1017/S0963548309009894.

19 Sasank Mouli. Polynomial calculus sizes over the Boolean and Fourier bases are incomparable. In *Proceedings of the 65th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 790–796, 2024. doi:10.1109/FOCS61266.2024.00055.

20 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.

21 Toby Walsh. SAT v CSP. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 1894, pages 441–456, 2000. doi:10.1007/3-540-45349-0_32.

Redundancy Rules for MaxSAT

Ilario Bonacina
UPC Universitat Politècnica de Catalunya, Barcelona, Spain

Maria Luisa Bonet
UPC Universitat Politècnica de Catalunya, Barcelona, Spain

Sam Buss
University of California, San Diego, CA, USA

Massimo Lauria
Sapienza Università di Roma, Rome, Italy

Abstract

The concept of redundancy in SAT leads to more expressive and powerful proof search techniques, e.g., able to express various inprocessing techniques, and originates interesting hierarchies of proof systems [Heule *et.al*'20, Buss-Thapen'19]. Redundancy has also been integrated in MaxSAT [Ihalainen *et.al*'22, Berg *et.al*'23, Bonacina *et.al*'24].

In this paper, we define a structured hierarchy of redundancy proof systems for MaxSAT, with the goal of studying its proof complexity. We obtain MaxSAT variants of proof systems such as SPR, PR, SR, and others, previously defined for SAT.

All our rules are polynomially checkable, unlike [Ihalainen *et.al*'22]. Moreover, they are simpler and weaker than [Berg *et.al*'23], and possibly amenable to lower bounds. This work also complements the approach of [Bonacina *et.al*'24]. Their proof systems use different rule sets for soft and hard clauses, while here we propose a system using only hard clauses and blocking variables. This is easier to integrate with current solvers and proof checkers.

We discuss the strength of the systems introduced, we show some limitations of them, and we give a short cost-SR proof that any assignment for the weak pigeonhole principle PHP^m_n falsifies at least $m - n$ clauses.

2012 ACM Subject Classification Theory of computation → Proof complexity

Keywords and phrases MaxSAT, Redundancy Rules, Pigeonhole Principle

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.7

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-C21 (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).

Acknowledgements The authors would like to thank the Simons Institute for the Theory of Computing: part of this work has been done during the "Extended Reunion: Satisfiability" program (Spring 2023). Another part of this work has been done during the Oberwolfach workshop 2413 "Proof Complexity and Beyond" and during the 2023 Workshop on Proof Theory and its Applications organized by the Proof Society.

1 Introduction

This paper investigates new proof systems for MaxSAT that incorporate redundancy inferences tailored to work for MaxSAT. Redundancy inferences were introduced as extensions to SAT solvers to allow non-implicational inferences that preserve satisfiability and non-satisfiability. For resolution and SAT solvers, the first redundancy inferences were based on blocked


© Ilario Bonacina, Maria Luisa Bonet, Sam Buss, and Massimo Lauria;
licensed under Creative Commons License CC-BY 4.0



clauses (BC) [25] and Resolution Asymmetric Tautology (RAT) [22, 15]. Other work on redundancy reasoning includes [14, 16, 20, 23, 17]; and, of particular relevance to the present paper, are the work of Heule, Kiesl, and Biere [19], and the work of Buss and Thapen [9]. Redundancy inferences formalize "*without loss of generality*" reasoning [30] and can substantially strengthen resolution and, in some cases, the effectiveness of SAT solvers for hard problems such as the pigeonhole principle (PHP) [20]. Indeed, in their full generality, redundancy inferences allow resolution to polynomially simulate extended resolution.

MaxSAT is a generalization of SAT; it is the problem of determining a truth assignment for a CNF formula that minimizes the number of falsified clauses. Although the MaxSAT problem is inherently more difficult than SAT, in some cases MaxSAT can be adapted to be more efficient in practice than CDCL solvers for hard problems such as PHP [7]. There are several approaches to MaxSAT solvers, including MaxSAT resolution [8, 26], core-guided MaxSAT [12, 1, 28, 27, 29], and maximum-hitting-set MaxSAT [2, 11, 31] ; the present paper discusses only MaxSAT resolution. The MaxSAT resolution proof system was first defined by Larrosa and Heras [26] and proved completed by Bonet, Levy and Manya [8].

We define cost preserving redundancy rules for MaxSAT mirroring the redundancy rules for SAT, called "cost-BC", "cost-LPR", "cost-SPR", "cost-PR", and "cost-SR" (see Definition 3.2). The strongest of these is "cost-SR" based on the substitution redundancy (SR) [9]. All five of these new inferences are sound for MaxSAT reasoning (Theorem 4.2). Furthermore, we prove that cost-SPR, cost-PR and cost-SR are complete for MaxSAT (Theorem 4.3). On the other hand, we prove that cost-LPR and cost-BC are incomplete for MaxSAT (Corollary 5.2). We illustrate the power of cost-SR by giving polynomial size proofs of the cost of the blocking-variable version of weak pigeonhole principle bPHP_n^m for arbitrary numbers $m > n$ of pigeons and holes (Theorem 6.3).

Ours is not the first paper bringing redundancy reasoning to the context of optimization. For instance, the work of Ihalainen, Berg and Järvisalo [21], building on [4], introduced versions of redundancy inferences that work with MaxSAT. In contrast to the system CPR of [21], all of our "cost-" inferences are polynomial-checkable for validity, and thus all give traditional Cook-Reckhow proof systems for MaxSAT.

Another system with redundancy rules for certifying unsatisfiability and optimization is veriPB [5]. 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 [3]. The propositional fragment of veriPB is as strong as Extended Resolution and DRAT, and plausibly even stronger [24]. In contrast, our systems are plausibly weaker and simpler, but yet strong enough to prove efficiently interesting formulas. Still our systems might be amenable to proving lower bounds. Indeed, our explicit goal is to study the proof complexity of redundancy rules for MaxSAT in a similar vein of what [9] does for SAT, something that is beyond the scope of veriPB, focused on proof logging actual solvers. As a starting point, we show that "cost-BC" and "cost-LPR" are not complete (Corollary 5.2) and we show width lower bounds on "cost-SPR" and an analogue of a width lower bound for "cost-SR" (Corollary 5.3). Finally there is another work that tries to modify the redundancy rules to make them compatible with the MaxSAT model of soft and hard clauses [6]. That work has a similar spirit to this one, but departs in the redundancy test, which checks for inclusion rather than for reverse unit propagation. Such choice makes the inference weaker but preserves the number of unsatisfied clauses, hence the rule applies directly to soft clauses.

Before proceeding with the description and analysis of our systems, we should highlight two aspects of redundancy inference that are somehow orthogonal to the choice of the concrete inference rule: clause deletion and new variable introduction. The applicability

of a redundancy rule depends on the clauses present in the database, and it seems that allowing deletion of past clauses makes the system stronger, and indeed collapses together the power of several types of inference rules (see [9]). Likewise the possibility of introducing new variables in redundant clauses makes all such systems as powerful as extended resolution [25]. Coherently with the stated goal of having systems that are simple and amenable to proof complexity analysis, in this paper we do not allow neither clause deletion nor new variables.

Structure of the paper

Section 2 contains all the necessary preliminaries, including notation on MaxSAT and the blocking variables encoding of MaxSAT instances (blocking variables are also used by [21]). Section 3 introduces the redundancy rules for MaxSAT, proves their basic properties, and defines calculi based on those rules. Section 4 shows their soundness, and their completeness. Section 5 shows the incompleteness of cost-BC/cost-LPR and some limitations of cost-SPR and even cost-SR. Section 6 gives examples of applications of the redundancy rules, including a polynomial size proof of the optimal cost of the weak Pigeonhole Principle and a general result about the polynomial size provability of minimally unsatisfiable formulas. Section 7 gives some concluding remarks.

2 Preliminaries

For a natural number n, let $[n]$ be the set $\{1, \dots, n\}$. Sets and multi-sets are denoted with capital Roman or Greek letters.

Propositional logic notation

A *Boolean variable* x takes values in $\{0, 1\}$. A *literal* is either a variable x or its negation $\overline{x}$. A *clause* is a finite disjunction of literals, i.e., $C = \bigvee_i \ell_i$. The empty clause is $\bot$. A formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses $\Gamma = \bigwedge_j C_j$. We identify a CNF with the multiset of its clauses, and denote as $|\Gamma|$ the number of its clauses (counted with multiplicity). We denote as $\mathrm{Var}(\Gamma)$ the set of variables in Γ.

Substitutions and assignments

A *substitution* σ for a set of variables X is a function so that $\sigma(x)$ is either 0, 1 or some literal defined on X. For convenience, we extend a substitution σ to constants and literals, setting $\sigma(0) = 0$, $\sigma(1) = 1$, and $\sigma(\overline{x}) = \overline{\sigma(x)}$ for any variable $x \in X$. The composition of two substitutions σ, τ is the substitution $\sigma \circ \tau$, where $\sigma \circ \tau(x) = \sigma(\tau(x))$ for $x \in X$. A substitution σ is an *assignment* when $\sigma(x) \in \{0, 1, x\}$ for any $x \in X$. The *domain* of an assignment σ is $\mathrm{dom}(\sigma) = \sigma^{-1}(\{0, 1\})$, and σ is a *total assignment* over X if X is its domain, i.e., σ maps all variables in X to Boolean values. Given a clause $C = \bigvee_i \ell_i$ and a substitution σ, the clause C *restricted* by σ, is $C\restriction_\sigma = \bigvee_i \sigma(\ell_i)$, simplified using the usual logic rules, i.e., $D \vee 0 = D$, $D \vee 1 = 1$, and $D \vee \ell \vee \ell = D \vee \ell$. Another notation for $C\restriction_\sigma$ is $\sigma(C)$. If $\sigma(C) = 1$ or $\sigma(C)$ is tautological we say that $\sigma \vDash C$, i.e., σ *satisfies* C.

The *restriction* of a CNF formula Γ by σ, denoted as $\Gamma\restriction_\sigma$, is the conjunction of all clauses $C\restriction_\sigma$ where $C \in \Gamma$ and $\sigma(C) \neq 1$. The CNF $\Gamma\restriction_\sigma$ is also a multiset. We say that σ *satisfies* Γ ($\sigma \vDash \Gamma$) if for every $C \in \Gamma$, $\sigma \vDash C$, i.e., $\Gamma\restriction_\sigma = \emptyset$. We say that $\Gamma \vDash C$ if for every substitution σ, if $\sigma \vDash \Gamma$ then $\sigma \vDash C$.

We identify a literal ℓ with the substitution that assigns ℓ to 1 and leaves all other variables unassigned. Hence we use notations like $\Gamma\restriction_\ell$. Likewise, given a clauses C we denote as $\overline{C}$ the assignment that maps all literals in C to false, and we use the notation $\Gamma\restriction_{\overline{C}}$.

Unit propagation

A *unit clause* is a clause of just one literal. Unit propagation works as follows. Start with a CNF Γ: if Γ has no unit clauses, the process ends, otherwise pick some unit clause ℓ in Γ arbitrarily, remove from Γ all clauses containing ℓ and remove literal $\bar{\ell}$ from all clauses containing it. Keep repeating until Γ has no more unit clauses. Regardless of the choice of the unit clause, the process always ends with the same formula.

We say that $\Gamma \vdash_1 C$ when the application of unit propagation to the formula $\Gamma\restriction_{\overline{C}}$ produces the empty clause. For two CNF formulas Γ, Δ we say that $\Gamma \vdash_1 \Delta$ if for every $D \in \Delta$, $\Gamma \vdash_1 D$. Clearly, if $\Gamma \supseteq \Delta$ then $\Gamma \vdash_1 \Delta$, and if $\Gamma \vdash_1 \Delta$, then $\Gamma \vDash \Delta$. It is important to stress that the $\vdash_1$ relation is *efficiently checkable*.

▶ **Observation 2.1** ([9, Fact 1.3])**.** *Let σ be a substitution and Γ, Δ be CNF formulas, if $\Gamma \vdash_1 \Delta$, then $\Gamma\restriction_\sigma \vdash_1 \Delta\restriction_\sigma$.*

Resolution

Resolution is a well-studied propositional deduction system with two inference rules: (i) from a clause A we can deduce any B s.t. $A \subseteq B$; (ii) from clauses $A \vee x$ and $B \vee \overline{x}$ we can deduce $A \vee B$. A *resolution proof* from a set of clauses Γ is a sequence of clauses $D_1, D_2, \ldots, D_t$ where each D_i is either already in Γ or is deduced from earlier clauses in the sequence using one of the two inference rules. Resolution is complete, thus deciding whether a clause C can be deduced from Γ is the same as deciding whether $\Gamma \vDash C$.

MaxSAT

Given a CNF formula F, MaxSAT asks to find the maximum number of clauses in F which can be simultaneously satisfied. In applications, it is useful to consider a generalization for which we divide the clauses into *hard* or *soft* (*partial* MaxSAT). Hard clauses must be satisfied, while soft clauses can be falsified with a cost. Consider $F = H \wedge S$ where H is the multiset of hard clauses and S is the multiset of soft ones. In this model, MaxSAT asks to find the maximum number of clauses in S that can be simultaneously satisfied by an assignment that satisfies all clauses in H. Observe that the optimization problem is not well defined if H is not satisfiable.[1] It is not relevant whether H is a set or a multiset. In S, on the other hand, the multiplicity of soft clauses must be accounted for.

Proof systems for MaxSAT aim to show lower bounds on the cost of (partial) MaxSAT instances, one such system is MaxSAT resolution.

MaxSAT with blocking variables

Without loss of generality we can assume that all soft clauses in a MaxSAT instance are unit clauses; indeed, using a new variable b, a soft clause C can be replaced with a hard clause $C \vee b$ and a soft clause $\overline{b}$, without affecting the cost. The variable b is usually called a *blocking variable*. This appears in [13], but it might have been used even earlier.

[1] An even more general version is *weighted* MaxSAT, where we would consider *weighted* set of clauses (F, w), i.e., each clauses $C \in F$ has an associated weight $w(C)$ where $w : F \to \mathbb{N} \cup \{\infty\}$. In this model the goal is to minimize the weight of the falsified clauses. The role of the weight ∞ is to model *hard* clauses. In this paper we do not focus on this model.

▶ **Definition 2.2.** *Let $F = H \wedge S$ with soft clauses $S = C_1 \wedge \cdots \wedge C_m$. The* blocking variables formulation *of F is $F' = H' \wedge S'$ where*

- $H' = H \wedge (C_1 \vee b_1) \wedge \cdots \wedge (C_m \vee b_m)$,
- $S' = \overline{b_1} \wedge \cdots \wedge \overline{b_m}$,

and $b_1, \ldots, b_m$ are new variables (blocking variables) *not appearing in F. We say that Γ is a MaxSAT instance* encoded with blocking variables*, when it is given as a set of hard clauses of the form as in H' above. The soft clauses, then, are implicit.*

▶ **Observation 2.3.** *Let $F = H \wedge S$ be a MaxSAT instance and $F' = H' \wedge S'$ be the blocking variables formulation of F. Any assignment that satisfies H and falsifies k clauses in S can be extended to an assignment that satisfies H' and sets k blocking variables to true. Vice versa, any assignment that satisfies H' and sets k blocking variables to true satisfies H too and falsifies at most k clauses in S.*

Because of Observation 2.3, for the rest of this work we consider Γ to be a MaxSAT instance encoded with blocking variables usually named $\{b_1, \ldots, b_m\}$. The goal is to satisfy Γ while setting to true the least number of blocking variables. More formally, given a total assignment α for Γ, we define

$$\mathrm{cost}(\alpha) = \sum_{i=1}^{m} \alpha(b_i) \quad \text{and} \quad \mathrm{cost}(\Gamma) = \min_{\alpha \,:\, \alpha \vDash \Gamma} \mathrm{cost}(\alpha)$$

and the goal is to find the value of $\mathrm{cost}(\Gamma)$. Notice that, the notation $\mathrm{cost}(\alpha)$ is defined even for assignments not satisfying Γ.

3 Redundancy rules for MaxSAT

In the context of SAT, a clause C is redundant w.r.t. a CNF instance Γ if Γ and $\Gamma \cup \{C\}$ are equisatisfiable, that is either they both are satisfiable or both unsatisfiable [25]. The natural adaptation of this notion to MaxSAT is a clause C that does not affect the cost of Γ.

▶ **Definition 3.1** (redundant clause, [21])**.** *A clause C is* redundant *w.r.t. a MaxSAT instance Γ when*

$$\mathrm{cost}(\Gamma) = \mathrm{cost}(\Gamma \cup \{C\}) \,. \tag{1}$$

Clauses that logically follow from Γ are obviously redundant, but there may be other useful clauses that do not follow logically, and yet do not increase the cost if added.

The condition in eq. (1) is not polynomially checkable (unless, say P = NP). Therefore, we consider efficiently certifiable notions of redundancy, i.e., ways to add redundant clauses (in the sense of eq. (1)) while certifying efficiently their redundancy. This is done showing how to extend in a systematic way the notions of efficiently certifiable redundancy already studied in the context of SAT (BC, RAT, LPR, SPR, PR, SR) [19, 9] to the context of MaxSAT. This is an alternative to the approach of [21]. This definition could also be seen as a very special case of veriPB [5] (see Section 3.1 for more details on the connections with veriPB).

▶ **Definition 3.2.** *A clause C is* cost substitution redundant *(*cost-SR*) w.r.t. to Γ if there exists a substitution σ such that*

1. $\Gamma \restriction_{\overline{C}} \vdash_1 (\Gamma \cup \{C\})|_\sigma$ (redundancy)
2. *for all total assignments $\tau \supseteq \overline{C}$,* $\mathrm{cost}(\tau \circ \sigma) \leq \mathrm{cost}(\tau)$ (cost).

If the substitution σ has some additional structure, we have the following redundancy rules listed in decreasing order of generality:

Cost propagation redundant (cost-PR) *if σ is a partial assigment.*

Cost subset propagation redundant (cost-SPR) *if σ is a partial assigment with the same domain as $\overline{C}$. In other words, σ flips some variables in $\overline{C}$.*

Cost literal propagation redundant (cost-LPR) *if σ is a partial assignment with the same domain as $\overline{C}$, but differs from $\overline{C}$ on exactly one variable.*

Cost blocked clause (cost-BC) *if σ is a partial assignment with the same domain as $\overline{C}$, which differs from $\overline{C}$ on exactly one variable v, and moreover, for every clause $D \in \Gamma$ containing the variable v, $\sigma \vDash D$.*[2]

Item 1 in Definition 3.2 claims that adding C does not make Γ unsatisfiable, unless it was already the case. Together with Item 2, it ensures that any assignment that falsifies the new clause C can be patched with a substitution σ so that C is satisfied without increasing the minimum cost (see Lemma 3.4).

Indeed, Item 1 in Definition 3.2 and its variants correspond to the redundancy rules of proof systems SR, PR, SPR, and LPR from [9], adapted here to consider cost. Since LPR is the same as RAT (see [9, Theorem 1.10]), the notion of cost literal propagation redundancy could as well be called *cost*-RAT *redundancy*.

▶ Remark 3.3. It is important to compare Definition 3.2 with [21, Definition 2]. Redundancy conditions are very similar and the main differences are in the cost conditions. Let us compare their `CPR` rule with our cost-PR . In cost-PR, the cost condition requires the witness σ to be at least as good as all possible extensions of $\overline{C}$, while in `CPR` the requirement is enforced only on those extensions of $\overline{C}$ that satisfy Γ. The latter more permissive condition allows to derive more clauses, but it is unlikely to be polynomially checkable, while the condition in cost-PR is polynomially checkable (see Lemma 3.5). In [21], the authors also define two polynomially checkable rules where the cost condition is not present, but implicitly enforced via restrictions on the type of assignments used. Those rules are special cases of cost-LPR and cost-SPR respectively.

▶ **Lemma 3.4.** *If C is* cost-SR *w.r.t. to Γ, then C is redundant w.r.t. Γ.*

Proof. It is enough to show that $\text{cost}(\Gamma) \geq \text{cost}(\Gamma \cup \{C\})$. Let $\text{cost}(\Gamma) = k$. To show that adding C to Γ does not increase the cost, consider an optimal total assignment α that satisfies Γ and sets to true exactly k blocking variables. If $\alpha \vDash C$ we already have that $\alpha \vDash \Gamma \cup \{C\}$ and $\text{cost}(\alpha) = k$. Otherwise, α extends $\overline{C}$ and, by assumption, there is a substitution σ such that $\text{cost}(\alpha \circ \sigma) \leq k$. To show that $\text{cost}(\Gamma \cup \{C\}) \leq k$, it remains to show that indeed $\alpha \circ \sigma \vDash \Gamma \cup \{C\}$. By assumption,

$$\Gamma \restriction_{\overline{C}} \vdash_1 (\Gamma \cup \{C\})|_\sigma \ ,$$

and, since $\alpha \vDash \Gamma$ and extends $\overline{C}$, then $\alpha \vDash (\Gamma \cup \{C\})|_\sigma$ too. Equivalently, $\alpha \circ \sigma \vDash \Gamma \cup \{C\}$. ◀

Both Item 1 and Item 2 of Definition 3.2 are stronger than what is actually needed for Lemma 3.4 to hold. Indeed, for Item 1, it would be enough that $\Gamma \restriction_{\overline{C}} \vDash (\Gamma \cup \{C\})|_\sigma$, and, for Item 2, it would be sufficient to check it for any $\tau \supseteq \overline{C}$ such that $\tau \vDash \Gamma$. Unfortunately, these latter versions of Item 1 and Item 2 are in general not polynomially checkable. Instead, our conditions are checkable in polynomial time.

[2] The definition of *blocked clause* is written to match the previous definitions. In this case, the redundancy condition is always satisfied.

▶ **Lemma 3.5.** *Let Γ be a MaxSAT instance, C a clause and σ a substitution. There is a polynomial time algorithm to decide whether C is* cost-SR *w.r.t. Γ, given the substitution σ.*

Proof (sketch). The redundancy condition in Definition 3.2 is polynomially checkable, since it is a unit propagation. To check the cost condition we need to check whether

$$\max_{\tau \supseteq \overline{C}} (\text{cost}(\tau \circ \sigma) - \text{cost}(\tau)) \tag{2}$$

is at most 0. This amounts to maximizing the function

$$\Big(\sum_{i=1}^{n} \sigma(b_i) - b_i \Big) \Big|_{\overline{C}} \tag{3}$$

over all assignments on variables in $\text{Var}(\Gamma) \setminus \text{Var}(\overline{C})$. Observe that eq. (3) is a function of the form $c + \sum_{v \in \text{Var}(\Gamma) \setminus \text{Var}(\overline{C})} c_v \cdot v$, for suitable constants c and c_vs. Therefore its maximum can be easily determined in polynomial time and corresponds to assigning each v to 1 when $c_v > 0$ and to 0 otherwise. ◀

Lemma 3.5 allows the definition of proof systems that extend resolution using cost redundancy rules in the sense of Cook-Reckhow [10].

▶ **Definition 3.6** (cost-SR calculus)**.** *The* cost-SR calculus *is a proof system for MaxSAT. A derivation of a clause C from a MaxSAT instance Γ (encoded with blocking variables) is a sequence of clauses $D_1, D_2, \ldots, D_t$ where, $C \in \Gamma \cup \{D_i\}_{i \in [t]}$, each D_i is either already in Γ or is deduced from earlier clauses in the sequence using a resolution rule, or D_i is* cost-SR *w.r.t. to $\Gamma \cup \{D_1, \ldots, D_{i-1}\}$ with $\text{Var}(D_i) \subseteq \text{Var}(\Gamma)$.*[3] *The length of such derivation is t, i.e., the number of derived clauses. To check the validity of derivations in polynomial time, any application of the* cost-SR *rule comes accompanied by the corresponding substitution that witnesses its soundness.*

If the goal is to certify that $\text{cost}(\Gamma) \geq s$, we can accomplish this deriving s distinct unit clauses of the form $\{b_{i_1}, \ldots, b_{i_s}\}$ (see Theorem 4.2). If the goal is to certify that $\text{cost}(\Gamma) = s$, we can accomplish this deriving s distinct unit clauses of the form $\{b_{i_1}, \ldots, b_{i_s}\}$ together with the unit clauses $\{\overline{b}_j : j \notin \{i_1, \ldots, i_s\}\}$ (see Theorem 4.2).

In a similar fashion we define cost-PR, cost-SPR, cost-LPR, and cost-BC calculi. A remarkable aspect of these calculi is that a proof must somehow identify the blocking variables to be set to true. When there are multiple optimal solutions, it is quite possible that none of the b_i follows logically from Γ. Nevertheless, the redundancy rules, often used to model "without loss of generality" reasoning, can reduce the solution space.

3.1 Simulation by veriPB

We show that cost-SR and its subsystems are p-simulated by veriPB. This is actually not surprising since veriPB proof steps are based on cutting planes, which in turn easily simulates resolution. Furthermore veriPB includes the SAT redundancy rules equipped with some criteria of cost preservation. veriPB can argue within the proof whether some substitution is cost preserving. If veriPB can do that for the substitutions that respect the second condition in Definition 3.2, then it p-simulates cost-SR.

[3] We only consider the case where no new variables are added via cost-SR rules. To be coherent with [9], cost-SR should be cost-SR$^-$, following the notational convention of adding an exponent with "$-$" to denote SR, PR and SPR when the systems are not allowed to introduce new variables. We ignore that convention to ease an already rich notation.

Since veriPB is a complex system, we describe the minimal fragment needed to simulate cost-SR. Consider a CNF formula F and some objective linear function f to be minimized. A veriPB derivation starts with F encoded as linear inequalities. At each step a linear inequality C is derived from the set Γ of previously derived inequalities in one of the following ways:

1. C is derived from Γ using some cutting planes rule;
2. there exists a substitution σ so that $\Gamma \cup \overline{C} \vdash (\Gamma \cup \{C\}) \restriction_\sigma \cup \{f \restriction_\sigma \leq f\}$,where the symbol $\vdash$ means that there is a cutting planes proof that witnesses the implication.

In Item 2 the cutting plane proof is included in the veriPB proof to make it polynomially verifiable, unless it just consists in unit propagation steps or certain uses of Boolean axioms. In the latter cases the verifier can efficiently recover such derivation steps on its own.

The resolution inference rule of cost-SR can be simulated using 1. The redundancy inference rule of cost-SR is simulated by 2. The redundancy condition is syntactically the same, while the cost condition is argued via a cutting planes proof. The latter is the only non-trivial part. This argument gives a proof sketch of the following proposition.

▶ **Proposition 3.7.** *Let F be a MaxSAT instance over n variables with blocking variables $b_1, b_2, \ldots, b_k$. If there is a t step* cost-SR *derivation of a set of clauses Γ from it, then there is a* veriPB *proof from F and objective $f = \sum_{i=1}^{k} b_i$ of some $\Gamma' \supseteq \Gamma$ in* poly(nt) *steps.*

4 Soundness and completeness

The calculi cost-SR/cost-PR/cost-SPR are *sound* and *complete*. Before proving the soundness we show an auxiliary lemma, that shows that when the calculus certifies the lower bound for the optimal values, it can also certify the optimality.

▶ **Lemma 4.1.** *Let Γ be a MaxSAT instance encoded with blocking variables $b_1, \ldots, b_m$, of* cost$(\Gamma) = k$*, and suppose Γ contains the unit clauses $b_{i_1}, \ldots, b_{i_k}$. Then* cost-PR *can prove $\overline{b}_j$ for each $j \notin \{i_1, \ldots, i_k\}$ in $\mathcal{O}(km)$ steps.*

Proof. Let σ be a total assignment satisfying Γ with cost$(\sigma) = k$, that is σ maps all the b_{i_ℓ}s to 1 and the other blocking variables to 0. We derive all the clauses

$$C_j = \overline{b}_{i_1} \vee \cdots \vee \overline{b}_{i_k} \vee \overline{b}_j$$

with $j \notin \{i_1, \ldots, i_k\}$ using the cost-PR rule. For all clauses C_j, the substitution witnessing the validity of the cost-PR rule is always σ. The redundancy condition from Definition 3.2 is trivially true since Γ union an arbitrary set of C_js is mapped to 1 under σ. The cost condition is true because for every $\tau \supseteq \overline{C_j}$, cost$(\tau) \geq k + 1$ and cost$(\tau \circ \sigma)$ = cost$(\sigma) = k$.

To conclude, by resolution, derive $\overline{b}_j$ from C_j and the unit clauses $b_{i_1}, \ldots, b_{i_k}$. ◀

▶ **Theorem 4.2** (soundness of cost-SR)**.** *Let Γ be a MaxSAT instance encoded with blocking variables. If there is a* cost-SR *proof of k distinct blocking variables, then* cost$(\Gamma) \geq k$*. If there is a* cost-SR *proof of k distinct blocking variables $\{b_{i_1}, \ldots, b_{i_k}\}$ and all the unit clauses $\overline{b}_j$ for $j \notin \{i_1, \ldots, i_k\}$, then* cost$(\Gamma) = k$*.*

Proof. Let $b_1, \ldots, b_m$ be the blocking variables of Γ. Let Γ' the set of clauses in Γ plus all the clauses derived in the proof of cost$(\Gamma) \geq k$. That is Γ' also contains contains k distinct unit clauses $b_{i_1}, \ldots, b_{i_k}$, hence cost$(\Gamma') \geq k$. By Lemma 3.4, the cost is preserved along proof steps, therefore cost(Γ) = cost$(\Gamma') \geq k$. In the case where we have all the $\overline{b}_j$s then cost$(\Gamma') = k$ and therefore cost$(\Gamma) = k$. ◀

As an immediate consequence of Theorem 4.2, also all the cost-PR, cost-SPR, cost-LPR calculi are sound. Moreover, we can always prove the optimal lower bound in cost-SPR calculus.

▶ **Theorem 4.3** (completeness of cost-SPR). *Let Γ be a MaxSAT instance encoded with blocking variables, of* $\mathrm{cost}(\Gamma) = k$. *There is a* cost-SPR *derivation of the unit clauses $b_{i_1}, \ldots, b_{i_k}$ for some distinct k blocking literals and all the $\bar{b}_j$ for $j \notin \{i_1, \ldots, i_k\}$.*

Proof. Let $b_1, \ldots, b_m$ be the blocking variables of Γ. Take α_{opt} to be an optimal assignment, that is $\alpha_{\mathrm{opt}} \vDash \Gamma$, $\mathrm{cost}(\alpha_{\mathrm{opt}}) = k$, and for every total assignment β that satisfies Γ, $\mathrm{cost}(\beta) \geq k$. Without loss of generality we can assume α_{opt} sets variables $b_1, \ldots, b_k$ to 1 and the remaining b_js to 0.

Given any assignment γ, let $\bar{\gamma}$ be the largest clause falsified by γ. Let Σ be the set of all clauses $\bar{\gamma}$ where γ is a total assignment that satisfies Γ is different from α_{opt}. We want to derive Σ from Γ, essentially forbidding any satisfying assignment except for α_{opt}.

We can add all clauses in Σ one by one by the cost-SPR rule. Indeed, for any clause $\bar{\gamma} \in \Sigma$ and any $\Sigma' \subseteq \Sigma$, the clause $\bar{\gamma}$ is cost-SPR w.r.t. $\Gamma \cup \Sigma'$, with α_{opt} as the witnessing assignment. The redundancy condition

$$(\Gamma \cup \Sigma') \restriction_{\gamma} \vdash_1 (\Gamma \cup \Sigma' \cup \bar{\gamma}) \restriction_{\alpha_{\mathrm{opt}}}$$

holds because $\alpha_{\mathrm{opt}} \vDash \Gamma \cup \Sigma' \cup \bar{\gamma}$ so the RHS is just true. The cost condition holds by optimality of α_{opt}. In the end, the only assignment that satisfies $\Gamma \cup \Sigma$ is α_{opt}. By the completeness resolution we can prove all its literals, in particular literals b_i for $1 \leq i \leq k$ and literals $\bar{b}_i$ for $i > k$. ◀

5 Incompleteness and Width Lower bounds

Theorem 4.3 shows the completeness of cost-SPR, hence for cost-PR and cost-SR. It is not a coincidence that the proof does not apply to cost-LPR, indeed the latter is not complete. We will see that for some redundant clause derived according to Definition 3.2, the number of values that partial assignment σ flips with respect to any $\tau \supseteq \overline{C}$ may have to be large. In a cost-LPR proof this number is always at most one. For a redundant clause C derived in some subsystem of cost-SR via a witness substitution σ we define

$$\mathrm{flip}(C, \sigma) = \max_{\tau \supseteq \overline{C}} \mathrm{HammingDistance}(\tau, \tau \circ \sigma) \ ,$$

where HammingDistance is the number of different bits between two total assignments. The following result shows sufficient conditions for $\mathrm{flip}(C, \sigma)$ to be large.

▶ **Theorem 5.1.** *Let Γ be a MaxSAT instance encoded with blocking variables, of* $\mathrm{cost}(\Gamma) = k$, *and let A be the set of optimal total assignments for Γ, i.e., $\alpha \in A$ when $\alpha \vDash \Gamma$ and* $\mathrm{cost}(\alpha) = k$. *If A is such that*

1. *all pairs of assignments in A have Hamming distance at least d, and*
2. *for every blocking variable b there are $\alpha, \beta \in A$ s.t. $\alpha(b) = 0$ and $\beta(b) = 1$,*

then to derive any blocking literal b, cost-SR *must derive a redundant clause with* $\mathrm{flip}(C, \sigma) \geq d$, *where σ is the witnessing substitution for C.*

Proof. Consider a cost-SR derivation from Γ as a sequence $\Gamma_0, \Gamma_1, \ldots, \Gamma_s$ where each $\Gamma_{i+1} := \Gamma_i \cup \{C\}$ with C either derived by resolution from clauses in Γ_i, or C is cost-SR w.r.t. Γ_i. For $0 \leq j \leq s$, let $\mu(j)$ be the number of the optimal assignments for Γ_j.

At the beginning $\mu(0) = |A|$ by construction. If at some point Γ_j contains some clause b_i, then the value $\mu(j)$ must be strictly smaller than $|A|$ because A contains at least some assignment with $\{b_i \mapsto 0\}$ (by assumption 2).

Let j be the first step where $\mu(j)$ drops below $|A|$. The clause C introduced at that moment must be cost-SR w.r.t. Γ_{j-1}, because the resolution steps do not change the set of optimal assignments. Let then σ be the witnessing substitution used to derive C, we have

$$\Gamma_{j-1}\restriction_{\overline{C}} \vdash_1 (\Gamma_{j-1} \cup \{C\})\restriction_{\sigma} \ . \tag{4}$$

Since $\mu(j)$ dropped below $|A|$, clause C must be incompatible with some $\tau \in A$, that is $\tau \supseteq \overline{C}$. Therefore, by cost preservation,

$$\text{cost}(\tau \circ \sigma) \leq \text{cost}(\tau) = k \ . \tag{5}$$

By Observation 2.1, eq. (4) implies that

$$\Gamma_{j-1}\restriction_{\tau} \vdash_1 (\Gamma_{j-1} \cup \{C\})\restriction_{\tau\circ\sigma} \ .$$

Since j was the first moment when $\mu(j) < |A|$ we have that $\tau \vDash \Gamma_{j-1}$ and therefore $\tau \circ \sigma \vDash \Gamma_{j-1} \cup \{C\}$. In particular, $\tau \circ \sigma \vDash \Gamma$. By eq. (5) then it must be $\tau \circ \sigma \in A$. But then, by assumption 1, τ and $\tau \circ \sigma$ have Hamming distance at least d, which implies $\text{flip}(C, \sigma) \geq d$. ◀

We see an example application of this result to cost-LPR, where the number of values allowed to be flipped by the rule is at most 1.

The formula $F = \{x \vee y \vee b_1, \overline{x} \vee b_2, \overline{y} \vee b_3\}$ has cost 1 and its optimal assignments to variables x, y, b_1, b_2, b_3 are $\{00100, 10010, 01001\}$. These assignments fulfil the premises of Theorem 5.1, with Hamming distance > 1. Therefore cost-LPR cannot prove the cost of F to be 1, and hence cost-LPR is incomplete.

▶ **Corollary 5.2.** *Proof systems* cost-LPR*,* cost-BC *are incomplete.*

▶ **Corollary 5.3.** *There is a formula family F_n with $O(n)$ variables, $O(n)$ clauses and $\text{cost}(F_n) = \Omega(n)$ where, in order to prove $\text{cost}(F_n) \geq 1$, any* cost-SR *proof derives a redundant clause with $\text{flip}(C, \sigma) = \Omega(n)$, where σ is the witnessing substitution for C.*

Proof. We define F_n on variables $x_0, x_1, \ldots, x_n$, and variables $y_0, y_1, \ldots, y_n$. The formula contains hard clauses to encode the constraint $x_0 \neq y_0$, and constraints $x_0 = x_i$, $y_0 = y_i$ for $i \in [n]$. Furthermore F_n has the soft clauses, encoded as hard clauses with blocking variables, $\neg x_i \vee b_i$ and $\neg y_i \vee b_{i+n}$ for $i \in [n]$.

To satisfy the formula an assignment must either set all x's to true and y's to false, or vice versa. Both such assignments set to true n of the blocking variables, and no blocking variable is fixed to a constant value. Therefore the claim follows from Theorem 5.1. ◀

▶ **Corollary 5.4.** *Any* cost-SPR *proof for a formula respecting the hypothesis of Theorem 5.1 requires some redundant clauses of width at least d.*

Proof. Any redundant clause C derived in cost-SPR must have $|C| \geq \text{flip}(C, \sigma)$, and by Theorem 5.1, $\text{flip}(C, \sigma) \geq d$. ◀

For instance, the formula F_n from Corollary 5.3 requires cost-SPR refutations of width $\Omega(n)$.

6 Short proofs using redundancy rules

We show applications demonstarting the power of the redundancy rules on notable families of CNF formulas. In Section 6.1 we consider minimally unsatisfiable formulas, while in Section 6.2 we consider the *weak Pigeonhole Principle*.

▶ Remark 6.1. Due to Theorem 4.2 and Theorem 4.3, we refer to a cost-SPR (resp. cost-PR, cost-SR) derivation from Γ of $b_{i_1}, \dots, b_{i_k}$ for some distinct k blocking literals and all the $\overline{b}_j$ for $j \notin \{i_1, \dots, i_k\}$, as a *proof of* $\text{cost}(\Gamma) = k$ in cost-SPR (resp. cost-PR, cost-SR).

6.1 Short proofs of minimally unsatisfiable formulas

Recall the definition of PR from [18] (see also [9, Definition 1.16]). A PR calculus refutation of a CNF formula Γ is a sequence of clauses $D_1, \dots, D_t$ where $D_t = \bot$, and each D_{i+1} is either a clause in Γ, or derived by resolution, or is PR w.r.t. $\Gamma_i = \Gamma \cup \{D_1, \dots, D_i\}$, that is D_{i+1} satisfies

$$\Gamma_i \restriction_{\overline{D_{i+1}}} \vdash_1 (\Gamma_i \cup \{D_{i+1}\})|_\sigma \ ,$$

that is, Item 1 of Definition 3.2 for a σ which is a partial assignment. A PR refutation is a PR derivation of $\bot$. The *size* of a refutation is the number of clauses in it.

An unsatisfiable set Γ of clauses is *minimally unsatisfiable* if no proper subset of Γ is unsatisfiable.

▶ **Theorem 6.2.** *If a minimally unsatisfiable CNF formula* $\{C_1, \dots, C_m\}$ *has a* PR *refutation of size* s*, then there is a* cost-PR *proof of* $\text{cost}(\{C_1 \vee b_1, \dots, C_m \vee b_m\}) = 1$ *of at most* $\mathcal{O}(s+m)$ *many clauses.*

Proof. Let $F = \{C_1, \dots, C_m\}$ and $\Gamma = \{C_1 \vee b_1, \dots, C_m \vee b_m\}$ be the corresponding MaxSAT instance. Let $\pi = (D_1, \dots, D_s)$ be a PR refutation of F, so $D_s = \bot$. First we show that

$$\pi_B = (D_1 \vee B, \dots, D_s \vee B) \ ,$$

with $B = \bigvee_{i \in [m]} b_i$, is a valid cost-PR derivation of B from Γ. In particular, assuming we already derived the first i steps of π_B, we show how to derive $D_{i+1} \vee B$.

When $D_{i+i} \in F$, the clause $D_{i+i} \vee B$ is the weakening of some clause in Γ. If D_{i+1} was derived using a resolution rule on some premises in π, then $D_{i+i} \vee B$ can be derived in the same way from the corresponding premises in π_B. The remaining case is when D_{i+1} is PR w.r.t. $F_i = F \cup \{D_1, \dots, D_i\}$. Let α be the assignment that witnesses it. This assignment only maps variables from the original formula F, so we extend it to $\alpha' = \alpha \cup \{b_1 \mapsto 0, \dots, b_m \mapsto 0\}$, and then use α' to witness that indeed $D_{i+1} \vee B$ is cost-PR w.r.t. $\Gamma_i = \Gamma \cup \{D_1 \vee B, \dots D_i \vee B\}$. For the cost condition in Definition 3.2, just observe that any extension of α' has cost 0. For the redundancy condition, observe that, by construction, $\Gamma_i \restriction_{\overline{D_{i+1}} \wedge \overline{B}} = F_i \restriction_{\overline{D_{i+1}}}$, $(F_i \cup \{D_{i+1}\}) \restriction_\alpha = (\Gamma_i \cup \{D_{i+1} \vee B\}) \restriction_{\alpha'}$, and $F_i \restriction_{\overline{D_{i+1}}} \vdash_1 (F_i \cup \{D_{i+1}\}) \restriction_\alpha$.

The last clause of π_B is B. Let α_{opt} be an optimal assignment of Γ. Since F is minimally unsatisfiable, $\text{cost}(\alpha_{opt}) = 1$. W.l.o.g. assume α_{opt} sets $b_m = 1$ and all $b_i = 0$ for $i < m$.

Now, for each $i < m$, the clause $E_i = \overline{b}_i \vee b_m$ is cost-PR w.r.t. $\pi_B \cup \{E_j : j < i\}$, using α_{opt} itself as the witnessing assignment: redundancy holds since α_{opt} satisfies every clause in π_B and all clauses E_j. The cost condition follow since $\text{cost}(\tau) \geq 1$ for any $\tau \supseteq \overline{E_i}$ and $\text{cost}(\tau \circ \alpha_{opt}) = \text{cost}(\alpha_{opt}) = 1$.

In the end we use $O(m)$ steps to derive b_m from B and $E_1, \dots, E_{m-1}$, and to derive in cost-PR calculus all the units $\overline{b}_1, \dots, \overline{b}_{m-1}$ via Lemma 4.1. ◀

Theorem 6.2 shows that the propositional refutations for the minimally unsatisfiable formulas in [9] translate immediately to certificates in the MaxSAT. In particular, as a corollary of Theorem 6.2, we have that cost-PR proves in polynomial size that

- the *Pigeonhole Principle* with $n+1$ pigeons and n holes [9, Theorem 4.3] and [19, Section 5],
- the *Bit-Pigeonhole Principle* [9, Theorem 4.4],
- the *Parity Principle* [9, Theorem 4.6],
- the *Tseitin Principle* on a connected graph [9, Theorem 4.10],

have all cost 1, since they are all minimally unsatisfiable. In MaxSAT resolution that would require exponentially long derivations.

6.2 Short proofs of the minimum cost of PHP_n^m

Let $m > n \geq 1$. The pigeonhole principle from m pigeons to n holes, with blocking variables, has the following formulation, that we call bPHP_n^m: the *totality* clauses $\bigvee_{j\in[n]} p_{i,j} \vee b_i$ for $i \in [m]$, and the *injectivity* clauses $\overline{p}_{i,j} \vee \overline{p}_{k,j} \vee b_{i,k,j}$ for $1 \leq i < k \leq m$ and $j \in [n]$. We use $b_{k,i,j}$ as an alias of the variable $b_{i,k,j}$, given that $i < k$.

▶ **Theorem 6.3.** cost-SR *proves* $\mathrm{cost}(\mathsf{bPHP}_n^m) = m - n$ *in polynomial size.*

This is the main result of the section. Before proving it we show two useful lemmas. The first lemma is used to "clean up" the set of clauses during a derivation. For each new step in a cost-SR calculus derivation the redundancy condition must be checked against an ever increasing set of clauses. It turns out that some already derived clauses can be completely ignored for the rest of the derivation under some technical conditions. This makes up for the lack of a deletion rule, that we do not have, and in the context of SAT seems to give more power to the systems [9].

▶ **Lemma 6.4.** *Let* Γ *and* Σ *be two sets of clauses. Any* cost-SR *derivation* $D_1 \dots, D_t$ *from* Γ *is also a valid derivation from* $\Gamma \cup \Sigma$ *if either of the two cases applies*

1. *Variables in* Σ *do not occur in* $\Gamma \cup \{D_1, \dots, D_t\}$.
2. *For every clause* $C \in \Sigma$ *there is a clause* $C' \in \Gamma$ *so that* $C' \subseteq C$.

Proof. The cost condition does not depend on the set of clauses, therefore we only need to check the validity of the redundancy condition. In the first case, the redundancy condition applies because the clauses of Σ are unaffected by the substitutions involved.

For the second case, consider the derivation of a clause D_i witnessed by σ_i. The clauses in $\Sigma\restriction_{\overline{D_i}}$ and $\Sigma\restriction_{\sigma_i}$ are subsumed by clauses in $\Gamma\restriction_{\overline{D_i}}$ and $\Gamma\restriction_{\sigma_i}$, respectively. Hence

$$(\Gamma \cup \{D_1, \dots, D_{i-1}\})\restriction_{\overline{D_i}} \vdash_1 (\Gamma \cup \Sigma \cup \{D_1, \dots, D_i\})\restriction_{\sigma_i}$$

which implies the validity of the redundancy condition. ◀

The second lemma is used as a general condition to enforce clauses to be cost-SR.

▶ **Lemma 6.5.** *Let* C *be a clause and* Γ *a set of clauses. If there exists a permutation* π *such that*

1. π *maps the set of blocking variables to itself,*
2. *the substitution* $\overline{C} \circ \pi$ *satisfies* C*, and* $\Gamma\restriction_{\overline{C}} \supseteq \Gamma\restriction_{\overline{C}\circ\pi}$,

then C *is* cost-SR *w.r.t.* Γ*. Notice that the second condition in item* (2) *is automatically satisfied if* π *is a symmetry of* Γ*, i.e.* $\Gamma = \Gamma\restriction_{\pi}$.

Proof. The cost condition follows from Item 1. The redundancy condition is immediate by Item 2 using as σ the substitution $\overline{C} \circ \pi$. ◀

Proof of Theorem 6.3. The proof is by induction. The goal is to reduce the formula to $m-1$ pigeons and $n-1$ holes. First we do some preprocessing: from bPHP_n^m we derive a slightly more structured formula F_n^m. Then we show how to derive F_{n-1}^{m-1} in a polynomial number of steps. The results follows because after n such derivations we obtain the formula F_0^{m-n} that contains the clauses $b_1, \ldots, b_{m-n}$. Moreover, we also derive $\overline{b_{m-n+1}}, \ldots, \overline{b_m}$ along the way.

We derive F_{n-1}^{m-1} from F_n^m using the rules of cost-SR calculus. We divide the argument into several steps, but first we show how to derive F_n^m from bPHP_n^m.

Preprocessing 1. *"Make b_i full-fledged extension variables".* [4] Turn all the variables b_i into full-fledged extension variables that satisfy $b_i \leftrightarrow \neg(p_{i,1} \vee \cdots \vee p_{i,n})$, by adding the clauses

$$\mathsf{Ext} = \{\overline{p_{i,j}} \vee \overline{b_i} : i \in [m], j \in [n]\}$$

one by one in cost-LPR.

We need to derive clause $D_j = \overline{p_{1,j}} \vee \overline{b_1}$ for every $j \in [n]$. Assume that we already got $D_1, \ldots, D_{j-1}$, we derive D_j as a cost-LPR clause w.r.t. $\mathsf{bPHP}_n^m \cup \{D_1, \ldots, D_{j-1}\}$. The witnessing assignment is $\sigma_j := \{p_{1,j} = 1, b_1 = 0\}$. Since $\overline{D_j} = \{p_{1,j} = 1, b_1 = 1\}$, the cost condition is satisfied. The redundancy condition follows from

$$\mathsf{bPHP}_n^m \restriction_{\overline{D_j}} \supseteq (\mathsf{bPHP}_n^m \cup \{D_1, \ldots, D_j\}) \restriction_{\sigma_j} \ .$$

Indeed, on clauses of bPHP_n^m that do not contain the variable b_j, the assignments $\overline{D_j}$ and σ_j behave identically, while all the clauses containing b_j are satisfied by σ_j. Repeat the previous argument to get all the clauses $\overline{p_{i,j}} \vee \overline{b_i}$. The current database of clauses is $\mathsf{bPHP}_n^m \cup \mathsf{Ext}$.

Preprocessing 2. *"Enforce injectivity".* Optimal assignments for bPHP_n^m can have unassigned pigeons or have collisions between pigeons. It is more convenient to avoid collisions and just focus on assignments that are partial matching. A moment's thought suffices to realize that such restriction does not change the optimal cost but simplifies the solution space. We enforce collisions to never occur by deriving all the unit clauses $\overline{b_{i,k,j}}$ by cost-PR.

These clauses can be derived in any particular order: to show that $\overline{b_{i,k,j}}$ is cost-SR w.r.t. Γ_0 and the previously derived $\overline{b_{i',k',j'}}$ we pick one of the two pigeons involved (say k) and use $\sigma = \{b_{i,k,j} = 0, b_k = 1, p_{k,1} = \cdots = p_{k,n} = 0\}$ as the witnessing assignment. The cost is not increased, and to check the redundancy condition observe that σ satisfies all the clauses that touches, so on the right side of the redundancy condition has a subset of $\mathsf{bPHP}_n^m \cup \mathsf{Ext}$ with no occurrences $b_{i,k,j}$, while the left side has the same set of clauses, but restricted with $b_{i,k,j} = 0$.

Now that we have all clauses $\overline{b_{i,k,j}}$ we resolve them with the corresponding clauses $\overline{p_{i,j}} \vee \overline{p_{k,j}} \vee b_{i,k,j}$ to get the set of clauses

$$\mathsf{Inj} = \{\overline{p_{i,j}} \vee \overline{p_{k,j}} : 1 \leq i < k \leq m \text{ and } j \in [n]\} \ ,$$

for all holes j and pair of pigeons i and k.

[4] This is true in general: if a MaxSAT instance contains a clause $C \vee b$ then it is possible to make the blocking variable b a full-fledged extension variable ($b \leftrightarrow \overline{C}$) by cost-LPR.

We do not need variables $b_{i,k,j}$ anymore. By one application of Lemma 6.4, from now on we can ignore all clauses $\overline{p_{i,j}} \vee \overline{p_{k,j}} \vee b_{i,k,j}$. By another application, we can also ignore the clauses $\overline{b_{i,k,j}}$. We will do induction on the current database of clauses.

For clarity we list all its clauses again.

$$\text{Formula } \mathsf{F}_n^m \quad \begin{array}{lll} \bigvee_{j\in[n]} p_{i,j} \vee b_i & \text{for } i \in [m] & (\textit{totality 1}), \\ \overline{p_{i,j}} \vee \overline{b_i} & \text{for } i \in [m] \text{ and } j \in [n] & (\textit{totality 2}), \\ \overline{p_{i,j}} \vee \overline{p_{k,j}} & \text{for } 1 \le i < k \le m \text{ and } j \in [n] & (\textit{injectivity}). \end{array}$$

The core idea of the induction is that if a pigeon flies to a hole, we can assume without loss of generality that it is pigeon m that flies into hole n.

Step 1. *"If some pigeon i flies, we can assume it is pigeon m who flies".* We want to derive, in this order, the set of clauses

$$\Delta_1 = \{\overline{b_m} \vee b_1, \overline{b_m} \vee b_2, \ldots, \overline{b_m} \vee b_{(m-1)}\}$$

from F_n^m, to claim that if some pigeon is mapped, then pigeon m is mapped too. For each $C_i = \overline{b_m} \vee b_i$ we apply Lemma 6.5 using as the witnessing permutation π_i, the permutation that swaps pigeons m and i.

Namely, $\pi_i(p_{m,j}) = p_{i,j}$, $\pi_i(p_{i,j}) = p_{m,j}$, $\pi_i(b_m) = b_i$, $\pi_i(b_i) = b_m$, and π_i is the identity on all other variables, therefore π_i satisfies the first requirement for the lemma. Likewise $\overline{C_i} \circ \pi_i \vDash C_i$, and we need to check that

$$(\mathsf{F}_n^m \cup \{C_1, \ldots, C_{(i-1)}\}) \restriction_{\overline{C_i}} \supseteq (\mathsf{F}_n^m \cup \{C_1, \ldots, C_{(i-1)}\}) \restriction_{\overline{C_i} \circ \pi_i} .$$

By symmetry $\mathsf{F}_n^m \restriction_{\overline{C_i}} = \mathsf{F}_n^m \restriction_{\overline{C_i} \circ \pi_i}$, and for $1 \le i' < i$, $C_{i'} \restriction_{\overline{C_i} \circ \pi_i} = 1$, hence the inclusion is true. The current database of clauses is $\Gamma_1 = \mathsf{F}_n^m \cup \Delta_1$.

Step 2. *"If pigeon m flies to some hole, we can assume it flies to hole n".* Using cost-SR inferences, we derive from Γ_1, in this order, the clauses

$$\Delta_2 = \{\overline{p_{m,1}} \vee p_{m,n}, \overline{p_{m,2}} \vee p_{m,n}, \ldots, \overline{p_{m,(n-1)}} \vee p_{m,n}\}$$

expressing that if pigeon m flies to some hole, this hole is the last one.

For each $C_j = \overline{p_{m,j}} \vee p_{m,n}$ we apply Lemma 6.5 with the witnessing permutation π_j swapping holes n and j.

Namely $\pi_j(p_{i,n}) = p_{i,j}$ and $\pi_j(p_{i,j}) = p_{i,n}$, and π_j is the identity on all other variables. By construction π_j satisfies the first requirement for the lemma, and likewise $\overline{C_j} \circ \pi_j \vDash C_j$, and, again, we need to check

$$(\Gamma_1 \cup \{C_1, \ldots, C_{(j-1)}\}) \restriction_{\overline{C_j}} \supseteq (\Gamma_1 \cup \{C_1, \ldots, C_{(j-1)}\}) \restriction_{\overline{C_j} \circ \pi_j} .$$

By symmetry $\Gamma_1 \restriction_{\overline{C_j}} = \Gamma_1 \restriction_{\overline{C_j} \circ \pi_j}$, and for $1 \le j' < j$, $C_{j'} \restriction_{\overline{C_j} \circ \pi_j} = 1$, hence the inclusion is true. The current database of clauses is $\Gamma_2 = \Gamma_1 \cup \Delta_2 = \mathsf{F}_n^m \cup \Delta_1 \cup \Delta_2$.

Step 3. *"Obtain $\overline{p_{k,n}}$ for every $1 \le k < m$ via resolution".* Resolve the clause $(p_{m,1} \vee p_{m,2} \vee \cdots \vee p_{m,n} \vee b_m)$ (totality 1) with $\overline{p_{m,n}} \vee \overline{p_{k,n}}$, the resulting clause with all clauses $\overline{p_{m,j}} \vee p_{m,n}$ from step 2, to get $b_m \vee p_{m,n} \vee \overline{p_{k,n}}$. Then resolve $b_m \vee p_{m,n} \vee \overline{p_{k,n}}$ again with the injectivity clause $\overline{p_{m,n}} \vee \overline{p_{k,n}}$, then the result with clause $\overline{b_m} \vee b_k$ (from step 1), and again this latter result with clause $\overline{b_k} \vee \overline{p_{k,n}}$ (totality 2). The final result is $\overline{p_{k,n}}$.

The clauses $\overline{p_{k,n}}$ subsume the clauses in Inj of the form $\overline{p_{m,n}} \vee \overline{p_{k,n}}$ and all the intermediate clauses from the previous resolution steps. Therefore we use Lemma 6.4 to be able to ignore the subsumed clauses.

The current database of clauses is Γ_3 is equal to

$$\mathsf{F}_n^m \cup \Delta_1 \cup \Delta_2 \cup \{\overline{p_{k,n}} : 1 \le k < m\} \setminus \{\overline{p_{m,n}} \vee \overline{p_{k,n}} : 1 \le k < m\}.$$

Step 4. *"Assign pigeon m to hole n: derive unit clauses $p_{m,n}$ and $\overline{b_m}$".* The goal is to enforce pigeon m to be mapped to hole n, by deriving the clause $p_{m,n}$ using the cost-PR rule. Then we get $\overline{b_m}$ immediately by resolving $p_{m,n}$ with $\overline{p_{m,n}} \vee \overline{b_m}$ (totality 2).

The unit clause $p_{m,n}$ is cost-PR w.r.t. Γ_3, using partial assignment $\sigma = \{p_{m,n} = 1, b_m = 0\}$ as witness.

Clearly σ satisfies the cost condition. To see that the redundancy condition holds as well, we need to show that $\Gamma_3|_{\overline{C}} \vdash_1 D|_\sigma$ for all D in Γ_3 that contain $\overline{p_{m,n}}$, but the only such clause that remains in Γ_3 is $\overline{p_{m,n}} \vee \overline{b_m}$, which is satisfied by σ. The current database of clauses is $\Gamma_4 = \Gamma_3 \cup \{p_{m,n}, \overline{b_m}\}$.

Step 5. *"Derive $\overline{p_{m,1}}, \dots, \overline{p_{m,(n-1)}}$ by* cost-SR*".* We can derive them in any order using as witnessing substitution of the cost-SR rule the assignment σ setting $p_{m,n} = 1$, $p_{m,1} = \cdots = p_{m,(n-1)} = 0$, and $b_m = 0$.

The cost condition is immediate, and the redundancy condition follows from the fact that $\Gamma_4 \restriction_\sigma \subseteq \Gamma_4$.

Step 6. *"Reduction to $m-1$ pigeons and $n-1$ holes".* First we derive by unit propagation all the the totality clauses of F_{n-1}^{m-1}. That is, we remove the hole n from the totality axioms of the pigeons $1, \dots, m-1$ in the current database. Now, the current database is F_{n-1}^{m-1}, the unit clauses $\overline{b_m}$, $p_{m,n}$, $\overline{p_{k,n}}$ for $k \neq m$ and $\overline{p_{m,j}}$ for $j \neq n$, and clauses that are subsumed by one of these unit clauses. Therefore by Lemma 6.4 we can ignore all the unit clauses and all the clauses subsumed by them. That is we can carry on the derivation using only F_{n-1}^{m-1}.

Thus steps (1)–(6) are repeated $n-1$ times, up to derive F_0^{m-n}.

The unit clauses derived in the whole process include

- $b_1, \dots, b_{(m-n)}$ (totality clauses in F_0^{m-n}).
- $\overline{b_{n+1}}, \dots, \overline{b_m}$ (derived at each step of the induction),
- $\overline{b_{i,k,j}}$ for all $i < k$ and j (derived at the preprocessing).

Therefore $\mathrm{cost}(\mathsf{bPHP}_n^m) = m - n$. ◀

7 Conclusions and open problems

We proposed a way to extend redundancy rules, originally introduced for SAT, into polynomially verifiable rules for MaxSAT. We defined sound and complete calculi based on those rules and we showed the strength of some of the calculi giving short derivations of notable principles and we showed the incompleteness of the weaker ones and width lower bounds for the stronger ones. We conclude this article with a list of open problems:

1. The cost constraint for the redundancy rules is very strict, for example compared to the rule `CPR` in [21]. Indeed, cost-PR enforces the check on the cost even on assignments falsifying the hard clauses of the formula. Is it possible to relax cost-PR without giving up on efficient verification as in [21]?
2. Does cost-SR simulate MaxSAT Resolution? That is, if we have a MaxSAT instance Γ with blocking variables and MaxSAT Resolution proves in size s that $\mathrm{cost}(\Gamma) = k$, is there a proof of $\mathrm{cost}(\Gamma) = k$ in cost-SR of size $\mathsf{poly}(s)$?
3. We proved a width lower bound for cost-SPR and an analogue of a width lower bound for cost-SR on formulas with optimal assignments far from each other in the Hamming distance. We reiterate the open problem of proving size lower bounds for cost-SPR and stronger systems.

References

1 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. SAT-based MaxSAT algorithms. *Artif. Intell.*, 196:77–105, 2013. doi:10.1016/J.ARTINT.2013.01.002.

2 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.

3 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)*, volume 14132, pages 1–22, 2023. doi:10.1007/978-3-031-38499-8_1.

4 Jeremias Berg and Matti Järvisalo. Unifying reasoning and core-guided search for maximum satisfiability. In *Proceedings of the 16th European Conference on Logics in Artificial Intelligence (JELIA)*, volume 11468, pages 287–303, 2019. doi:10.1007/978-3-030-19570-0_19.

5 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, 2023. doi:10.1613/JAIR.1.14296.

6 Ilario Bonacina, Maria Luisa Bonet, and Massimo Lauria. MaxSAT Resolution with Inclusion Redundancy. In *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 305, pages 7:1–7:15, 2024. doi:10.4230/LIPIcs.SAT.2024.7.

7 Maria Luisa Bonet, Sam Buss, Alexey Ignatiev, João Marques-Silva, and António Morgado. MaxSAT resolution with the dual rail encoding. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, (AAAI), the 30th innovative Applications of Artificial Intelligence (IAAI), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI)*, pages 6565–6572, 2018. doi:10.1609/AAAI.V32I1.12204.

8 Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8-9):606–618, 2007. doi:10.1016/J.ARTINT.2007.03.001.

9 Sam Buss and Neil Thapen. DRAT and propagation redundancy proofs without new variables. *Logical Methods in Computer Science*, 17(2), 2021. URL: https://lmcs.episciences.org/7400.

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

11 Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 225–239, 2011. doi:10.1007/978-3-642-23786-7_19.

12 Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 252–265, 2006. doi:10.1007/11814948_25.

13 Michel X. Goemans and David P. Williamson. New $\frac{3}{4}$-approximation algorithms for the maximum satisfiability problem. *SIAM Journal on Discrete Mathematics*, 7(4):656–666, 1994. doi:10.1137/S0895480192243516.

14 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Proceedings of the conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 181–188, 2013. URL: https://ieeexplore.ieee.org/document/6679408/.

15 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE)*, volume 7898, pages 345–359, 2013. doi:10.1007/978-3-642-38574-2_24.

16 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In *Proceedings of the 25th International Conference on Automated Deduction (CADE)*, volume 9195, pages 591–606, 2015. doi:10.1007/978-3-319-21401-6_40.

17 Marijn J. H. Heule and Armin Biere. What a difference a variable makes. In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS)*, volume 10806, pages 75–92, 2018. doi:10.1007/978-3-319-89963-3_5.

18 Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In *Proceedings of the 26th International Conference on Automated Deduction (CADE)*, volume 10395, pages 130–147, 2017. doi:10.1007/978-3-319-63046-5_9.

19 Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong extension-free proof systems. *Journal of Automated Reasoning*, 64(3):533–554, 2019. Extended version of [18]. doi:10.1007/s10817-019-09516-0.

20 Marijn J. H. Heule, Benjamin Kiesl, Martina Seidl, and Armin Biere. Pruning through satisfaction. In *Proceedings of the Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference (HVC)*, volume 10629, pages 179–194, 2017. doi:10.1007/978-3-319-70389-3_12.

21 Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Clause redundancy and preprocessing in maximum satisfiability. In *Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR)*, volume 13385, pages 75–94, 2022. doi:10.1007/978-3-031-10769-6_6.

22 Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364, pages 355–370, 2012. doi:10.1007/978-3-642-31365-3_28.

23 Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. Extended resolution simulates DRAT. In *Proceedings of the 9th International Joint Conference on Automated Reasoning (IJCAR), Held as Part of the Federated Logic Conference, FloC 2018*, volume 10900, pages 516–531, 2018. doi:10.1007/978-3-319-94205-6_34.

24 Leszek Aleksander Kołodziejczyk and Neil Thapen. The Strength of the Dominance Rule. In *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 305, pages 20:1–20:22, 2024. doi:10.4230/LIPIcs.SAT.2024.20.

25 Oliver Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97:149–176, 1999. doi:10.1016/S0166-218X(99)00037-2.

26 Javier Larrosa and Federico Heras. Resolution in max-sat and its relation to local consistency in weighted csps. In *Proceedings of the 19h International Joint Conference on Artificial Intelligence (IJCAI)*, pages 193–198, 2005. URL: http://ijcai.org/Proceedings/05/Papers/0360.pdf.

27 António Morgado, Carmine Dodaro, and João Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 564–573, 2014. doi:10.1007/978-3-319-10428-7_41.

28 António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints An Int. J.*, 18(4):478–534, 2013. doi:10.1007/S10601-013-9146-2.

29 Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided MaxSAT resolution. In *Proceedings of the 28th Conference on Artificial Intelligence (AAAI)*, pages 2717–2723, 2014. doi:10.1609/AAAI.V28I1.9124.

30 Adrián Rebola-Pardo and Martin Suda. A theory of satisfiability-preserving proofs in SAT solving. In *Proceedings of the 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 57, pages 583–603, 2018. doi:10.29007/TC7Q.

31 Paul Saikko, Jeremias Berg, and Matti Järvisalo. LMHS: a SAT-IP hybrid MaxSAT solver. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 539–546, 2016. doi:10.1007/978-3-319-40970-2_34.

Certifying Projected Knowledge Compilation

Randal E. Bryant
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Yong Kiam Tan
Institute for Infocomm Research (I^2R), A*STAR, Singapore
Nanyang Technological University, Singapore

Marijn J. H. Heule
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Knowledge compilers convert Boolean formulas, given in conjunctive normal form (CNF), into representations that enable efficient evaluation of unweighted and weighted model counts, as well as a variety of other useful properties. With *projected* knowledge compilation, the generated representation describes the restriction of the formula to a designated set of *data* variables, with the remaining ones eliminated by existential quantification. Projected knowledge compilation has applications in a variety of domains, including formal verification and synthesis.

This paper describes a formally verified proof framework for certifying the output of a projected knowledge compiler. It builds on an earlier clausal proof framework for certifying the output of a standard knowledge compiler. Extending the framework to projected compilation requires a method to represent Skolem assignments, describing how the quantified variables can be assigned, given an assignment for the data variables. We do so by extending the representation generated by the knowledge compiler to also encode Skolem assignments. We also refine the earlier framework, moving beyond purely clausal proofs to enable scaling certification to larger formulas.

We present experimental results obtained by making small modifications to the D4 projected knowledge compiler and extensions of our earlier proof generator. We detail a soundness argument stating that a compiler output that passes our certifier is logically equivalent to the quantified input formula; the soundness argument has been formally validated using the HOL4 proof assistant. The checker also ensures that the compiler output satisfies the properties required for efficient unweighted and weighted model counting. We have developed two proof checkers for the certification framework: one written in C and designed for high performance and one written in CakeML and formally verified in HOL4.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Knowledge Compilation, Propositional model counting, Proof checking

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.8

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.15628075`

Funding *Randal E. Bryant*: National Science Foundation, NSF grant CCF-2108521.
Yong Kiam Tan: Singapore NRF Fellowship Programme NRF-NRFF16-2024-0002.
Marijn J. H. Heule: National Science Foundation, NSF grant CCF-2108521.

1 Motivation: (Projected) Knowledge Compilation

Although Boolean satisfiability (SAT) solvers serve as effective tools for many automated reasoning tasks, some applications require evaluating more detailed properties about formulas than whether or not they are satisfiable. For example, *model counting* [26] determines the number of satisfying assignments to a formula, while *probabilistic inference* [11] determines the probability that a formula will evaluate to 1 (true), given individual probabilities that


© Randal E. Bryant, Yong Kiam Tan, and Marijn J. H. Heule;
licensed under Creative Commons License CC-BY 4.0



each variable is assigned value 1. Probabilistic inference is a special case of the more general problem of *weighted model counting*. Generalizing to weight functions computed over algebraic semirings expands properties that can be evaluated even further [28].

Rather than implementing specialized programs for these different analysis tasks, a *knowledge compiler* [18, 34] converts a Boolean formula into a form for which the tasks can be performed in polynomial time, relative to the size of the representation. Most commonly, knowledge compilers generate a restricted class of Boolean formulas for which the analysis becomes tractable, such as the deterministic, decomposable, negation-normal form (d-DNNF, defined in Section 4) [18]. These formulas are encoded as directed acyclic graphs, enabling efficient sharing of common subformulas. We can therefore state the operation of a knowledge compiler as converting a Boolean formula over a set of variables X, denoted $\phi(X)$, into a logically equivalent formula $\psi(X)$, such that (weighted) model counting can be performed in time polynomial in the size of ψ.

With *projected* knowledge compilation, the variables X are partitioned into *data* variables D and *quantified* variables Y, and so the input formula has the form $\phi(D, Y)$. The task is to produce a formula $\psi(D)$ where the quantified variables have been eliminated via existential quantification. That is, the satisfying assignments of ψ consist of those assignments α to the data variables, such that for some assignment β to the quantified variables, their combination $\alpha \cdot \beta$ satisfies the input formula ϕ. This can be written as $\psi(D) \equiv \exists Y\, \phi(D, Y)$. This form of projection is also referred to as "forgetting" [18, 21, 33].

The need to eliminate quantified variables arises in many contexts. For example, auxiliary variables are commonly introduced when encoding general Boolean formulas into conjunctive normal form (CNF), via either a Tseitin [38] or a Plaisted-Greenbaum [37] encoding. Although the former preserves model counts, the latter does not, and both can complicate the task of weighted model counting. As a second example, when a hardware system is described as the composition of a set of components, Boolean variables are introduced to encode the connections between components [13]. Existentially quantifying these variables abstracts away the internal system details. As a third example, a state transition system can be described via a *transition relation* $T(S, S')$ defining the allowed combinations of current states S and next states S' [8]. For some predicate $P(S)$ describing the current states, the quantified formula $P'(S') \doteq \exists S\, [P(S) \wedge T(S, S')]$ describes the set of possible successor states.

Our work makes use of version 2 of the D4 knowledge compiler [30].[1] Given a CNF file that also includes a declaration of the data variables (referred to as "show" variables), as is required for the projected model counting competitions,[2] it will generate a d-DNNF representation of the set of data models.

2 Certification Framework and Tools

For many applications of automated reasoning, including formal verification and mathematics, it is essential that the tools be fully trustworthy. In our context, there should be some method of ensuring that the output $\psi(D)$ generated by a projected knowledge compiler is truly equivalent to $\exists Y\, \phi(D, Y)$. We have devised a proof framework for expressing such proofs and a set of tools both to generate and to check these proofs. Such a proof enables the equivalence to be established for individual executions of the compiler, even if the overall correctness

[1] Available at `https://github.com/crillab/d4v2`
[2] `https://mccompetition.org`

of the compiler cannot be guaranteed. In prior work [6, 7], we devised a framework for certifying the results of standard knowledge compilation. This paper extends this framework and associated tools to handle projected knowledge compilation.

For standard knowledge compilation, the task is to prove that $\phi(X) \equiv \psi(X)$, i.e., that the CNF and the d-DNNF representations are logically equivalent. In our framework, d-DNNF formulas are represented by directed acyclic graphs that we refer to as "partitioned-operation graphs" (POGs). These graphs have Boolean variables and their complements as terminal nodes, Boolean sum and product operations as nonterminal nodes, and edges from each operation to its arguments. The root node of the POG is denoted $\mathbf{r}$. In the CPOG (for "Certified POG") framework, a proof consists of a sequence of equivalence-preserving clause addition and deletion steps.

The equivalence proof for standard knowledge compilation has two phases. The *forward implication* phase serves to prove $\phi \Rightarrow \psi$, i.e., that any assignment satisfying the input formula also satisfies the compiled formula. It starts with the clauses for ϕ, along with clauses providing a Tseitin encoding of the compiled formula ψ, denoted $\theta(X, Z)$, where Z is the set of auxiliary variables introduced by the encoding. The proof continues with a sequence of added clauses, each of which must satisfy the *reverse unit propagation* (RUP) property [25, 43] with respect to the existing clauses. This property guarantees that the added clause is logically implied by the earlier clauses. The forward implication phase terminates with the addition of a unit clause $[r]$, where r is the variable in Z denoting root node $\mathbf{r}$. Adding this clause proves that any assignment α that satisfies ϕ must cause ψ to evaluate to 1. This phase can have exponential complexity. For example, when the input formula is unsatisfiable, the forward implication proof must be able to certify unsatisfiability.

The reverse implication phase for standard knowledge compilation serves to prove $\psi \Rightarrow \phi$, completing the proof of equivalence. It does so by showing that each input clause satisfies the RUP property with respect to the combination of $\theta(X, Z)$ and unit clause $[r]$. That is, any assignment that falsifies the clause must cause ψ to evaluate to 0 (false). This clause can therefore be deleted without affecting the set of satisfying assignments. This portion of the proof is guaranteed to have polynomial complexity, since clausal entailment can be tested efficiently for d-DNNF formulas [9, 18].

For certifying projected knowledge compilation, we can build on the CPOG framework. The forward implication phase requires no change to the framework and only small changes to how the proofs are generated. Having the phase culminate with the addition of unit clause $[r]$ proves that any pair of assignments α to the data variables and β to the quantified variables that satisfies input formula ϕ will cause compiled formula ψ to evaluate to 1. This result holds even when ψ contains only data variables and therefore depends only on assignment α.

On the other hand, it is less clear how to generalize the reverse implication phase to handle projection. The input clauses contain both data and quantified variables, and so the simple strategy of showing that each individual input clause is implied by the compiled formula (containing only data variables) will not suffice. Instead, we handle the existential quantification via a form of Skolemization [3], associating some assignment β to the quantified variables for each satisfying assignment α to the data variables. We do so by extending the POG representation to include *Skolem nodes*, representing partial assignments to the quantified variables. This "Skolem POG" then represents two d-DNNF formulas:

- Interpreting the Skolem nodes as tautologies gives the compiled representation $\psi(D)$. This version is used in the forward implication phase.

- By interpreting the Skolem nodes as Boolean products, the formula assigns values to both data and quantified variables. This makes it possible to deduce a Skolem assignment β for any satisfying data assignment α. This version serves as the basis for the input clause deletion steps in the reverse implication phase.

The Skolem POG can be generated by a combination of minor modifications to the projected knowledge compiler and postprocessing to partition the Boolean products into product and Skolem nodes.

In a prototype implementation of a proof generator and checker for projected model compilation, we found that storing the clausal representation of a Skolem POG, plus the proof steps for a RUP derivation of the input clauses in the reverse implication phase, led to excessive file sizes for the proofs – the degree for many of the Skolem nodes can be in the thousands and the size of each RUP step is linear in the size of the Skolem POG. To reduce file sizes and to speed up proof generation and checking, we shifted responsibility for proving that the input clauses can be deleted from the proof generator to the proof checker. As a result of this shift, we can also have the checker make inferences directly on the Skolem POG, rather than on its clausal representation. Since the test for each input clause has linear complexity (in the size of the Skolem POG), this shift in responsibility still fulfills a requirement for propositional proof systems that the proof must be checkable in polynomial time, with respect to the size of the proof [14].

3 Overview and Prior Work

This paper describes how our earlier work on certifying the results of a standard knowledge compiler can be extended to those of a projected knowledge compiler. It provides background on the relevant logical foundations, the extension of the POG representation to include Skolem nodes, the structure of both the forward and the reverse implication phases, and a sketch of a soundness proof for the framework. We also discuss the use of HOL4 and CakeML to formally verify the soundness proof and to implement a formally verified proof checker.

We present experimental results for our implementation, consisting of modified versions of D4 and our earlier proof generation and checking tools. By shifting the reverse implication phase of the proof to the proof checker, our framework scales to formulas having large d-DNNF representations.

In terms of prior research, there has been some work on proof frameworks for model counting [5, 12, 23], along with comparisons of the expressive power of the different frameworks [4]. In addition, another proof framework has been developed for standard knowledge compilation [10]. We know of no previous work extending any of these frameworks to handle projection. Indeed, the literature on projected model counting and projected knowledge compilation is very sparse. Several algorithms and tools for projected knowledge compilation and model counting have been published, based on top-down approaches [1, 27, 31, 32, 35, 39], decision diagrams [20], and dynamic programming [22], but none of these can certify their results.

We have found that formal proof assistants are essential in uncovering the many subtle details that can arise in devising a sound proof framework. Working with a proof assistant also let us explore possible optimizations in the proof framework and checker with complete confidence of their soundness. In this work, we shifted from Lean 4 [19] to the HOL4 proof assistant [40]. We did so in part to take advantage of the CakeML programming language [29], which has a formally verified compiler [42] and a large body of support for developing verified machine-code implementations of propositional proof frameworks [24, 41].

Clause	File line	Formula	Comment
	`p cnf 4 3`		CNF formula with 4 variables and 3 clauses
	`c t pmc`		Projected model counting
	`c p show 1 2 0`	$D = \{a, b\}$	Declare data variables
1	`1 3 4 0`	$\overline{a} \Rightarrow (x \vee y)$	$\overline{a}$ triggers either x or y
2	`2 -3 0`	$x \Rightarrow b$	When x is triggered, force b to 1
3	`2 -4 0`	$y \Rightarrow b$	When y is triggered, force b to 1

Figure 1 CNF representation of disjunction of data variables a and b (numbered 1 and 2), with quantified variables x and y (numbered 3 and 4). Clause identifiers are implicitly assigned in the format and are shown here in red.

4 Logical Foundations

A Boolean formula ϕ over one or more disjoint sets of variables $X_1, X_2, \ldots, X_m$, is written as $\phi(X_1, X_2, \ldots, X_m)$ for clarity or as simply ϕ when the context is clear.

The *dependency set* for Boolean formula ϕ, denoted $\mathcal{V}(\phi)$, consists of all variables occurring in it. An *assignment* for a set of variables X is a mapping $\alpha\colon X \to \{0, 1\}$. When an assignment includes a value for every variable in $\mathcal{V}(\phi)$, it is said to be *total*; otherwise it is partial. We assume assignments are total unless explicitly stated. The set of all total assignments for X is denoted $\mathcal{U}(X)$. For formula ϕ with dependency set X, its set of *models*, denoted $\mathcal{M}(\phi)$ consists of all assignments in $\mathcal{U}(X)$ that satisfy the formula. For assignments $\alpha \in \mathcal{U}(X)$ and $\beta \in \mathcal{U}(Y)$, where $X \cap Y = \emptyset$, we write $\alpha{\cdot}\beta$ as the assignment in $\mathcal{U}(X \cup Y)$ assigning x to $\alpha(x)$ when $x \in X$ and to $\beta(x)$ when $x \in Y$. A *literal* is either a variable x or its complement $\overline{x}$. The symbol ℓ denotes a generic literal. Clauses are written as disjunctions, except that a unit clause, consisting of a single literal ℓ is written as $[\ell]$.

With a *projected formula* $\phi(D, Y)$ the variables are partitioned into a set of *data variables* D and a set of *quantified variables* Y. Its set of *data models*, denoted $\mathcal{M}_D(\phi)$ is defined as

$$\mathcal{M}_D(\phi) \;=\; \{\alpha \in \mathcal{U}(D) \mid \exists \beta \in \mathcal{U}(Y) \,.\, \alpha{\cdot}\beta \in \mathcal{M}(\phi)\} \tag{1}$$

The task of *projected knowledge compilation* is to convert a CNF formula $\phi(D, Y)$ into a formula $\psi(D)$ such that $\mathcal{M}(\psi) = \mathcal{M}_D(\phi)$. Furthermore, ψ should be in a form for which computing important properties, such as the number of models, can be performed in polynomial time, relative to the size of ψ.

The class of Boolean formulas known as *deterministic, decomposable, negation-normal form* (d-DNNF) is defined recursively as formulas ψ of the form:

Literals: x and $\overline{x}$ for Boolean variable x.

Decomposable Products [15, 18]:

$\psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_m$ where $\mathcal{V}(\psi_i) \cap \mathcal{V}(\psi_j) = \emptyset$ for $1 \leq i < j \leq m$

Deterministic Sums [16, 18]:

$\psi_1 \vee \psi_2 \vee \cdots \vee \psi_m$ where $\mathcal{M}(\psi_i) \cap \mathcal{M}(\psi_j) = \emptyset$ for $1 \leq i < j \leq m$. Here we assume for all $1 \leq i \leq m$, that $\mathcal{M}(\psi_i) \subseteq \mathcal{U}(X)$ for a common set of variables X.

We represent d-DNNF formulas as *partitioned-operation graphs* (POGs) [6, 7]. These are directed acyclic graphs where the terminal nodes represent literals and the nonterminal nodes represent product and sum operations. The root node is denoted $\mathbf{r}$.

As a running example throughout this paper, Figure 1 shows the declaration for a CNF formula encoding the disjunction of data variables a and b, with quantified variables x and y. Following the DIMACS CNF notation, literals are represented as integers. In this case,

we number the variables a and b as 1 and 2 and the variables x and y as 3 and 4. The two comment lines (beginning with 'c') indicate that the file encodes a projected model counting problem with variables 1 (a) and 2 (b) as the data variables. With clause 1, when a is assigned 0, either x or y must be assigned 1. With clauses 3 and 4, assigning 1 to either x or y will force b to be assigned 1.

5 Proof Requirements

Given a CNF formula $\phi(D, Y)$ and a d-DNNF formula $\psi(D)$, represented as a POG, we would like to generate a proof that $\mathcal{M}(\psi) = \mathcal{M}_D(\phi)$. We do so by converting ψ into clausal form as a CNF formula $\theta(D, Z)$, where Z is a set of *extension* variables used to encode the product and sum operations, according to a Tseitin encoding [38]. The *root variable* for the POG is defined as the extension variable r associated with root node $\mathbf{r}$. This encoding has the property that for any $\alpha \in \mathcal{U}(D)$, there is a unique assignment $\gamma \in \mathcal{U}(Z)$ such $\alpha{\cdot}\gamma \in \mathcal{M}(\theta)$. We refer to this assignment as γ_α. Furthermore, this assignment will satisfy $\gamma_\alpha(r) = 1$ if and only if $\alpha \in \mathcal{M}(\psi)$. We use the notation $\theta_r(D, Z)$ to represent the formula $\theta(D, Z) \cup \{[r]\}$, the combination of the Tseitin encoding of the POG and a unit clause with the root literal requiring the POG to evaluate to 1. Formula θ_r therefore serves as the clausal representation of d-DNNF formula ψ.

The proof of $\mathcal{M}(\psi) = \mathcal{M}_D(\phi)$ consists of two phases. The *forward implication* phase serves to prove that any data model α for $\phi(D, Y)$ will be a model for $\psi(D)$. With $\theta_r(D, Z)$ serving as the clausal representation of ψ, we can write this as

$$\forall\alpha\big[\exists\beta\,.\,\alpha{\cdot}\beta \in \mathcal{M}(\phi) \;\Rightarrow\; \exists\gamma\,.\,\alpha{\cdot}\gamma \in \mathcal{M}(\theta_r)\big] \tag{2}$$

We can transform this formula by bringing β outward and inverting the sense of the quantification. Furthermore, we can make use of the fact that γ is uniquely determined by α. These changes yield the formula

$$\forall\alpha\;\forall\beta\big[\alpha{\cdot}\beta \in \mathcal{M}(\phi) \;\Rightarrow\; \alpha{\cdot}\gamma_\alpha \in \mathcal{M}(\theta_r)\big] \tag{3}$$

We can see here that the assignments to both the data variables and the quantified variables are quantified in the same manner. We can therefore use the same method to generate and check this proof as is done with standard knowledge compilation [6, 7]. Starting with the combined clauses $\phi(D, Y)$ and $\theta(D, Z)$, the proof of (3) consists of a sequence of clause addition steps leading to the addition of the unit clause $[r]$. In doing so the proof transforms θ, the clausal representation of the POG operations, into θ_r, the clausal representation of the d-DNNF formula ψ.

The *reverse implication* phase serves to prove that, for any assignment α satisfying $\psi(D)$, there is a $\beta \in \mathcal{U}(Y)$ such that $\alpha{\cdot}\beta$ satisfies $\phi(D, Y)$. Working directly with the d-DNNF formula ψ, we can write this as:

$$\forall\alpha\big[\alpha \in \mathcal{M}(\psi) \;\Rightarrow\; \exists\beta\,.\,\alpha{\cdot}\beta \in \mathcal{M}(\phi)\big] \tag{4}$$

We can bring β outward, yielding

$$\forall\alpha\;\exists\beta\big[\alpha \in \mathcal{M}(\psi) \;\Rightarrow\; \alpha{\cdot}\beta \in \mathcal{M}(\phi)\big] \tag{5}$$

Here, the quantifications of α and β differ: they require the implication to hold for *all* data assignments α but for only *some* quantified assignment β for a given α – and even then only when α is a data model of ψ. The task of generating such an assignment is known as *Skolemization*, and β is referred to as a *Skolem assignment* for α [2, 3].

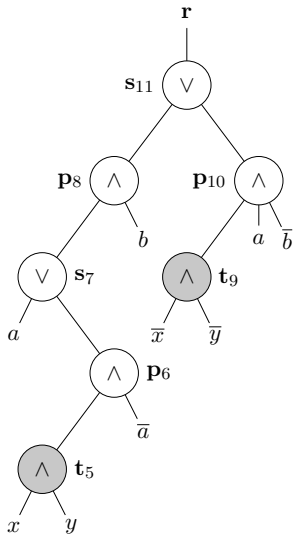


Figure 2 Skolem POG representing disjunction of data variables a and b, with quantified variables x and y. The Skolem nodes (shown in gray) define an assignment to the quantified variables for a given assignment to the data variables.

6 Skolem POGs

A Skolem POG is an extension of the POG representation that serves two roles:

Compiler Output: It provides the POG representation of a d-DNNF formula $\psi(D)$ forming the output of a projected knowledge compiler.

Skolemization: It augments this representation with a method for computing a Skolem assignment β for any assignment $\alpha \in \mathcal{M}(\psi)$ such that $\alpha \cdot \beta \in \mathcal{M}(\phi)$.

6.1 Skolem POG Example

Figure 2 illustrates a Skolem POG generated from the CNF formula $\phi(D, Y)$ of Figure 1 with $D = \{a, b\}$ and $Y = \{x, y\}$. As can be seen, the graph is itself a POG – every product operation is defined over disjoint sets of variables, and every sum operation is defined over disjoint sets of models. The gray nodes, known as *Skolem nodes*, indicate a special class of product operation. Skolem nodes only have literals of quantified variables as arguments, and they are the only nodes having such arguments.

The nodes shown in white comprise the d-DNNF generated by D4. We can see that D4 split on variable b at the top level, with node $\mathbf{p}_8$ representing the case where $\alpha(b) = 1$ and node $\mathbf{p}_{10}$ indicating the case where $\alpha(b) = 0$, with these being combined by the root node $\mathbf{s}_{11}$. On the left-hand side, we can see that $\mathbf{s}_7$ shows a split on variable a, resulting in the tautology $a \vee \overline{a}$. On the right-hand side, we see that node $\mathbf{p}_{10}$ requires $\alpha(a) = 1$ when $\alpha(b) = 0$ to satisfy the disjunction $a \vee b$.

The Skolem nodes provide Skolem assignments for some of the assignments to a and b. For $\alpha(b) = 1$ and $\alpha(a) = 0$, there are three possible assignments to variables x and y that satisfy the input clauses. Skolem node $\mathbf{t}_5$[3] indicates the choice $\beta(x) = \beta(y) = 1$. For $\alpha(b) = 0$

[3] The symbol "$\mathbf{t}$" is chosen to honor Thoralf Albert Skolem (1887–1963), the Norwegian logician for whom Skolemization is named.

and $\alpha(a) = 1$, there is a single satisfying assignment with $\beta(x) = \beta(y) = 0$, as indicated by Skolem node $\mathbf{t}_9$. We can see that no Skolem assignment is given for $\alpha(a) = \alpha(b) = 0$. This case is not needed, since it does not satisfy the input formula. We can also see that no Skolem assignment is given for $\alpha(a) = \alpha(b) = 1$. This is not required, since the data assignment alone guarantees that all input clauses are satisfied.

This example shows that, by considering or ignoring the Skolem nodes, a single graph can serve the dual roles of knowledge compiler output and Skolemization.

6.2 Clausal Encoding

In generating a clausal representation of a POG, we introduce an extension variable for each sum and product node. For a node with k children, we add $k + 1$ clauses, providing a Tseitin encoding of the operation over its arguments. Generating a clausal encoding of a Skolem POG is only required for the forward implication phase of the proof. We can therefore encode a Skolem node $\mathbf{t}$ as an extension variable t with a single unit clause $[t]$ indicating that the node should be interpreted as a tautology.

7 Proof Generation and Checking

Generating the Skolem POG and the forward implication proofs involves only modest changes to the knowledge compiler and to the proof generator. A more substantial change is required for the shift of the reverse implication phase to the proof checker.

7.1 Compiler Modification

We make use of the knowledge compiler D4, which already supports projected knowledge compilation. As with other top-down knowledge compilers [17, 36], D4 proceeds in a manner similar to a CDCL SAT solver. On each step, it performs unit propagations and simplifies the set of clauses by eliminating satisfied clauses and by removing falsified literals from the remaining clauses. When it selects an unassigned variable x, it must, in general, consider both assignments x and $\overline{x}$. These *splitting variables* become the basis for deterministic sums, with one argument satisfied only by assignments α with $\alpha(x) = 1$ and the other only with $\alpha(x) = 0$. The assigned and propagated literals then become arguments for decomposable product operations.

When running in projected compilation mode, D4 splits only on data variables. Once the simplified set of clauses contains only quantified variables, it calls a SAT solver to determine whether or not the formula is satisfiable. If so, it emits a tautology for the compiled result, and if not it emits a conflict. In addition, when unit propagation detects a unique assignment for a variable, it only includes the corresponding literal in the output for a data variable.

Modifying D4 to generate the information required to construct a Skolem POG involves two small changes:

1. After the SAT solver determines that a set of clauses containing only quantified variables is satisfiable, request a satisfying assignment. These become the arguments for a Skolem node. For example, node $\mathbf{t}_5$ in Figure 2 was generated after the assignment $\alpha(b) = 1$ and $\alpha(a) = 0$ reduced the input clauses in Figure 1 to the single clause $x \vee y$. The SAT solver detected that this clause was satisfiable and returned the assignment $\beta(x) = \beta(y) = 1$. This case only occurs as D4 reaches a terminal condition; it causes a Skolem node to be generated rather than a tautology.

2. Issue literals for both data and quantified variables when unit propagation occurs. These are then grouped into arguments for product and Skolem nodes. For example, assigning $\alpha(b) = 0$ to the clauses of Figure 1 first causes the unit propagations $\beta(x) = \beta(y) = 0$ and then $\alpha(a) = 1$. These assignments are grouped into nodes $\mathbf{t}_9$ and $\mathbf{p}_{10}$ in Figure 2. This case can occur at any point where unit propagation occurs; it can cause a Skolem node to be generated as the argument to a product node at upper levels in the POG.

Our modified version of D4 issues the additional literals as part of its regular output. We postprocess these into the required structure for a Skolem POG.

7.2 Proof Generation

The *monolithic* proof generation method devised for standard knowledge compilation [6] can be adapted for projected knowledge compilation. The generator runs a proof-generating SAT solver on the (hopefully) unsatisfiable CNF formula consisting of the clauses in formulas $\phi(D, Y)$ and $\theta(D, Z)$, as well as the unit clause $[\overline{r}]$. The solver effectively proves that the POG cannot generate value 0 as its output for any data assignment $\alpha \in \mathcal{M}_D(\phi)$. Each of the proof clauses is then augmented with the literal r, and so the empty clause of the proof becomes the unit clause $[r]$. This approach works regardless of any preprocessing transformations made by the knowledge compiler.

We also support a *hybrid* proof generation method, where the proof generator recursively traverses the structure of the POG, as with a structural proof generation method [7], but stopping once the size of a subgraph is sufficiently small. The proof for the subgraph is generated monolithically. This approach only works if the recursive traversal of the d-DNNF formula generated by the knowledge compiler yields corresponding simplifications and unit propagations of the CNF input formula. This may not hold if the CNF has been transformed by preprocessing.

7.3 Proof Checking

We have shifted the responsibility for the reverse implication phase to the proof checker. Letting $\hat{\psi}(D, Y)$ denote the d-DNNF formula encoded by the Skolem POG, the checker should ensure that $\hat{\psi}(D, Y) \Rightarrow \phi(D, Y)$. It can so by checking that for each input clause C, any assignment to the data and quantified variables that falsifies this clause will also cause $\hat{\psi}$ to evaluate to 0. It does this by performing a bottom-up unit propagation through the Skolem POG. Figure 3 demonstrates how this process works for the three input clauses from the formula of Figure 1. In each case, we can see that negating the literals of the clause causes some of the arguments to the Skolem POG operations to be assigned value 0. These are shown as dashed lines. Arguments that are assigned value 1, as well as unassigned arguments are shown as solid lines. A product or Skolem operation having an argument assigned value 0 will also evaluate to 0, and a sum operation will evaluate to 0 when both arguments evaluate to 0. The result in all three cases is to assign 0 to root variable r.

Section 8 presents a soundness proof for the forward and reverse implication phases.

7.4 SCPOG Example

Our SCPOG proof framework modifies and extends the earlier CPOG framework to enable certifying the results of a projected model counter. A single SCPOG file describes both a Skolem POG representation of a formula, as well as the forward implication proof. Here, we present the main features of the file format via an example. A more comprehensive presentation is included as Appendix A.

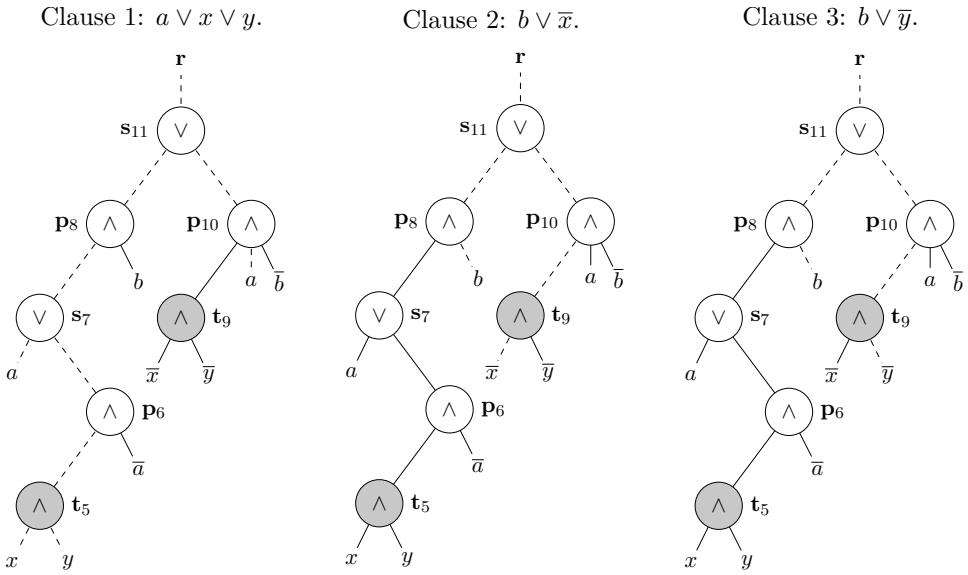


Figure 3 Justifying input clause deletions. Negating the clause literals and then unit propagating through the operations cause the Skolem POG to evaluate to 0. Literals assigned value 0 are shown as dashed lines.

Figure 4(A) shows the complete SCPOG file for the example with the CNF file shown in Figure 1 and the Skolem POG shown in Figure 2. The CNF has four variables and three clauses, and so it starts numbering extension variables with 5 and added clauses with ID 4. The first line declares the root variable r as the extension variable we denote as s_{11}. The next seven lines declare the Skolem POG operations, as shown in Figure 2. These implicitly generate the set of clauses shown in Figure 4(B). Skolem nodes $\mathbf{t}_5$ and $\mathbf{t}_9$ require just single (unit) clauses, while the other node declarations implicitly add either three or four clauses, and so the clause identifiers in the declarations must be spaced apart.

Every sum node must include a mutual exclusion proof to certify determinism. Each proof is given as list of clauses providing a RUP proof that the two arguments cannot both evaluate to 1. In the example, the declaration of sum node $\mathbf{s}_7$ lists clause 6 ($\overline{p}_6 \vee \overline{a}$) as proof that arguments p_6 and a cannot both evaluate to 1. The declaration of sum node $\mathbf{s}_{11}$ references clauses 12 ($\overline{p}_8 \vee b$) and 17 ($\overline{p}_{10} \vee \overline{b}$) for the mutual exclusion proof. These resolve to the clause $\overline{p}_8 \vee \overline{p}_{10}$, proving that arguments p_8 and p_{10} cannot both evaluate to 1.

Clause IDs 22–28 show the forward implication proof leading to the addition of unit clause $[s_{11}]$ as the final step.

Overall, we can see that this file contains a full documentation of the d-DNNF formula generated by the knowledge compiler, the Skolem assignments to the quantified variables associated with the assignments to the data variables, and a forward implication proof.

In addition to checking the mutual exclusion and forward-implication proofs, the proof checker must ensure that the arguments to each product operation are decomposable, and it must check that reverse implication holds.

8 Soundness

We have formally verified the soundness of our proof framework in HOL4 and developed a verified proof checker using CakeML. Here, we provide an informal justification of the proof framework to give some intuition for its soundness. Section 8.2 provides more details on the formally verified checker.

(A) SCPOG Declaration.

ID	File line	Formula
Root declaration		
	`r 11`	$r = s_{11}$
Skolem POG declaration		
4	`t 5 3 4 0`	$t_5 = x \wedge y$
5	`p 6 -1 5 0`	$p_6 = \overline{a} \wedge t_5$
8	`s 7 6 1 6 0`	$s_7 = p_6 \wedge a$
11	`p 8 2 7 0`	$p_8 = b \wedge s_7$
14	`t 9 -3 -4 0`	$t_9 = \overline{x} \wedge \overline{y}$
15	`p 10 1 -2 9 0`	$p_{10} = a \wedge \overline{b} \wedge t_9$
19	`s 11 8 10 12 17 0`	$s_{11} = p_8 \vee p_{10}$
Forward implication proof		
22	`6 -2 1 0 4 5 0`	$b \wedge \overline{a} \Rightarrow p_6$
23	`7 -2 0 9 22 10 0`	$b \Rightarrow s_7$
24	`8 -2 0 23 11 0`	$b \Rightarrow p_8$
25	`1 2 0 3 2 1 0`	$a \vee b$
26	`10 2 0 25 14 15 0`	$\overline{b} \Rightarrow p_{10}$
27	`-2 11 0 20 24 0`	$b \Rightarrow s_{11}$
28	`11 0 27 21 26 0`	$[s_{11}]$

(B) Implicit Definitions.

ID	Clauses	Node
4	`5 0`	$\mathbf{t}_5$
5	`6 1 -5 0`	$\mathbf{p}_6$
6	`-6 -1 0`	
7	`-6 5 0`	
8	`-7 6 1 0`	$\mathbf{s}_7$
9	`7 -6 0`	
10	`7 -1 0`	
11	`8 -2 -7 0`	$\mathbf{p}_8$
12	`-8 2 0`	
13	`-8 7 0`	
14	`9 0`	$\mathbf{t}_9$
15	`10 -1 2 -9 0`	$\mathbf{p}_{10}$
16	`-10 1 0`	
17	`-10 -2 0`	
18	`-10 9 0`	
19	`-11 8 10 0`	$\mathbf{s}_{11}$
20	`11 -8 0`	
21	`11 -10 0`	

Figure 4 SCPOG Example. A single file (A) declares the Skolem POG structure and provides the forward implication proof. The declaration of the Skolem POG implicitly defines a set of clauses (B). Clause IDs are shown in red.

8.1 Informal Justification

As a recap of notation, the POG has two forms: $\psi(D)$, including the sum and product nodes, with the Skolem nodes treated as tautologies, and $\hat{\psi}(D, Y)$, where the Skolem nodes are treated as Boolean products of their arguments (which are literals of quantified variables). The operations in POG formula $\psi(D)$ have Tseitin clausal representation $\theta(D, Z)$, and when we add unit clause $[r]$, we obtain $\theta_r(D, Z)$, the clausal representation of $\psi(D)$.

As further notation, we require a way to describe an implication relation between formulas over different sets of variables, considering only some of the variables they have in common. For formulas $\phi_1(S, T_1)$ and $\phi_2(S, T_2)$, define the relation $\phi_1 \Rightarrow_S \phi_2$ as holding when $\mathcal{M}_S(\phi_1) \subseteq \mathcal{M}_S(\phi_2)$. This notation makes no assumption about whether or not sets T_1 and T_2 are disjoint. We can see that for formulas $\phi_1(X)$ and $\phi_2(X)$, the standard implication $\phi_1 \Rightarrow \phi_2$ is equivalent to $\phi_1 \Rightarrow_X \phi_2$.

The following propositions state two key properties of this relation: narrowing the set of variables preserves implication, and dropping conjuncts in the consequent preserves implication. Note that in conjunctive normal form, the conjunction of two formulas can be expressed as the union of their clauses.

▶ **Proposition 1.** *For formulas $\phi_1(S, T_1)$, $\phi_2(S, T_2)$, if $\phi_1 \Rightarrow_S \phi_2$, then any subset $S' \subseteq S$ maintains the implication $\phi_1 \Rightarrow_{S'} \phi_2$.*

▶ **Proposition 2.** *For formulas $\phi_1(S,T_1)$, $\phi_2(S,T_2)$, and $\phi_3(S,T_3)$, if $\phi_1 \Rightarrow_S \phi_2 \wedge \phi_3$, then $\phi_1 \Rightarrow_S \phi_2$.*

The forward implication proof must show that $\phi(D,Y) \Rightarrow_D \theta_r(D,Z)$, fulfilling the requirement for the forward implication property (2). It begins with the clauses $\theta^1(D,Y,Z) \doteq \phi(D,Y) \cup \theta(D,Z)$, the combination of the input formula and the Tseitin encoding of the POG operations.[4] Since the union of two sets of clauses is equivalent to their conjunction, $\theta^1 = \phi \wedge \theta$. Furthermore, $\theta(D,Z)$ induces an assignment $\gamma_\alpha \in \mathcal{U}(Z)$ for every $\alpha \in \mathcal{U}(D)$, such that $\alpha \cdot \gamma_\alpha$ satisfies θ. Therefore appending θ to ϕ does not reduce the set of data models. That is, $\phi(D,Y) \Rightarrow_{D\cup Y} \theta^1(D,Y,Z)$, and therefore by Proposition 1, $\phi(D,Y) \Rightarrow_D \theta^1(D,Y,Z)$.

The RUP addition steps define a sequence of formulas $\theta^1, \theta^2, \ldots, \theta^s$ concluding with $\theta^s(D,Y,Z) \supseteq \phi(D,Y) \cup \theta(D,Z) \cup \{[r]\} \equiv \phi(D,Y) \wedge \theta_r(D,Z)$. Each proof step preserves implication: $\theta^i \Rightarrow_D \theta^{i+1}$ for $1 \leq i < s$, and therefore $\theta^1 \Rightarrow_D \theta^s$. By Proposition 2, we can see that these steps provide a proof that $\phi(D,Y) \Rightarrow_D \theta_r(D,Z)$, and therefore $\phi(D,Y) \Rightarrow_D \psi(D)$.

The reverse implication phase involves making sure that $\hat{\psi}(D,Y) \Rightarrow_D \phi(D,Y)$, fulfilling property (4). The proof checker does so by showing that $\neg C \Rightarrow \neg\hat{\psi}$, and therefore $\hat{\psi} \Rightarrow C$ for each clause $C \in \phi$. Combining all of these checks proves that $\hat{\psi}(D,Y) \Rightarrow \phi(D,Y)$, and therefore $\hat{\psi}(D,Y) \Rightarrow_D \phi(D,Y)$.

The combination of the forward and reverse implications shows that the two formulas for the Skolem POG bracket the input formula:

$$\hat{\psi}(D,Y) \;\Rightarrow_D\; \phi(D,Y) \;\Rightarrow_D\; \psi(D)$$

In terms of the data models, we can see therefore that $\hat{\psi}$ provides a lower bound, while ψ provides an upper bound: $\mathcal{M}_D(\hat{\psi}) \subseteq \mathcal{M}_D(\phi) \subseteq \mathcal{M}_D(\psi)$.

Importantly, we can also see that the two formulas for the Skolem POG are equivalent in terms of their data models:

▶ **Proposition 3.** *A Skolem POG with representations $\hat{\psi}(D,Y)$ and $\psi(D)$ has $\mathcal{M}_D(\hat{\psi}) = \mathcal{M}_D(\psi)$.*

This equivalence is proved by induction over the POG structure. The key observation is that the decomposability of each product operation makes it possible to derive the Skolem assignments for its arguments independently. Our bracketing is therefore an equivalence: $\mathcal{M}_D(\hat{\psi}) = \mathcal{M}_D(\phi) = \mathcal{M}_D(\psi)$, completing the proof that the POG formula $\psi(D)$ serves as the projected compilation of input formula $\phi(D,Y)$.

8.2 End-to-End Verification

We have formally verified the proof framework and used a refinement-based process to develop a formally verified CakeML implementation of the SCPOG proof checker in HOL4. This is broadly similar to the development of verified machine-code implementations of prior propositional proof frameworks [24, 41].

Checking proofs in these other frameworks only requires checking a sequence of clause additions. The SCPOG proof format also requires checking the decomposability and determinism of the specified POG and to check each input clause for the reverse implication. The CakeML checker starts with a clausal representation of the SCPOG nodes when checking the forward

[4] In our proof format specification (Appendix A), SCPOG proof steps are allowed to arbitrarily interleave RUP additions and node declaration steps; the soundness argument is similar.

implication proof. Upon success of this phase, it internally converts this representation into an equivalent adjacency matrix-like representation where all nodes and their children can be looked up in constant time. For each input clause C, this allows the CakeML checker to run a fast, top-down evaluation from root node $\mathbf{r}$ to check that the POG evaluates to 0 under the assignment corresponding to $\neg C$; the checker short-circuits evaluation wherever possible (e.g., when a child of a product node is falsified, it does not recurse into other children). The constant-time lookup representation is particularly important for Skolem nodes, which can contain a large number of quantified literals. For such nodes, we iterate over the literals in C and check if any of them are in the Skolem node.

The final correctness theorem for the CakeML proof checker is obtained by compiling its verified source implementation with CakeML's formally verified compiler [42]. Informally, the resulting verified theorem states that:

▶ **Proposition 4.** *Assuming the SCPOG checker's machine code is executed on a system satisfying the standard CakeML assumptions (for the x64 ISA), then:*

- *if it prints out* `"s VERIFIED UNSAT"` *on standard output, then the input CNF file parsed to an unsatisfiable CNF formula;*
- *if it prints out* `"s VERIFIED CPOG REPRESENTATION"` *on standard output, then the input CNF file parsed to a CNF formula and the input proof file contains a POG such that the CNF and POG have the same set of models on the specified data variables; moreover, the CPOG indeed represents a d-DNNF, i.e., it is decomposable and deterministic; and*
- *all proof checking failures lead to error messages printed to standard error.*

9 Experimental Results

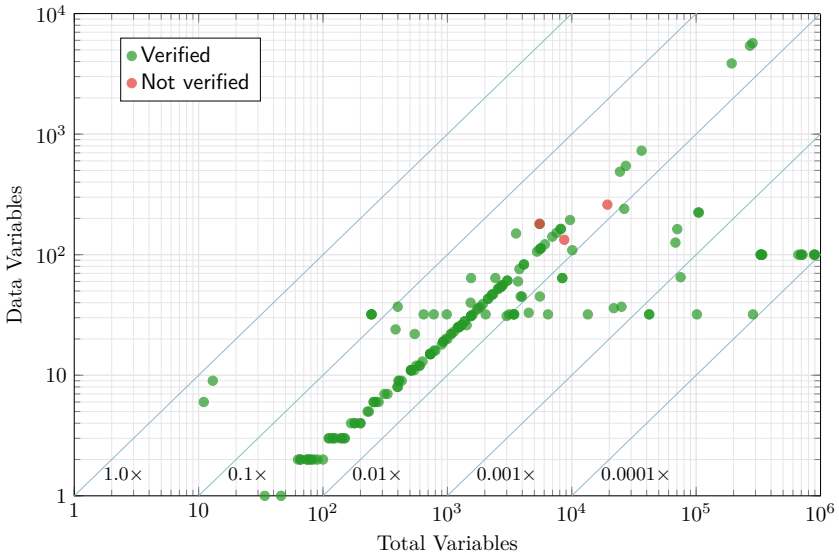


Figure 5 Data Variables vs. Total Variables. The median fraction was 2.0%. Colors indicate the level of success achieved in verification.

We implemented a proof generator in C++, along with two proof checkers: one written in C, and the other in CakeML and verified with HOL4. The C checker uses multithreading when performing the reverse-implication check, taking advantage of the property that the input clause deletion checks can be performed independently.

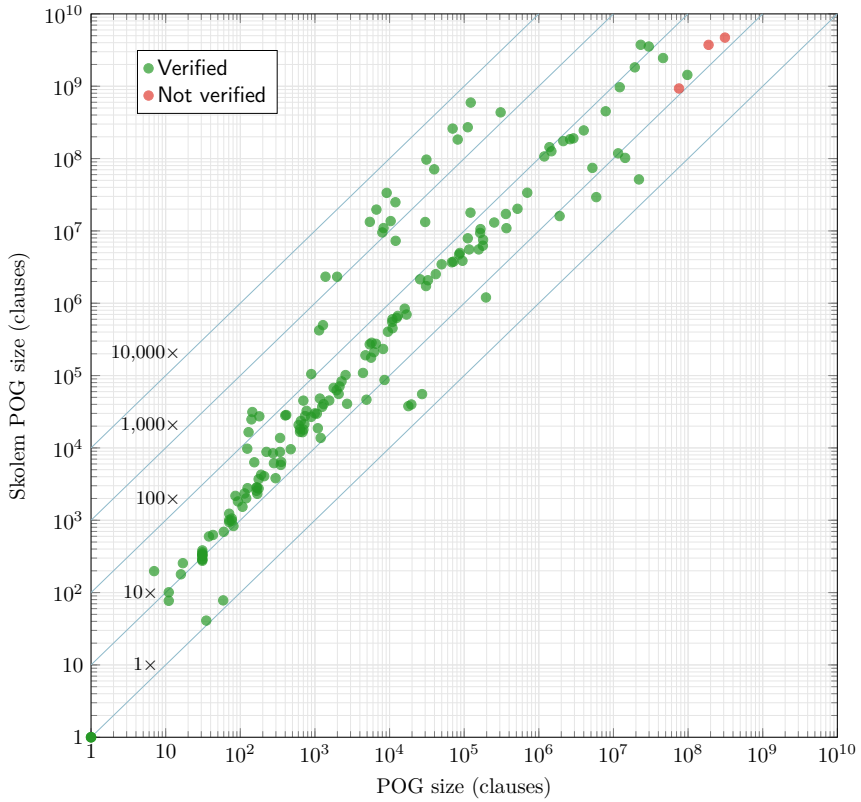


Figure 6 Skolem POG overhead. Skolem POG size vs. POG size. The median size ratio was 34×. Colors indicate the level of success achieved in verification.

All experiments were run on a 2021 Apple MacBook Pro, with a 3.2 Ghz Apple M1 8-core processor and 64 GB of RAM. All reported times are for elapsed ("wall-clock") times.

For benchmark problems, we used the private and public formulas from the 2024 unweighted and weighted projected model counting competitions.[5] There were 200 formulas for each track, but with some duplicates across the two tracks, yielding a set of 389 unique benchmark formulas.

We ran two versions of D4 on the 389 benchmark formulas: an unmodified version generating just the d-DNNF data, and the modified version that included the information from which the Skolem POG could be constructed. Both successfully compiled 175 formulas within the time limit of 1,000 seconds. The modified version of D4 ran fast enough: at most 1.8× slower than the unmodified version, with a median of 1.11×.

Figure 5 documents an important feature of the 175 compiled benchmark formulas, showing the number of data variables (Y-axis), compared to the total number of variables (X-axis). For many formulas, the data variables constitute only a small fraction of the total. For one, only 100 of its 906,806 (0.0001%) variables are data variables. The median ratio of data variables to total variables is 2.0%. This property has significant impact on the sizes of the Skolem POGs and the challenge of checking reverse implication. The color coding of the data in the plots of Figure 5 and 6 gives a preview of our success in verifying these benchmarks: 172 of them (green) completed the full verification. For the remaining 3 (red), the proof generator timed out, with a time limit of 10,000 seconds.

[5] Available at `https://zenodo.org/records/14249068`.

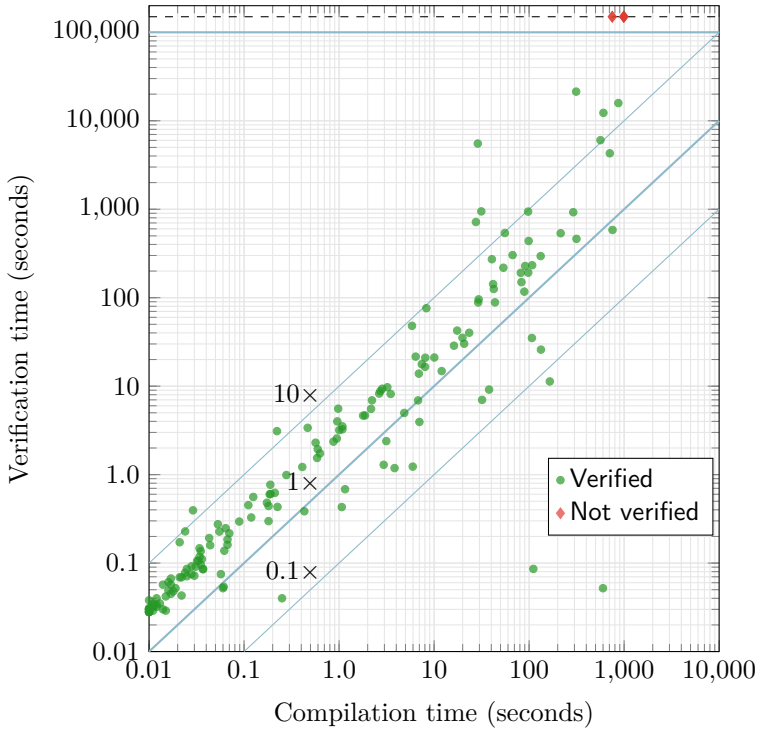


Figure 7 Verification vs. compilation time. Verification includes proof generation and checking. The reverse implication check uses 7-way multithreading.

We generated the Skolem POGs from the output of the modified version of D4, and we could also determine how large the POG would be without the Skolem information, as illustrated on the Y and X axes of Figure 6, respectively. We measure POG sizes in terms of the number of clauses in their Tseitin encodings. This is also equal to the number of nodes plus the number of edges in the graphs. As can be seen, the Skolem POGs tend to me much larger than the regular POGs, with size ratios ranging up to 4857× with a median of 34×. Considering the large fraction of quantified variables in these formulas, these large size ratios are to be expected.

Figure 7 compares the runtime for verification versus for compilation for the 172 formulas that were fully verified. The X axis shows the time to run the modified version of D4, while the Y axis shows the time to postprocess the D4 output, generate and check the proof, and compute an unweighted or weighted model count. The reverse-implication check uses 7-way multithreading. For 19 formulas, compilation took longer than verification, while it required less for the others. At worst, verification required 191.7× longer, with a median of 2.95×.

Our use of multithreading provides a significant benefit. Verifying all 172 benchmarks requires a total of 20.8 hours with the multithreaded version and 37.9 hours with a sequential version. This includes proof generation, forward and reverse checking, and model counting. Parallelism is only used when performing the reverse-implication check on the input clauses. This portion requires 8.4 hours with multithreading and 25.5 hours sequentially, a speedup of 3.0×. This is a reasonable performance gain for a memory-intensive application. Multithreading also reduces the benchmark with the longest sequential verification time from 15.3 to 4.4 hours.

We ran the formally verified checker on the 172 formulas for which the proof generator completed. It verified 156 of them, and it exceeded its heap memory limit of 48 GB for the remaining 16. Comparing its performance to the unverified checker for the 156 completed checks, it ran between 2.9× faster and 17.7× slower, with a median of 1.5× slower. This is remarkable performance for a formally verified tool.

Overall, our tool handled these formulas well. On the other hand, we found that letting D4 run for longer than 1000 seconds leads to d-DNNF formulas that are beyond the capacity of our current tool chain. The major weakness is in generating forward implication proofs.

10 Conclusions and Further Research

We have shown that the output of a projected knowledge compiler can be certified by extending our prior proof framework for standard knowledge compilation. The compiler already has access to information about how the quantified variables can be assigned according to the assignment to the data variables. By having it provide this information and then postprocessing the output, the proof generator can construct a Skolem POG, serving the dual roles of encoding the projected formula and the Skolem assignments. Certification proceeds by proving implication in both directions, within forward implication proved by clausal additions and reverse implication checked directly on the generated formula.

We were able to formally verify the soundness of our proof framework in HOL4 and develop a formally verified checker. A successful run of the checker gives full confidence that the compiled formula $\psi(D)$ has the same data models as the input formula $\theta(D, Y)$. In our earlier effort to devise a proof framework for standard knowledge compilation, the formal verification effort uncovered some subtle requirements that we had overlooked. Our experience was quite different this time. We correctly anticipated the requirements to achieve soundness. On the other hand, having the support of formal verification enabled us to be more aggressive in exploring refinements to improve the combined proof generator and checker performance. These optimizations greatly improved the scaling of our tools.

Projected knowledge compilation, and the closely related topic of projected model counting, are active areas of research, with many optimizations being discovered. Our framework will only be of value if it can keep up with these innovations. We describe two examples here.

First, some have discovered conditions under which it is possible for the compiler or model counter to split on quantified variables, remove the literals for these variable in the compiled output, and still have the result be a d-DNNF formula [27]. This requires more extensive mutual exclusion proofs for the sum operations than the simple cases illustrated in the SCPOG example of Figure 4. Our framework, described in Appendix A, includes support for such proofs. For a sum node with children represented by literals ℓ_1 and ℓ_2, the mutual exclusion proof can be generated with a proof-generating SAT solver that includes clauses encoding the two branches, plus the unit literals $[\ell_1]$ and $[\ell_2]$. The proof clauses in the UNSAT proof are augmented by the literals $\overline{\ell}_1$ and $\overline{\ell}_2$, and so the empty clause of the proof becomes the mutual exclusion $\overline{\ell}_1 \vee \overline{\ell}_2$.

Second, some have discovered cases where input clauses can be removed via *blocked-clause elimination* [32]. Significantly, they do so as an inprocessing step, where the conditions for the elimination hold only under a partial assignment to the variables. Our framework can handle the special case of *pure-literal elimination*, where the blocked clause has been reduced to a unit literal, by having the knowledge compiler record the literal as being satisfied. Handling more general cases would seem to require a new way to prove reverse implication for those input clauses that have been eliminated.

References

1 Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. #∃SAT: Projected model counting. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 of *LNCS*, pages 121–137, 2015. doi:10.1007/978-3-319-24318-4_10.

2 Valeriy Balabanov and Jie-Hong Roland Jiang. Unified QBF certification and its applications. *Formal Methods in Systems Design*, 41:45–65, 2012. doi:10.1007/S10703-012-0152-6.

3 Marco Benedetti. Evaluating QBFs via symbolic Skolemization. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 3452 of *LNCS*, pages 285–300, 2005. doi:10.1007/978-3-540-32275-7_20.

4 Olaf Beyersdorff, Johannes K. Fichte, Markus Hecher, Tim Hoffmann, and Kasche Kaspar. The relative strength of #SAT proof systems. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl, July 2024.

5 Olaf Beyersdorff, Tim Hoffman, and Luc Nicolas Spachmann. Proof complexity of propositional model counting. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl, July 2023.

6 Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Certified knowledge compilation with application to verified model counting. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl, July 2023.

7 Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Certified knowledge compilation with application to formally verified model counting. *Journal of Artificial Intelligence Research*, 82:2057–2099, 2025. doi:10.1613/JAIR.1.15958.

8 Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992. doi:10.1016/0890-5401(92)90017-A.

9 Florent Capelli. Knowledge compilation languages as proof systems. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 11628 of *LNCS*, pages 90–91, 2019.

10 Florent Capelli, Jean-Marie Lagniez, and Pierre Marquis. Certifying top-down decision-DNNF compilers. In *AAAI Conference on Artificial Intelligence*, 2021.

11 Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172:772–799, 2008. doi:10.1016/J.ARTINT.2007.11.002.

12 Sravanthi Chede, Leroy N. Chew, and Anil Shukla. Circuits, proofs, and propositional model counting. In *Foundations of Software Technology and Theoretical Computer Science (FTTCS)*, 2024.

13 Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *Symposium on Logic in Computer Science (LICS)*, pages 353–362. IEEE, 1989. doi:10.1109/LICS.1989.39190.

14 Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, 2014.

15 Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001. doi:10.1145/502090.502091.

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

17 Adnan Darwiche. New advances in compiling CNF to decomposable negation normal form. In *European Conference on Artificial Intelligence*, pages 328–332, 2004.

18 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17, 2002. doi:10.1613/JAIR.989.

19 Leonardo de Moura and Sebastian Ulrich. The Lean 4 theorem prover and programming language. In *Conference on Automated Deduction (CADE)*, volume 12699 of *LNAI*, pages 625–635, 2021. doi:10.1007/978-3-030-79876-5_37.

20 Jeffrey M. Dudek, Vu H. N. Phan, and Moshe Y. Vardi. ProCount: Weighted projected model counting with graded project-join trees. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 12831 of *LNCS*, pages 152–170, 2021. doi:10.1007/978-3-030-80223-3_11.

21 Thomas Eiter and Gabriele Kern-Isberner. A brief survey of forgetting from a knowledge representation and reasoning perspective. *Künstliche Intelligenz*, 33:9–33, 2019. doi:10.1007/S13218-018-0564-6.

22 Johannes K. Fichte, Markus Hecher, Michael Morak, Patrick Thier, and Stefan Woltran. Solving projected model counting by utilizing treewidth and its limits. *Artificial Intelligence*, 314, 2023. doi:10.1016/J.ARTINT.2022.103810.

23 Johannes K Fichte, Markus Hecher, and Valentin Roland. Proofs for propositional model counting. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl, 2022.

24 Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving. In *AAAI Conference on Artificial Intelligence*, pages 8038–8047, 2024. doi:10.1609/AAAI.V38I8.28642.

25 Evgueni I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe (DATE)*, pages 886–891, 2003.

26 Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In *Handbook of Satisfiability*, pages 633–654. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-633.

27 Rafael Kiesel and Thomas Eiter. Knowledge compilation and more with SharpSAT-TD. In *Principles of Knowledge Representation and Reasoning*, pages 406–416, 2023. doi:10.24963/KR.2023/40.

28 Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *Journal of Applied Logic*, 22:46–62, July 2017. doi:10.1016/J.JAL.2016.11.031.

29 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 179–192. ACM, 2014. doi:10.1145/2535838.2535841.

30 Jean-Marie Lagniez and Pierre Marquis. An improved decision-DNNF compiler. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 667–673, 2017. doi:10.24963/IJCAI.2017/93.

31 Jean-Marie Lagniez and Pierre Marquis. A recursive algorithm for projected model counting. In *AAAI Conference on Artificial Intelligence*, pages 1536–1543, 2019. doi:10.1609/AAAI.V33I01.33011536.

32 Jean-Marie Lagniez, Pierre Marquis, and Armin Biere. Dynamic blocked clause elimination for projected model counting. In *Theory and Applications of Satisfiability Testing (SAT)*, 2024.

33 Fangzhen Lin and Ray Reiter. Forget it! In *AAAI Fall Symposium on Relevance*, pages 154–159, 1994.

34 Pierre Marquis. Knowledge compilation using theory prime implicates. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 837–843, 1995.

35 Sybylle Möhle and Armin Biere. Dualizing projected model counting. In *IEEE Conference on Tools with Artificial Intelligence*, pages 702–709, 2018.

36 Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric Hsu. DSHARP: Fast d-DNNF compilation with sharpSAT. In *Canadian Conference on Artificial Intelligence*, volume 7310 of *LNCS*, pages 356–361, 2012. doi:10.1007/978-3-642-30353-1_36.

37 David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986. doi:10.1016/S0747-7171(86)80028-1.

38 Steven Prestwich. CNF encodings. In *Handbook of Satisfiability*, pages 75–97. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-75.

39 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1169–1176, 2019. doi:10.24963/IJCAI.2019/163.

40 Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.

41 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. Verified progagation redundancy and compositional UNSAT checking in CakeML. *International Journal of Software Tools for Technology Transfer*, 25:167–184, 2023. doi:10.1007/S10009-022-00690-Y.

42 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019. doi:10.1017/S0956796818000229.

43 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *Proc. of the 10th Int. Symposium on Artificial Intelligence and Mathematics (ISAIM 2008)*, 2008.

A The SCPOG Format

This appendix documents the SCPOG proof format. A single file declares the structure of the Skolem POG and provides the required proof steps. The file consists of a sequence of steps that either implicitly or explicitly add clauses. The clauses in the input CNF file are not in the SCPOG file, but they serve as part of the proofs. Overall, the clauses support two different proof types: the *mutual exclusion* proofs for the sum operations, guaranteeing that these operations are deterministic, and the *forward implication* proof that the generated POG is implied by the quantified input formula.

The variables are partitioned into three disjoint classes:

Data: The variables declared as "show" variables in the input CNF file

Quantified: All other variables occurring in the input CNF file

Extension: The variables associated with the POG Sum, Product, and Skolem operations

The clauses are also partitioned into disjoint classes. This categorization is determined by how they are generated and how they are used.

Input: The clauses in the input CNF file

Tseitin: The clauses implicitly defined by the Sum, Product, and Skolem operations

Structural: Clauses to express the mutual exclusion proofs for the Sum operations

Forward: Clauses added as steps in the forward implication proof

Table 1 SCPOG Clause Types. Different types have different rules for generating them.

	Clause Addition		
Type	How Generated	Variables	Hints
Input	Input formula	Data, Quantified	None
Tseitin	Sum and Product operations	Data, Extension	None
Tseitin	Skolem operations	Extension	None
Structural	RUP addition	Data, Extension	Tseitin, Structural
Forward	RUP addition	Data, Extension	Input, Tseitin, Structural

Table 1 provides more detail about the different clause types, and how these clauses are generated. According to its type, each clause has restrictions on how it is generated, what variables it contains, and what clauses can serve in the hints to support their addition.

- *Input* clauses contain only data and quantified variables. Their addition is implicit.
- *Tseitin* clauses encode the sum, product, and Skolem operations, each defining an extension variable in terms of data and other extension variables. Each Skolem operation is encoded by a single unit clause.

Table 2 SCPOG Step Types. C: clause identifier, L: literal, V: variable

Rule				Description
	`r`	V		Declare root variable
	`r`	`0`		Declare unsatisfiable formula
C	`a`	L^* `0`	C^+ `0`	Add Forward clause
C	`as`	L^* `0`	C^+ `0`	Add Structural clause
C	`p`	V L^* `0`		Declare product operation
C	`s`	V L L	C^+ `0`	Declare sum operation
C	`t`	V L^* `0`		Declare Skolem operation

- *Structural* clauses support the mutual exclusion proofs for the sum operations. They are generated via RUP addition starting with Tseitin clauses. They contain only data and extension variables. These clauses are only needed when the POG contains sum operations that are not decision nodes.[6]
- Forward clauses form the steps in the forward implication proof. They are added as RUP steps, starting with Input, Tseitin, and Structural clauses. They can contain only data and extension variables. The root clause is the final clause in the forward-implication proof. It consists of the unit clause $[r]$ for root variable r.

As can be seen, there are some ambiguities in how a clause should be classified. For example, a clause containing only data and extension variables and having only Tseitin and Structural clauses as hints could be a Structural or a Forward clause. We therefore have a distinct command for adding Structural clauses.

A.1 Syntax

Table 2 shows the declarations that can occur in a SCPOG file. The checker is provided with the input formula as a separate file. As with other clausal proof formats, a variable is represented by a positive integer v, with the first ones being input variables, and higher ones serving as extension variables. Literal ℓ is represented by a signed integer, with $-v$ being the logical negation of variable v. Each clause is indicated by a positive integer identifier C, with the first ones being the IDs of the input clauses and successive ones being the IDs of added clauses. Clause identifiers must be defined in order, with any clause identifier C' given in the hint when adding clause C having $C' < C$.

The `r` command declares the root of the POG. For a satisfiable formula, it declares the root as a variable $r > 0$, indicating an extension variable representing the root of a graph. It must be declared before the root clause is added. For an unsatisfiable formula, the root declaration has value 0, and there is no root clause.

The second set of proof rules allows different clause types to be added. Forward clauses are added via RUP addition (command `a`), with a sequence of antecedent clauses (the "hint"). Structural clauses are added similarly, but with command `as`.

The final three step types declare the product, sum, and Skolem operations. The clauses implicitly generated by these operations are shown in Table 3.

[6] A sum operation is a decision node when, for some variable x, the operation has one argument that is either the literal x or a product operation having x as an argument, while the other child is either the literal $\overline{x}$, or it is a product operation containing having literal $\overline{x}$ as an argument.

Table 3 Defining Clauses for Product (A) and Sum (B) Operations, and Skolem (C) Operations.

(A). Product Operation

ID	Clause				
i	v	$-\ell_1$	$-\ell_2$	$\cdots$	$-\ell_m$
$i{+}1$	$-v$	ℓ_1			
$i{+}2$	$-v$	ℓ_2			
$\cdots$					
$i{+}m$	$-v$	ℓ_k			

(B). Sum Operation

ID	Clause		
i	$-v$	ℓ_1	ℓ_2
$i{+}1$	v	$-\ell_1$	
$i{+}2$	v	$-\ell_2$	

(C). Skolem Operation

ID	Clause
i	v

The declaration of a *product* operation has the form:

$$i \quad \texttt{p} \quad v \quad \ell_1 \quad \ell_2 \quad \cdots \quad \ell_m \quad \texttt{0}$$

Integer i is a new clause ID, v is a positive integer that does not correspond to any previous variable, and $\ell_1, \ell_2, \ldots, \ell_m$ is a sequence of m integers, indicating the arguments as literals of existing variables. The variables associated with these literals must be data variables or previously defined extension variables. To guarantee negation-normal form, any argument that represents some other operation must be positive. This declaration implicitly causes $m + 1$ Tseitin clauses to be added to the proof, providing a Tseitin encoding that defines extension variable v as the product of its arguments. These clauses are shown in Table 3(A).

To guarantee decomposability, the dependency sets for the arguments represented by each pair of literals ℓ_i and ℓ_j must be disjoint, for $1 \leq i < j \leq k$. The dependency set for the operation then consists of the union of the dependency sets for its arguments. A product operation may have no arguments, representing Boolean constant 1. The only clause added to the proof will be the unit literal v.

The declaration of a *sum* operation has the form:

$$i \quad \texttt{s} \quad v \quad \ell_1 \quad \ell_2 \quad H \quad \texttt{0}$$

Integer i is a new clause ID, v is a positive integer that does not correspond to any previous variable, and ℓ_1 and ℓ_2 are signed integers, indicating the arguments as literals of existing variables. The variables associated with these literals must be data variables or previously defined extension variables. To guarantee negation-normal form, any argument that represents some other operation must be positive. Hint H consists of a sequence of clause IDs, all of which must be structural clauses.[7] The dependency set for the operation consists of the union of the dependency sets for its arguments. As Table 3(B) shows, this declaration implicitly causes three Tseitin clauses to be added to the proof, providing a Tseitin encoding that defines extension variable v as the sum of its arguments. The hint must provide a RUP proof of the clause $\overline{\ell}_1 \vee \overline{\ell}_2$, showing that the two arguments have disjoint models.

The declaration of a *Skolem* operation has the same form as a product operation:

$$i \quad \texttt{t} \quad v \quad \ell_1 \quad \ell_2 \quad \cdots \quad \ell_m \quad \texttt{0}$$

[7] The restriction to structural clauses in the hint is critical to soundness. Allowing the hint to include the IDs of input clauses creates an exploitable weakness. We discovered this weakness in the course of our earlier efforts at formal verification.

Integer i is a new clause ID, v is a positive integer that does not correspond to any previous variable, and $\ell_1, \ell_2, \ldots, \ell_m$ is a sequence of m integers, indicating the arguments as literals of existing variables. The variables associated with these literals must be quantified variables. This declaration implicitly causes a single clauses to be added to the proof, as is documented in Table 3(C). It is a unit clause indicating that the operation evaluates to 1 during the forward implication proof. The dependency set for the operation is the set of variables associated with the argument literals.

A.2 Overall Proof Requirements

We classify a compiled formula as *special* when it is unsatisfiable or a tautology. Otherwise it is *standard*. We describe the proof requirements for the special cases after we cover the standard ones. For standard cases, an SCPOG proof consists of the following:

- A declaration of the root variable.
- Declarations of POG sum, product, and Skolem operations
- RUP clause additions

Each of these steps must obey its respective rules. For a standard formula, the proof must, via a sequence of clause additions, generate the root clause $[r]$.

The standard case includes some formulas that might seem to require special consideration. A CNF formula containing only the unit clause $[\ell]$, for literal ℓ, can be compiled into a POG consisting of a product node with ℓ as its only argument. The forward implication proof can then proceed in the standard form. With projected knowledge compilation, a formula can become that of a single literal via projection. For example, the formula $(a \vee x) \wedge (a \vee \overline{x})$, when projected for quantified variable x becomes a. In this case, however, the Skolem POG for the formula will contain several nodes and satisfy the conditions for the standard cases.

A.3 Special Cases

For standard knowledge compilation, a formula can be a tautology only if it either contains no clause, or if every input clause contains some variable a and its complement $\overline{a}$. The POG representation consists of a single product node with no arguments, encoding constant 1. The declaration of this node yields an extension variable v that serves as the root variable: $r = v$. The unit clause for this variable is implicitly added by the declaration of the product operation. There is no need for a forward implication proof, since the implication holds trivially.

With projected knowledge compilation, a formula can become a tautology via projection. For example, the formula $(a \vee x) \wedge (\overline{a} \vee \overline{x})$, when projected for quantified variable x becomes $(a \vee \overline{a})$. In this case, however, the Skolem POG of this formula will contain several nodes and satisfy the conditions for the standard cases.

As is shown in Table 2, the SCPOG format representation of an unsatisfiable formula has a root declaration with value 0, rather than a literal. The forward implication proof consists of a sequence of clause addition steps leading to the addition of the empty clause. This special case holds for both standard and projected knowledge compilation – an unsatisfiable formula cannot become satisfiable via projection, nor can a satisfiable formula become unsatisfiable via projection.

CNFs and DNFs with Exactly k Solutions

L. Sunil Chandran
Indian Institute of Science, Bengaluru, India

Rishikesh Gajjala
Indian Institute of Science, Bengaluru, India

Kuldeep S. Meel
Georgia Institute of Technology, Atlanta, GA, USA
University of Toronto, Canada

Abstract

Model counting is a fundamental problem that consists of determining the number of satisfying assignments for a given Boolean formula. The weighted variant, which computes the weighted sum of satisfying assignments, has extensive applications in probabilistic reasoning, network reliability, statistical physics, and formal verification. A common approach for solving weighted model counting is to reduce it to unweighted model counting, which raises an important question: *What is the minimum number of terms (or clauses) required to construct a DNF (or CNF) formula with exactly k satisfying assignments?*

In this paper, we establish both upper and lower bounds on this question. We prove that for any natural number k, one can construct a monotone DNF formula with exactly k satisfying assignments using at most $O(\sqrt{\log k}\log\log k)$ terms. This construction represents the first $o(\log k)$ upper bound for this problem. We complement this result by showing that there exist infinitely many values of k for which any DNF or CNF representation requires at least $\Omega(\log\log k)$ terms or clauses. These results have significant implications for the efficiency of model counting algorithms based on formula transformations.

2012 ACM Subject Classification Theory of computation → Logic

Keywords and phrases Model counting, #SAT, Set Systems, Combinatorics

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.9

Funding This research was funded in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference number RGPIN-2024-05956.

1 Introduction

The (unweighted) model counting is a classical problem in which one has to find the number of satisfying assignments for a given boolean formula. Usually, Boolean formulae are considered in two forms – Conjunctive Normal Form (CNF) and Disjunctive Normal Form (DNF). In the former case, the formula is written as conjunctions (ANDs) of clauses. (Clauses are literals combined using ORs.) In the latter case the formula is written as disjunctions (ORs) of terms. (Terms are literals combined using ANDs). Note that a variable in a formula may appear either in positive form or in negated form. A monotone formula in DNF (or in CNF) consists of only variables in positive form.

The unweighted model counting problem was shown to be #P-complete for formulae in both CNF and DNF by Valiant [20]. The weighted model counting is a generalization of this problem. We are not providing a complete definition of this problem here since it is technical, but it can be found in [6]. Weighted model counting has been extensively studied [7, 10] due to its diverse applications across multiple domains. These applications include probabilistic reasoning [16], network reliability estimation [17], statistical physics, probabilistic databases [19], program synthesis, and system verification [8, 21]. The fundamental nature of weighted


© L. Sunil Chandran, Rishikesh Gajjala, and Kuldeep S. Meel;
licensed under Creative Commons License CC-BY 4.0



model counting has led to its emergence as a core computational problem in areas requiring reasoning under uncertainty, where the ability to compute weighted sums across large combinatorial spaces is essential.

A natural approach for solving the weighted model counting problem is to reduce it to the unweighted model counting problem and then to use the existing solvers for the latter [6]. This reduction approach has proven effective across various application domains as it leverages the significant advances in unweighted model counting algorithms. A key subroutine in this reduction involves finding DNFs or CNFs with exactly k satisfying assignments for a given positive integer k. The reduction becomes more efficient as the number of terms (or clauses) in the DNF or CNF decreases. While the number of terms is not the only factor of relevance, it raises a very natural question: *What is the minimum number of terms (or clauses) required to construct a DNF (or CNF) formula with exactly k satisfying assignments?*

This can be quantified using $\beta(k)$, which is defined as follows:

▶ **Definition 1.** *The minimum number of terms or clauses needed to generate a DNF or CNF with exactly k satisfying assignments is defined to be $\beta(k)$.*

It is known that $\beta(k) = O(\log k)$ [6]. In this work, using an interesting connection to ideals of set systems, we give new lower and upper bounds on $\beta(k)$.

It is easy to see that many arbitrarily large numbers k exist for which $\beta(k) = 1$. For example, consider the DNF in which (x_1) is the only term and $x_2, x_3 \cdots x_{q+1}$ are the free variables, i.e., the variables which do not appear in the DNF (and hence their truth value does not affect the satisfiability of the formulae). Therefore, all k of the form 2^q can be generated using a DNF with only 1 term, i.e., $\beta(k) = 1$. At the same time, there exists k for which we need at least $\Omega(\log \log k)$ terms to generate a DNF with exactly k solutions (we prove this statement later). Thus, $\beta(k)$ does not increase or decrease monotonically with k. This motivates us to introduce a parameter called *block count* of k, which is more intimately associated with the number of terms needed to generate a DNF with exactly a given number of satisfying assignments.

Let $\mathbf{1}_m$ denote $\mathbf{11}\cdots m$ times and $\mathbf{0}_q$ denote $\mathbf{00}\cdots q$ times. Using this notation, the binary representation of 49, namely 110001 can be represented as $\mathbf{1}_2\mathbf{0}_3\mathbf{1}_1$.

▶ **Definition 2.** *The block binary representation of any $k \in \mathbb{N}$, is defined to be the unique representation $\mathbf{1}_{q_b}\mathbf{0}_{l_b}\cdots\mathbf{1}_{q_2}\mathbf{0}_{l_2}\mathbf{1}_{q_1}\mathbf{0}_{l_1}$ where $q_i > 0$ and $l_j > 0$ for all $i \in [b]$ and $j \in [2, b]$. Note that l_1 can be 0. For k with such a representation, its block count, $bl(k) = b$.*

The main result of our paper establishes a relationship between the block count of a number and the minimum number of terms needed to construct a DNF with exactly that many satisfying assignments.

▶ **Theorem 3** (Main Result)**.** *For every $k \geq 3$,*

$$\log\left(bl(k)+1\right) \leq \beta(k) \leq \min\left\{20\sqrt{\log k}\log\log k,\ bl(k)+1\right\}.$$

This represents the first $o(\log k)$ construction for this problem. We also conjecture that the value of $\beta(k)$ is polynomial in $\log\left(bl(k)+1\right)$.

▶ **Conjecture 4.** *There exists a sufficiently large constant C and a function $f(x)$, which is polynomial in x such that for every $k \in \mathbb{N}$, $\beta(k) \leq C \cdot f(\log\left(bl(k)\right))$.*

Organization

The rest of the paper is organized as follows. In Section 2, we establish a connection between the problem of finding minimum-sized DNFs with exactly k satisfying assignments and the theory of ideals of set systems. Section 3 provides the proof of our lower bound, while Section 4, which forms the technical core of the paper, presents the proof of upper bound. We finally conclude in Section 5.

2 Connection to Ideals of Sets System

We now show that a natural problem on the ideals of sets system (which is also of independent interest) is equivalent to finding small monotone DNFs (formulas consisting of only variables in positive form) with exactly a given number of satisfying assignments. We use this formulation to derive our upper bounds for $\beta(k)$.

2.1 Notation

$\mathbb{N}$ denotes the set of natural numbers. We use $\log k$ to denote $\log_2 k$. For a set S, $|S|$ and 2^S denote its cardinality and power set, respectively. The union of A and B is denoted as $A \cup B$. The union of two disjoint sets A, B is denoted as $A \sqcup B$. The notation $[a, b]$ represents $\{a, a+1 \cdots b\}$ and $[b]$ represents $[1, b]$. For every number i, we create distinct copies $i_0, i_1, i_2 \cdots$. The set $[w]_i$ represents $\{1_i, 2_i \cdots w_i\}$. Note that the sets $[w], [w]_0, [w]_1 \cdots$ are all different from each other as these sets are pairwise disjoint. Given a family of sets $\mathcal{S} = \{S_1, S_2 \cdots S_t\}$ and a set X, we define $\mathcal{S} + X = \{S_1 \cup X, S_2 \cup X \cdots S_t \cup X\}$.

An anti-chain is a subset $\mathcal{A}$ of a partially ordered set P such that any two distinct elements of $\mathcal{A}$ are incomparable. An (order) ideal (also called semi-ideal, down-set, or monotone decreasing subset) of P is a subset I of P such that if $t \in I$ and $s \leq t$, then $s \in I$. Similarly, a dual order ideal (also called up-set or monotone increasing subset) is a subset I of P such that if $t \in I$ and $s \geq t$, then $s \in I$ [18]. When P is finite, there is a one-to-one correspondence between anti-chains of P and order ideals: the anti-chain $\mathcal{A}$ associated with the order ideal I is the set of maximal elements of I, while $I = \{\, s \in P \mid s \leq t \text{ for some } t \in \mathcal{A} \,\}$. Then the anti-chain $\mathcal{A}$ is said to generate the ideal $I = \mathbf{ID}(\mathcal{A})$.

▶ Remark 5. There may be some difference of opinion with the definition of ideal given above since, in some contexts, a slightly different definition is used for ideals. However, in this paper, we only study set systems, and with respect to set systems, most authors use the above definition for ideals. For example, see Bollabás [4].

2.2 Problem Definition

▶ **Definition 6.** *The ideal generated by a family of sets, $\mathcal{S} = \{S_1, S_2 \cdots S_\alpha\}$, is $\mathbf{ID}(\mathcal{S}) = 2^{S_1} \bigcup 2^{S_2} \bigcup \cdots 2^{S_\alpha}$.*

Note that the minimal family of sets that generates a given ideal is an antichain.

▶ **Definition 7.** *Given a natural number k, $\alpha(k)$ is defined to be the minimum $|\mathcal{S}|$ for which $|\mathbf{ID}(\mathcal{S})| = k$.*

Observe that for every natural number k, there is a family of sets $\mathcal{S} = \{\emptyset, \{1\}, \{2\} \cdots \{k-1\}\}$ such that $|\mathbf{ID}(\mathcal{S})| = k$. Therefore, $\alpha(k)$ exists for all k and moreover, $\alpha(k) \leq k$. In this work, we establish more meaningful bounds on $\alpha(k)$.

2.3 Combinatorial Background

Ideals and their symmetric counterpart filters are central concepts in the study of set systems. These concepts appear in some of the most fundamental theorems regarding set systems. For example, Bollobás and Thomason proved that every non-trivial monotone increasing/decreasing property of subsets of a set has a threshold function [5], in the probabilistic model where each element is chosen with probability p. Here, monotone decreasing property corresponds to ideals. This is one of the most significant results in the theory of random graphs (see chapter 6 of [4]).

Another well-known result on ideals and dual order ideals is Kleitman's lemma [12], which triggered a long line of research on correlation-type inequalities, culminating in the Four Functions Theorem of Ahlswede and Daykin [1] (see chapter 6 of [3]). When studying extremal problems on set systems, it is often sufficient to prove the extremality restricted to set systems that are ideals or dual-order ideals. For example, see Kleitman's proof establishing a tight bound for the cardinality of maximal l-intersecting families [13] (see chapter 13 of [4]).

In chapter 17 of [4], Bollabás discusses theorems of the form $(m, k) \to (r, s)$ regarding traces. Such a theorem means if the universe $X = [k]$ and a family $\mathcal{F}$ consists of m subsets of X, then there exists an s-element subset S of X such that when we take the intersection of S with the members of $\mathcal{F}$, we get at least r distinct subsets. Alon [2] and Frankl [11] independently proved that to establish any theorem of the form $(m, k) \to (r, s)$, it is sufficient to prove the corresponding statement when $\mathcal{F}$ is restricted to an ideal [11].

Ideals are also studied under the name abstract simplicial complex or abstract complex. This represents a combinatorial description of the geometric notion of a simplicial complex [15]. In the context of matroids and greedoids, these are also referred to as independence systems [14].

Several questions closely related to our work have been studied in the literature. For instance, Duffus, Howard, and Leader investigated the maximum cardinality of an anti-chain that can be present in a given ideal [9]. The problem we discuss in this paper – finding the minimum cardinality $\alpha(k)$ of the anti-chain that can generate an ideal of a given size k – has been examined by both computer scientists and combinatorialists due to its applications in model counting.

2.4 Connection between Ideals and Monotone DNFs

The model counting problem for monotone DNF formulae has an interesting connection to the ideals of set systems. In fact, these problems are essentially equivalent. Let $x_1, x_2, \ldots, x_m$ be the set of positive literals used in the monotone DNF formulae we consider. An assignment assigns a truth value to each of these m variables; if the formula evaluates to TRUE under this assignment, then the assignment is called a satisfying assignment.

Take the universe $U = [m] = \{1, \ldots, m\}$. Given any subset S of U, we can associate to S a term $T_S = \bigwedge_{i \in S} x_i$. Conversely, given a term $T = \{x_{i_1} \wedge x_{i_2} \wedge \ldots \wedge x_{i_t}\}$ of a monotone DNF formula, we can associate the subset $S_T = \{i_1, i_2, \ldots, i_t\}$ to T. Also, to a family $\mathcal{F} = \{S_1, S_2, \ldots, S_t\}$ of subsets of U, we can associate a monotone DNF formula $f_{\mathcal{F}} = T(S_1) \vee T(S_2) \vee \ldots \vee T(S_t)$. Conversely, to a monotone DNF formula $f = T_1 \vee T_2 \vee \ldots \vee T_\ell$, we can associate a family of subsets of U, namely $\mathcal{F}_f = \{S_{T_1}, S_{T_2}, \ldots, S_{T_\ell}\}$.

Thus, there is a one-to-one correspondence between monotone DNF formulae using variables $x_1, x_2, \ldots, x_m$ and families of subsets of U.

Let $\mathcal{F}$ be a family of subsets. Then let $\overline{\mathcal{F}} = \{\overline{S} : S \in \mathcal{F}\}$, where $\overline{S} = U \setminus S$ is the complement of S. We can show that the set of satisfying assignments of a monotone DNF formula f has a one-to-one correspondence with the ideal generated by the family $\overline{\mathcal{F}_f}$. This

is because for f to be satisfied, at least one term of f must be satisfied. If term T_i is satisfied, then all literals appearing in T_i must be set to TRUE. So, the set of literals that are set to FALSE must correspond to a set of indices S' such that $S' \subseteq \overline{S_{T_i}}$. In other words, $S' \in \mathbf{ID}(\overline{\mathcal{F}_f})$.

The converse is also true: For $S' \in \mathbf{ID}(\overline{\mathcal{F}_f})$, the assignment where each variable x_i with $i \in S'$ is set to FALSE and the remaining variables set to TRUE will be a satisfying assignment for f. This is because there will be a superset of S' in $\overline{\mathcal{F}_f}$, and the term in f that corresponds to the complement of this superset would evaluate to TRUE. Since every satisfying assignment can be bijectively mapped to the set of variables that are set to FALSE, the set of satisfying assignments of f are bijectively mapped to the ideal of $\overline{\mathcal{F}_f}$. From this discussion, we have:

▶ **Theorem 8.** *Let k be a positive integer. If $\mathcal{F}$ is a family of subsets with $|\mathbf{ID}(\mathcal{F})| = k$, then there exists a monotone DNF formula $f = f_{\overline{\mathcal{F}_f}}$ with exactly k satisfying assignments. In particular, a monotone DNF formula with the smallest number of terms and exactly k satisfying assignments will have $\alpha(k)$ terms.*

▶ **Corollary 9.** *For $k \geq 1$, $\alpha(k) \geq \beta(k)$.*

▶ Remark 10. A similar statement can be made about monotone CNF formulae. The difference is that k would represent the number of non-satisfying assignments, and a subset in the ideal would correspond to the variables assigned TRUE.

3 Proof of Lower Bound

We first state the inclusion-exclusion principle

▶ **Theorem 11.** *For finite sets $V_1, V_2 \cdots V_q$*

$$|\bigcup_{i=1}^{q} V_i| = \sum_{\emptyset \neq J \subseteq \{1,2\cdots,q\}} (-1)^{|J|+1} \left| \bigcap_{j \in J} V_j \right|.$$

For a formulae $\mathcal{F}$, let $Sol(\mathcal{F})$ denote the set of satisfying assignments for $\mathcal{F}$.

▶ **Observation 12.** *For a DNF formula $\mathcal{F} = T_1 \vee T_2 \cdots \vee T_q$, we have*

$$|Sol(\mathcal{F})| = \sum_{\emptyset \neq J \subseteq \{1,2\cdots,q\}} (-1)^{|J|+1} \left| Sol(\bigwedge_{j \in J} T_j) \right|.$$

Proof. For the DNF formula $\mathcal{F} = T_1 \vee T_2 \cdots \vee T_q$, it is easy to see that

$$Sol(\mathcal{F}) = \bigcup_{i=1}^{q} Sol(T_i).$$

Therefore, from Theorem 11

$$|\bigcup_{i=1}^{q} Sol(T_i)| = \sum_{\emptyset \neq J \subseteq \{1,2\cdots,q\}} (-1)^{|J|+1} \left| \bigcap_{j \in J} Sol(T_j) \right|$$

Observe that $Sol(T_i) \cap Sol(T_j) = Sol(T_i \wedge T_j)$. Therefore,

$$\sum_{\emptyset \neq J \subseteq \{1,2\cdots,q\}} (-1)^{|J|+1} \left| \bigcap_{j \in J} Sol(T_j) \right| = \sum_{\emptyset \neq J \subseteq \{1,2\cdots,q\}} (-1)^{|J|+1} \left| Sol(\bigwedge_{j \in J} T_j) \right| \quad \blacktriangleleft$$

▶ **Observation 13.** *For any non-empty set $J \subseteq \{1, 2 \cdots, q\}$, the value of $\left|Sol(\bigwedge_{j \in J} T_j)\right|$ is either 0 or of the form 2^α for some $\alpha \in \mathbb{N} \cup \{0\}$.*

Proof. Consider a literal y. We have the following cases

- If the literal y appears positively in T_i and negatively in T_j for some $i, j \in J$, then $\bigwedge_{j \in J} T_j$ has no solution.
- If the literal y appears positively for at least one T_i for $i \in J$ and never appears negatively for any T_j for $j \in J$, then y must be true in all satisfying assignments.
- Similarly, if the literal y appears negatively for at least one T_i for $i \in J$ and never appears positively for any T_j for $j \in J$, then y must be false in all satisfying assignments.
- If the literal y does not appear in any T_j for $j \in J$, then it can take the true or false value in an assignment.

It is now easy to see that if there are α such literals which never appeared in any T_j for $j \in J$, there will be 2^α many satisfying assignments. ◀

▶ **Observation 14.** *Let $t \in \mathbb{N}$ and $x_i, y_i \in \{0\} \cup \mathbb{N}$ for all $i \in [t]$. If $k = \sum_{i \in [t]} (-1)^{x_i} 2^{y_i} \geq 1$, then $bl(k) \leq t$.*

Proof. We prove this by induction on t. When $t = 1$, x_1 must be even since $k \geq 1$. Therefore, k is of the form 2^{y_1} and hence $bl(k) = 1 = t$.

Let $t > 1$ and $m = k - \min_{i \in [t]} (-1)^{x_i} 2^{y_i}$. For $i \in [t]$, if there exists an x_i which is odd, then $\min_{i \in [t]} (-1)^{x_i} 2^{y_i} < 0$ and hence $m \geq 1$. On the other hand, if for all $i \in [t]$, x_i is even, then m can be written as the sum of positive integers. Therefore, $m \geq 1$. Therefore, by the induction assumption $bl(m) \leq t - 1$.

Observe that when $(-1)^{x_i} 2^{y_i}$ is added to the binary representation of m, 1 is either added or subtracted at the $(y_i + 1)$th bit of m. Suppose the $(y_i + 1)$th bit of m be 0 (respectively 1). Then adding (respectively subtracting) 1 at the $(y_i + 1)$th position changes the number of blocks by at most 1. On the other hand, when the $(y_i + 1)$th bit of m is 1 (respectively 0), adding (respectively subtracting) 1 at the $(y_i + 1)$th position flips all contiguous 1s (respectively 0s) at and before $(y_i + 1)$th position and the first preceding 0 (respectively 1). Therefore, the number of blocks change by at most 1.

Therefore, as $k = m + (-1)^{x_t} 2^{y_t}$, $bl(k) \leq bl(m) + 1 \leq t$. ◀

▶ **Lemma 15.** *For every $k \in \mathbb{N}$, $\log(bl(k) + 1) \leq \beta(k)$.*

Proof. Towards a contradiction, let there exist some $k \in \mathbb{N}$ such that $\log(bl(k) + 1) > \beta(k)$. Let $\mathcal{F} = T_1 \vee T_2 \cdots \vee T_{\beta(k)}$ be a DNF such that it has exactly k solutions. From Observation 12,

$$|Sol(\mathcal{F})| = \sum_{\emptyset \neq J \subseteq \{1, 2 \cdots, \beta(k)\}} (-1)^{|J|+1} \left| Sol(\bigwedge_{j \in J} T_j) \right|$$

From Observation 13, it is easy to see that $|Sol(\mathcal{F})|$ can be written as the sum or difference of $2^{\beta(k)}$ or less terms that are powers of 2. Therefore, from Observation 14, $bl(k) \leq 2^{\beta(k)} - 1 < bl(k) + 1 - 1 = bl(k)$. This is a contradiction. ◀

From Corollary 9, it is also easy to see the following corollary now.

▶ **Corollary 16.** *For every $k \in \mathbb{N}$, $\log(bl(k) + 1) \leq \alpha(k)$.*

4 Proof of Upper Bound

We first establish a framework for analyzing the size of ideals generated by families of sets. Our approach leverages the inclusion-exclusion principle and introduces two key operations – splitting and lifting – that will be central to our construction.

4.1 Technical Preliminaries

A key insight for our analysis is understanding how the inclusion-exclusion principle applies to unions of power sets. This is captured in the following observation:

▶ **Observation 17.** *For finite sets* $V_1, V_2 \cdots V_q$

$$|\bigcup_{i=1}^{q} 2^{V_i}| = \sum_{j=1}^{q} (-1)^{j+1} (\sum_{1 \le i_1 < \cdots < i_j \le q} 2^{|V_{i_1} \cap \cdots \cap V_{i_j}|}).$$

Proof. From the inclusion-exclusion principle,

$$|\bigcup_{i=1}^{q} 2^{V_i}| = \sum_{j=1}^{q} (-1)^{j+1} (\sum_{1 \le i_1 < \cdots < i_j \le q} |2^{V_{i_1}} \cap \cdots \cap 2^{V_{i_j}}|)$$

$$= \sum_{j=1}^{q} (-1)^{j+1} (\sum_{1 \le i_1 < \cdots < i_j \le q} |2^{V_{i_1} \cap \cdots \cap V_{i_j}}|) = \sum_{j=1}^{q} (-1)^{j+1} (\sum_{1 \le i_1 < \cdots < i_j \le q} 2^{|V_{i_1} \cap \cdots \cap V_{i_j}|}) \quad \blacktriangleleft$$

4.2 Fundamental Operations: Splitting and Lifting

We now introduce two fundamental operations that will serve as building blocks for our upper-bound construction. These operations allow us to construct ideals with specific cardinalities efficiently.

▶ **Lemma 18** (Splitting lemma). *For* $m, k \in \mathbb{N}$, $\alpha(m + k) \le \alpha(m) + \alpha(k + 1)$.

Proof. Let $\mathcal{S}, \mathcal{T}$ be family of sets such that $S \cap T = \emptyset$ for all $S \in \mathcal{S}$ and $T \in \mathcal{T}$, where

$$|\mathcal{S}| = \alpha(m) \ \ \& \ \ |\mathbf{ID}(\mathcal{S})| = m$$

$$|\mathcal{T}| = \alpha(k+1) \ \ \& \ \ |\mathbf{ID}(\mathcal{T})| = k+1$$

Observe that, by construction $\mathbf{ID}(\mathcal{S}) \cap \mathbf{ID}(\mathcal{T}) = \{\emptyset\}$. Therefore,

$$|\mathbf{ID}(\mathcal{S} \cup \mathcal{T})| = |\mathbf{ID}(\mathcal{S})| + |\mathbf{ID}(\mathcal{T})| - 1 = m + k$$

It now follows that,

$$\alpha(m + k) \le |\mathcal{S} \cup \mathcal{T}| = |\mathcal{S}| + |\mathcal{T}| = \alpha(m) + \alpha(k + 1) \quad \blacktriangleleft$$

The splitting lemma essentially tells us that to construct an ideal of size $m + k$, we can combine ideals of sizes m and $k + 1$ that share only the empty set. This allows us to decompose the problem of constructing larger ideals into constructing smaller ones.

Our second fundamental operation is the lifting lemma, which provides an efficient way to construct ideals whose cardinality is a power of 2 multiplied by a given number:

▶ **Lemma 19** (Lifting lemma). *For every* $t, k \in \mathbb{N}$, $\alpha(2^t \cdot k) \le \alpha(k)$.

Proof. Let $\mathcal{S} = \{S_1, S_2 \cdots S_{\alpha(k)}\}$ be a family of sets such that $|\mathbf{ID}(\mathcal{S})| = k$. Let the set $X = \{x_1, x_2 \cdots x_t\}$ be such that $X \cap S_i = \emptyset$ for all $S_i \in \mathcal{S}$. We define $S'_i = S_i \sqcup X$ for all $i \in [\alpha(k)]$ and $\mathcal{S}' = \{S'_1, S'_2 \cdots S'_{\alpha(k)}\}$. By Observation 17,

$$|\mathbf{ID}(\mathcal{S}')| = |\bigcup_{i=1}^{\alpha(k)} 2^{S'_i}| = \sum_{j=1}^{\alpha(k)} (-1)^{j+1} (\sum_{1 \le i_1 < \cdots < i_j \le \alpha(k)} 2^{|S'_{i_1} \cap \cdots \cap S'_{i_j}|})$$

By construction,

$$= \sum_{j=1}^{\alpha(k)} (-1)^{j+1} (\sum_{1 \le i_1 < \cdots < i_j \le \alpha(k)} 2^{|(S_{i_1} \cap \cdots \cap S_{i_j}) \sqcup X|})$$

$$= 2^{|X|} \sum_{j=1}^{\alpha(k)} (-1)^{j+1} (\sum_{1 \le i_1 < \cdots < i_j \le \alpha(k)} 2^{|S_{i_1} \cap \cdots \cap S_{i_j}|})$$

By Observation 17,

$$2^{|X|} |\bigcup_{i=1}^{\alpha(k)} 2^{S_i}| = 2^t |\mathbf{ID}(\mathcal{S})| = 2^t \cdot k$$

Therefore, $\alpha(2^t \cdot k) \le |\mathcal{S}'| = |\mathcal{S}| = \alpha(k)$ ◀

The lifting lemma provides a powerful tool: it shows that we can construct an ideal of size $2^t \cdot k$ using the same number of generators as an ideal of size k. Intuitively, this is achieved by adding t new elements to each generator set in a way that preserves the relative structure of the original ideal.

4.3 Simple Upper Bound Based on Block Count

Using the operations we've developed, we can now establish a simple relationship between $\alpha(k)$ and the block count of k:

▶ **Lemma 20.** *For every $k \in \mathbb{N}$, $\alpha(k) \le bl(k) + 1$.*

Proof. We prove this by induction on the block count of k. We first give a construction for the base case, that is, for any k with $bl(k) = 1$, say $k = \mathbf{1}_{q_1}\mathbf{0}_{l_1}$, where $q_1 \ge 1$ and $l_1 \ge 0$. Consider the sets $S_1 = [q_1 + l_1 - 1]_1$ and $S_2 = [q_1 - 1]_2 \sqcup [l_1]_1$. Note that $S_1 \bigcap S_2 = [l_1]_1$. We now observe that,

$$|\mathbf{ID}(\{S_1, S_2\})| = 2^{|S_1|} + 2^{|S_2|} - 2^{|S_1 \bigcap S_2|} = 2^{q_1+l_1} - 2^{l_1} = \mathbf{1}_{q_1}\mathbf{0}_{l_1}$$

Thus for k, with $bl(k) = 1$, $\alpha(k) \le 2$. We now consider any number k with $bl(k) = b \ge 2$, say $1_{q_b}0_{l_b} \cdots 1_{q_2}0_{l_2}1_{q_1}0_{l_1}$, where $q_i > 0$ and $l_j > 0$ for all $i \in [b]$ and $j \in [2, b]$. From the *Lifting lemma*,

$$\alpha(k) = \alpha(1_{q_b}0_{l_b} \cdots 1_{q_2}0_{l_2}1_{q_1}0_{l_1}) \le \alpha(1_{q_b}0_{l_b} \cdots 1_{q_2}0_{l_2}1_{q_1}) = \alpha(1_{q_b}0_{l_b} \cdots 1_{q_2}0_{l_2+q_1} + 1_{q_1})$$

From the *Splitting lemma*,

$$\le \alpha(1_{q_b}0_{l_b} \cdots 1_{q_2}0_{l_2+q_1}) + \alpha(1_{q_1} + 1) = \alpha(1_{q_b}0_{l_b} \cdots 1_{q_2}0_{l_2+q_1}) + \alpha(2^{q_1})$$

From the induction assumption,

$$\le b + \alpha(2^{q_1}) = b + 1 = bl(k) + 1$$ ◀

This lemma establishes that $\alpha(k)$ grows no faster than the block count of k plus one. While this already gives us a non-trivial upper bound, we will develop tighter bounds in the next section.

4.4 Tighter Upper Bound Construction

We now present our main technical result, which establishes a much tighter bound on $\alpha(k)$. The key insight is to construct specialized sets for numbers of a particular form and then extend these constructions to all natural numbers.

▶ **Theorem 21.** *For m of the form $2^{3q^2} + \beta$ where $\beta < 2^{q^2}$, $\alpha(m) \leq (q+1)\lceil \log q \rceil + 4q + 6$.*

The proof of this theorem is technical and will be presented in Section 4.5. First, we'll show how this theorem helps us establish our main result, Theorem 3.

▶ **Observation 22.** *For every $k \in \mathbb{N}$, there exists a q such that $k = 2^{3q^2} + \gamma \cdot 2^{q^2} + \beta < 2^{3(q+1)^2}$ where $\gamma = \lfloor \frac{k - 2^{3q^2}}{2^{q^2}} \rfloor$ and $0 \leq \beta < 2^{q^2}$.*

Proof. For every number k, there exists a q such that $2^{3q^2} \leq k < 2^{3(q+1)^2}$. By dividing $k - 2^{3q^2}$ by 2^{q^2}, it follows that $\gamma = \lfloor \frac{k - 2^{3q^2}}{2^{q^2}} \rfloor$ is the quotient and $0 \leq \beta < 2^{q^2}$ is the remainder. ◀

This observation shows that any natural number can be decomposed into the form required by Theorem 21, plus an additional term. We now show how to handle this additional term:

▶ **Lemma 23.** $\alpha(2^{3q^2} + \gamma \cdot 2^{q^2} + \beta) \leq \alpha(2^{3q^2} + \beta) + \alpha(\gamma) + 1$

Proof. From the Splitting lemma,

$$\alpha(2^{3q^2} + \gamma \cdot 2^{q^2} + \beta) \leq \alpha(2^{3q^2} + \beta) + \alpha(\gamma \cdot 2^{q^2} + 1)$$

$$\leq \alpha(2^{3q^2} + \beta) + \alpha(\gamma \cdot 2^{q^2}) + \alpha(2) = \alpha(2^{3q^2} + \beta) + \alpha(\gamma \cdot 2^{q^2}) + 1$$

From the Lifting lemma,

$$= \alpha(2^{3q^2} + \beta) + \alpha(\gamma) + 1$$ ◀

With these pieces in place, we are now ready to prove our main result, which provides an $O(\sqrt{\log k} \log\log k)$ upper bound on $\alpha(k)$.

Proof of Theorem 3. From Observation 22, for every k, there exists a q such that $k = 2^{3q^2} + \gamma \cdot 2^{q^2} + \beta < 2^{3(q+1)^2}$ where $\gamma = \lfloor \frac{k - 2^{3q^2}}{2^{q^2}} \rfloor$ and $0 \leq \beta < 2^{q^2}$.

When $3 \leq k < 20$, it is obvious that $\alpha(k) \leq k < 20\sqrt{\log k} \log\log k$. So, we can assume that $20 \leq k$ and therefore $q \geq 1$.

▶ **Observation 24.** *If $\log 3 \leq \log k \leq 30000$, then $\lceil 0.5 \log k + 1 \rceil < 20\sqrt{\log k} \log\log k$.*

We prove Observation 24 in Section A.1.

From our simple upper bound in Lemma 20 along with Observation 24, we get that for all $q < 100$,

$$\alpha(k) \leq \lceil 0.5 \log k \rceil + 1 < 20\sqrt{\log k} \log\log k$$

We now inductively prove Theorem 3 for $q \geq 100$, inducting on q.

$\alpha(k) = \alpha(2^{3q^2} + \gamma \cdot 2^{q^2} + \beta)$

When $\gamma = 0$, by Theorem 21, $\alpha(k) \leq 20\sqrt{\log k} \log\log k$. For $\gamma > 0$, from Lemma 23,

$\leq \alpha(2^{3q^2} + \beta) + \alpha(\gamma) + 1$

From Theorem 21,

$\leq (q+1)\lceil \log q \rceil + 4q + 7 + \alpha(\gamma)$

When $\gamma \leq 2$ and $\log k > 30000$, as $\alpha(1) = \alpha(2) = 1$, $\alpha(k) \leq 20\sqrt{\log k} \log\log k$. Therefore, for $\gamma \geq 3$, by induction assumption,

$\alpha(k) \leq (q+1)\lceil \log q \rceil + 4q + 7 + 20\sqrt{\log \gamma} \log\log \gamma$

As $\sqrt{\log \gamma} \leq \sqrt{\log k - q^2} \leq \sqrt{3(q+1)^2 - q^2} \leq \sqrt{2.1q^2}$ for $q \geq 100$,

$\leq (q+1)\lceil \log q \rceil + 4q + 7 + 20\sqrt{2.1}q \log\log k$

As $(q+1)\lceil \log q \rceil + 4q + 7 \leq 2q \log q$ for $q \geq 100$,

$\leq 2q \log q + 20\sqrt{2.1}q \log\log k \leq q \log\log k + 20\sqrt{2.1}q \log\log k$

$= (20\sqrt{2.1} + 1)q \log\log k \leq 20\sqrt{3}q \log\log k \leq 20\sqrt{\log k} \log\log k$ ◀

4.5 Proof of Theorem 21

We now present the most technical part of our proof: constructing an ideal with specific properties for numbers of the form $2^{3q^2} + \beta$ where $\beta < 2^{q^2}$. The key idea is to carefully design a collection of sets based on the binary representation of β.

Let $m = 2^{3q^2} + \beta$ where $\beta < 2^{q^2}$. For $i, j \in [0, q-1]$, we define F_{ij} in the following way:

- If the $(jq+i+1)$th least significant bit of β is 1, then fix $F_{ij} = \emptyset$.
  Let $\mathcal{F}_0$ be the family of all such sets.
- If the $(jq+i+1)$th least significant bit of β is 0, then fix $F_{ij} = [i]_{jq+i}$.
  Let $\mathcal{F}_1$ be the family of all such sets.

We note that the least significant bit of β is indexed to be the 1st bit. On the other hand the indices i, j start from 0. From this construction, several important properties immediately follow:

▶ Remark 25. For all $i, j \in [0, q-1]$, $F_{ij} \cap [q^2] = \emptyset$.

▶ Remark 26. For any $F_{ij}, F_{i'j'} \in \mathcal{F}_1$, $F_{ij} \bigcap F_{i'j'} \neq \emptyset$ if and only if $i = i'$ and $j = j'$.

▶ Remark 27. $\beta + \sum_{F_{ij} \in \mathcal{F}_1} 2^{jq+|F_{ij}|} = \beta + \sum_{F_{ij} \in \mathcal{F}_1} 2^{jq+i} = \mathbf{1}_{q^2} = 2^{q^2} - 1$

▶ Remark 28. $\sum_{F_{ij} \in \mathcal{F}_0} 2^{jq+|F_{ij}|} = \sum_{F_{ij} \in \mathcal{F}_0} 2^{jq}$

These observations lead to the following key relationship:

▶ **Corollary 29.** $2^{q^2} - \sum_{i,j \in [0,q-1]} 2^{jq+|F_{ij}|} = \beta + 1 - \sum_{F_{ij} \in \mathcal{F}_0} 2^{jq}$

Using these F_{ij} sets, we construct two families of sets that will form the basis of our ideal. For all $i, j \in [0, q-1]$, let $S'_i = [q^2], T'_j = [jq]$ and let

$$S_i = (\bigcup_{j=0}^{q-1} F_{ij}) \bigsqcup S'_i \;\; \& \;\; T_j = (\bigcup_{i=0}^{q-1} F_{ij}) \bigsqcup T'_j$$

We define $\mathcal{S} = \{S_0, \cdots S_{q-1}\}, \mathcal{S}' = \{S'_0, \cdots S'_{q-1}\}, \mathcal{T} = \{T_0, \cdots T_{q-1}\}$ and $\mathcal{T}' = \{T'_0, \cdots T'_{q-1}\}$

Our strategy is to compute the cardinality of the ideal generated by $\mathcal{S} \cup \mathcal{T}$. We want to show that $\alpha(|\mathbf{ID}(\mathcal{S} \cup \mathcal{T})|) \leq 2q$, and since $q = O(\sqrt{\log k})$, this would give us the desired upper bound if $|\mathbf{ID}(\mathcal{S} \cup \mathcal{T})| = \beta = m - 2^{3q^2}$. However, this equality doesn't hold exactly, but we'll show that the difference has a small α-value.

Computing $|\mathbf{ID}(\mathcal{S} \cup \mathcal{T})|$ directly is complex, so we first relate it to $|\mathbf{ID}(\mathcal{S}' \cup \mathcal{T}')|$, which equals 2^{q^2} because all sets in $\mathcal{S}'$ and $\mathcal{T}'$ are subsets of $[q^2]$ and every $S'_i = [q^2]$.

▶ **Observation 30.** $S_i \cap T_j = [jq] \bigsqcup F_{ij}$

Proof. By Remark 25 and Remark 26,

$$S_i \cap T_j = ((\bigcup_{j'=0}^{q-1} F_{ij}) \sqcup [q^2]) \cap ((\bigcup_{i'=0}^{q-1} F_{ij}) \sqcup [jq]) = [jq] \bigsqcup F_{ij}$$ ◀

▶ **Observation 31.** *For $p < i$, $2^{T_i} \cap 2^{T_p} = 2^{[pq]} = 2^{T'_i} \cap 2^{T'_p}$ and $2^{S_i} \cap 2^{S_p} = 2^{[q^2]} = 2^{S'_i} \cap 2^{S'_p}$.*

Proof. By Remark 25 and Remark 26,

$$2^{T_i} \cap 2^{T_p} = 2^{T_i \cap T_p} = 2^{T'_i \cap T'_p} = 2^{[pq]} = 2^{T'_i} \cap 2^{T'_p}$$

$$2^{S_i} \cap 2^{S_p} = 2^{S_i \cap S_p} = 2^{S'_i \cap S'_p} = 2^{[q^2]} = 2^{S'_i} \cap 2^{S'_p}$$ ◀

These observations allow us to relate terms in the inclusion-exclusion expansions of $|\mathbf{ID}(\mathcal{S} \cup \mathcal{T})|$ and $|\mathbf{ID}(\mathcal{S}' \cup \mathcal{T}')|$. We define:

$$(A_1, A_2 \cdots A_{2q}) = (2^{S_0}, 2^{S_1} \cdots 2^{S_{q-1}}, 2^{T_0}, 2^{T_1} \cdots 2^{T_{q-1}})$$

$$(A'_1, A'_2 \cdots A'_{2q}) = (2^{S'_0}, 2^{S'_1} \cdots 2^{S'_{q-1}}, 2^{T'_0}, 2^{T'_1} \cdots 2^{T'_{q-1}})$$

We note that the indices of A and A' start at 1 while indices of S, S', T, T' start at 0.

▶ **Observation 32.** *For any $\ell \geq 3$, let $I = \{i_1, i_2 \cdots i_\ell\}$ such that $1 \leq i_1 < \cdots < i_\ell \leq 2q$*

$$|\bigcap_{j \in I} A_j| = |\bigcap_{j \in I} A'_j|.$$

Proof. As $\ell \geq 3$, by pigeon hole principle, there exists two sets A_{i_x} and A_{i_y} such that either $A_{i_x}, A_{i_y} \in \{2^{S_0}, 2^{S_1} \cdots 2^{S_{q-1}}\}$ or $A_{i_x}, A_{i_y} \in \{2^{T_0}, 2^{T_1} \cdots 2^{T_{q-1}}\}$. From Observation 31, $A_{i_x} \cap A_{i_y} \subset 2^{[q^2]}$. It follows that $\bigcap_{j \in I} A_j \subset 2^{[q^2]}$. Therefore,

$$|\bigcap_{j \in I} A_j| = |\bigcap_{j \in I} (A_j \bigcap 2^{[q^2]})| = |\bigcap_{j \in I} A'_j|$$ ◀

This allows us to compute the difference between the two ideals' cardinalities:

▶ **Observation 33.**

$$|\mathbf{ID}(\mathcal{S} \cup \mathcal{T})| - |\mathbf{ID}(\mathcal{S}' \cup \mathcal{T}')|$$

$$= (\sum_{i=0}^{q-1} |2^{S_i}| - \sum_{i=0}^{q-1} |2^{S'_i}|) + (\sum_{j=0}^{q-1} |2^{T_j}| - \sum_{j=0}^{q-1} |2^{T'_j}|) - (\sum_{i,j \in [0,q-1]} (|2^{S_i} \bigcap T_j|) - \sum_{i,j \in [0,q-1]} (|2^{S'_i} \bigcap T'_j|))$$

Through a series of algebraic manipulations (Proof in Section A.2), we derive the exact cardinality of our constructed ideal:

▶ **Lemma 34.** $|\mathbf{ID}(\mathcal{S} \cup \mathcal{T})| = \sum_{i=0}^{q-1} |2^{S_i}| + \sum_{j=0}^{q-1} |2^{T_j}| - \sum_{i,j \in [0,q-1]} 2^{jq}(2^{|F_{ij}|}) + (q-1)(\sum_{j=0}^{q-1} 2^{jq} - 2^{q^2})$

This leads to a bound on the α-value of our constructed ideal (as $|\mathcal{S} \cup \mathcal{T}| = 2q$):

▶ **Corollary 35.** $\alpha(\sum_{i=0}^{q-1} |2^{S_i}| + \sum_{j=0}^{q-1} |2^{T_j}| - \sum_{i,j \in [0,q-1]} 2^{jq}(2^{|F_{ij}|}) + (q-1)(\sum_{j=0}^{q-1} 2^{jq} - 2^{q^2})) = \alpha(|\mathbf{ID}(\mathcal{S} \cup \mathcal{T})|) \leq 2q$

Our goal is to show that $\alpha(m - |\mathbf{ID}(\mathcal{S} \cup \mathcal{T})|) = O(q \log q)$, which would imply $\alpha(m) = O(q \log q) = O(\sqrt{\log k} \log \log k)$ by the splitting lemma and Corollary 35. However, the expression for $|\mathbf{ID}(\mathcal{S} \cup \mathcal{T})|$ contains inconvenient terms like $\sum_{i=0}^{q-1} |2^{S_i}|$. We first eliminate these terms:

▶ **Observation 36.** $\alpha(2^{3q^2-1} - \sum_{i=0}^{q-1} |2^{S_i}| - \sum_{j=0}^{q-1} |2^{T_j}| + 2^{q^2}) \leq 2q + 3$

Proof. From Observation 14, $2^{3q^2-1} - \sum_{i=0}^{q-1} |2^{S_i}| - \sum_{j=0}^{q-1} |2^{T_j}| + 2^{q^2}$ has at most $2q+2$ blocks. Therefore, from our earlier results on block counts and α values, $\alpha(2^{3q^2-1} - \sum_{i=0}^{q-1} |2^{S_i}| - \sum_{j=0}^{q-1} |2^{T_j}| + 2^{q^2}) \leq 2q + 3$ ◀

For clarity, we define two auxiliary values: $t_1 = 2^{3q^2-1} + 2^{q^2} - 1 - \sum_{i,j \in [0,q-1]} 2^{jq}(2^{|F_{ij}|}) + (q-1)(\sum_{j=0}^{q-1} 2^{jq} - 2^{q^2})$

$t_2 = 2^{3q^2-1} + \sum_{F_{ij} \in \mathcal{F}_0} 2^{jq} - (q-1)(\sum_{j=0}^{q-1} 2^{jq} - 2^{q^2})$

Here, t_1 is the sum of $|\mathbf{ID}(\mathcal{S} \cup \mathcal{T})|$ and the expression from Observation 36. We now show that $t_2 = m - t_1$:

▶ **Observation 37.** $m = t_1 + t_2$

Proof. By definition, $t_1 + t_2 = 2^{3q^2} + 2^{q^2} - 1 - \sum_{i,j \in [0,q-1]} 2^{jq+|F_{ij}|} + \sum_{F_{ij} \in \mathcal{F}_0} 2^{jq}$

From Corollary 29, $= 2^{3q^2} + \beta = m$ ◀

▶ **Observation 38.** $\alpha(t_1) \leq 4q + 3$

Proof. This follows from the splitting lemma along with Corollary 35 and Observation 36. ◀

By the splitting lemma and Observation 37, it is now sufficient to prove that $\alpha(t_2 + 1)$ is small. We'll show that t_2 can be written in a special form that allows us to bound its α-value efficiently:

▶ **Observation 39.** *There exists* $-1 \leq a_j \leq q-1$ *for* $j \in [0, q-1]$ *and* $a_q = -(q-1)$*, for which* $t_2 = 2^{3q^2-1} - \sum_{j=0}^{q} a_j 2^{jq}$.

Proof. $t_2 = 2^{3q^2-1} + \sum_{F_{ij} \in \mathcal{F}_0} 2^{jq} - (q-1)(\sum_{j=0}^{q-1} 2^{jq} - 2^{q^2})$

For a given j, let $0 \leq b_j \leq q$ be the number of $F_{ij} \in \mathcal{F}_0 = 2^{3q^2-1} + (q-1)2^{q^2} - \sum_{j=0}^{q-1}(q-1-b_j)2^{jq}$

Therefore, for some $-1 \leq a_j \leq q-1$ for $j \in [0, q-1]$ and $a_q = -(q-1) = 2^{3q^2-1} - \sum_{j=0}^{q} a_j 2^{jq}$ ◀

Finally, we show that numbers with this special structure have a small α-value:

▶ **Lemma 40.** *For any* $|a_j| \leq q-1$ *for* $j \in [0, q]$, $\alpha(2^{3q^2-1} - \sum_{j=0}^{q} a_j 2^{jq} + 1) \leq (q+1)\lceil \log q \rceil + 3$.

Proof. The binary representation of $|a_j|$ has at most $\lceil \log q \rceil$ non-zero bits. Therefore, $|a_j|2^{jq}$ has at most $\lceil \log q \rceil$ non-zero bits in its binary representation. This means $\sum_{j=0}^{q} a_j 2^{jq}$ can be written as the sum or difference of $(q+1)\lceil \log q \rceil$ powers of 2. From our earlier results on block counts, $2^{3q^2-1} - \sum_{j=0}^{q} a_j 2^{jq} + 1$ has at most $(q+1)\lceil \log q \rceil + 2$ blocks. Therefore, $\alpha(2^{3q^2-1} - \sum_{j=0}^{q} a_j 2^{jq} + 1) \leq (q+1)\lceil \log q \rceil + 3$ ◀

▶ **Corollary 41.** $\alpha(t_2 + 1) \leq (q+1)\lceil \log q \rceil + 3$

Combining all these results, Theorem 21 follows from the splitting lemma along with Corollary 41, Observation 38, and Observation 37.

5 Conclusion and Open Problems

We have established that for every $k \geq 3$, there exists a DNF or CNF with exactly k satisfying assignments using at most $O(\sqrt{\log k} \log\log k)$ terms or clauses. Our construction provides the first $o(\log k)$ upper bound for this problem, significantly improving previous bounds [6]. The constructed DNFs also have the desirable property of being monotone, which simplifies their structure and analysis. On the other hand, we also give a lower bound showing that there exist infinitely many $k \in \mathbb{N}$ requiring at least $\Omega(\log\log k)$ terms or clauses. However, there remains a gap between our upper and lower bounds that presents an interesting avenue for future research.

We conjecture that the value of $\beta(k)$ is polynomial in $\log(bl(k)+1)$, which would provide a more precise characterization of the relationship between the number of terms needed and the block structure of the number of solutions. Resolving this conjecture would further deepen our understanding of the structural properties of boolean formulas with a specific number of satisfying assignments.

The connection we established between this problem and the theory of ideals in set systems may also lead to further applications in other areas of combinatorics and computational complexity theory. In particular, the construction techniques we developed might be useful in addressing related questions about the expressiveness and succinctness of different representations of boolean functions.

References

1 Rudolf Ahlswede and David E. Daykin. An inequality for the weights of two families of sets, their unions and intersections. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 43:183–185, 1978.

2 Noga Alon. On the density of sets of vectors. *Discrete Mathematics*, 46(2):199–202, 1983. doi:10.1016/0012-365X(83)90253-4.

3 Noga Alon and Joel H. Spencer. *The Probabilistic Method*, chapter 6, pages 81–92. John Wiley & Sons, Ltd, 2000.

4 Béla Bollobás. *Combinatorics: Set Systems, Hypergraphs, Families of Vectors, and Combinatorial Probability*. Cambridge University Press, USA, 1986.

5 Béla Bollobás and Andrew G. Thomason. Threshold functions. *Combinatorica*, 7:35–38, 1987. doi:10.1007/BF02579198.

6 Supratik Chakraborty, Dror Fried, Kuldeep S. Meel, and Moshe Y. Vardi. From weighted to unweighted model counting. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 689–695. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/103.

7 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002. doi:10.1613/jair.989.

8 Carmel Domshlak and Jörg Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *J. Artif. Intell. Res.*, 30:565–620, 2007. doi:10.1613/jair.2289.

9 Dwight Duffus, David Howard, and Imre Leader. The width of downsets. *European Journal of Combinatorics*, 79:46–59, 2019. doi:10.1016/j.ejc.2018.11.005.

10 Jörg Flum and Martin Grohe. The parameterized complexity of counting problems. *SIAM J. Comput.*, 33(4):892–922, 2004. doi:10.1137/S0097539703427203.

11 Peter Frankl. On the trace of finite sets. *J. Comb. Theory, Ser. A*, 34:41–45, 1983. doi:10.1016/0097-3165(83)90038-9.

12 Daniel J. Kleitman. Families of non-disjoint subsets. *Journal of Combinatorial Theory*, 1(1):153–155, 1966.

13 Daniel J. Kleitman. On a combinatorial conjecture of erdös. *Journal of Combinatorial Theory*, 1(2):209–214, 1966.

14 Bernhard Körte, Rainer Schräder, and László Lovász. Greedoids. *Springer*, January 1991.

15 John M. Lee. Introduction to topological manifolds. *Springer*, January 2011.

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

17 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.

18 Richard P. Stanley. *Enumerative Combinatorics*, volume 1. Cambridge University Press, USA, 2nd edition, 2011. pg. 282.

19 Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic databases*. Springer Nature, 2022.

20 Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979. doi:10.1137/0208032.

21 Yexiang Xue, Arthur Choi, and Adnan Darwiche. Basing decisions on sentences in decision diagrams. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*, pages 842–849. AAAI Press, 2012. doi:10.1609/aaai.v26i1.8221.

A Deferred proofs

A.1 Proof of Observation 24

We claim that for every integer $3 \leq k \leq 2^{30000}$,

$$\left\lceil \tfrac{1}{2} \log_2 k + 1 \right\rceil \;<\; 20 \sqrt{\log_2 k}\, \log_2\!\big(\log_2 k\big).$$

Proof. Put $x := \log_2 k$. Then $x \in [\log_2 3,\, 30000]$. The inequality $\lceil y \rceil < y + 1$ gives

$$\left\lceil \tfrac{1}{2}x + 1 \right\rceil \;<\; \tfrac{1}{2}x + 2,$$

so it suffices to prove

$$f(x) := \tfrac{1}{2}x + 2 - 20\sqrt{x}\,\log_2 x < 0 \quad \text{for } x \in [\log_2 3,\, 30000].$$

It is easy to see that

$$f'(x) = \frac{1}{2} - \frac{10\log_2 x}{\sqrt{x}} - \frac{20\log_2 e}{\sqrt{x}}, \qquad f''(x) = \frac{5\,\log_2 x}{x^{3/2}} \;>\; 0 \quad (x > 1).$$

Hence f is convex on the entire interval. We can also compute that

$$f(\log_2 3) \approx -13.8 \;<\; 0, \qquad f(30000) \approx -3.66 \times 10^4 \;<\; 0.$$

As both the end points of the convex function are negative, we know that the function is also negative everywhere between them. So, $f(x) < 0$ for all $x \in [\log_2 3, 30000]$. Substituting $x = \log_2 k$ yields the claimed inequality. ◀

A.2 Proof of Lemma 34

Proof. Observe that

$$\sum_{i=0}^{i=q-1} |2^{S'_i}| = q \cdot 2^{q^2}, \sum_{j=0}^{j=q-1} |2^{T'_j}| = \sum_{j=0}^{j=q-1} 2^{jk}$$

Moreover, since all sets in $\mathcal{S}'$ and $\mathcal{T}'$ are subsets of $[q^2]$

$$|\mathbf{ID}(\mathcal{S}' \cup \mathcal{T}')| = 2^{q^2}$$

$$\sum_{i,j\in[0,q-1]} (|2^{S'_i} \bigcap T'_j|) = \sum_{i,j\in[0,q-1]} (|2^{[jk]}|) = q \cdot \sum_{i=0}^{i=q-1} 2^{ik}$$

Substituting these values in Observation 33, we get $|\mathbf{ID}(\mathcal{S} \cup \mathcal{T})|$

$$= \sum_{i=0}^{i=q-1} |2^{S_i}| + \sum_{j=0}^{j=q-1} |2^{T_j}| - \sum_{i,j\in[0,q-1]} (|2^{S_i} \bigcap T_j|) - q \cdot 2^{q^2} - \sum_{i=0}^{i=q-1} 2^{ik} + q \cdot \sum_{i=0}^{i=q-1} 2^{ik} + 2^{q^2}$$

$$= \sum_{i=0}^{i=q-1} |2^{S_i}| + \sum_{j=0}^{j=q-1} |2^{T_j}| - \sum_{i,j\in[0,q-1]} (|2^{S_i} \bigcap T_j|) + (q-1)(\sum_{j=0}^{j=q-1} 2^{jk} - 2^{q^2})$$

From Observation 30,

$$= \sum_{i=0}^{i=q-1} |2^{S_i}| + \sum_{j=0}^{j=q-1} |2^{T_j}| - \sum_{i,j\in[0,q-1]} 2^{jk}(2^{|F_{ij}|}) + (q-1)(\sum_{j=0}^{j=q-1} 2^{jk} - 2^{q^2}).$$ ◀

Fine-Grained Complexity Analysis of Dependency Quantified Boolean Formulas

Che Cheng
Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan

Long-Hin Fung
Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan

Jie-Hong Roland Jiang
Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan

Friedrich Slivovsky
Department of Computer Science, University of Liverpool, UK

Tony Tan
Department of Computer Science, University of Liverpool, UK

Abstract

Dependency Quantified Boolean Formulas (DQBF) extend Quantified Boolean Formulas by allowing each existential variable to depend on an explicitly specified subset of the universal variables. The satisfiability problem for DQBF is NEXP-complete in general, with only a few tractable fragments known to date. We investigate the complexity of DQBF with k existential variables (k-DQBF) under structural restrictions on the matrix – specifically, when it is in Conjunctive Normal Form (CNF) or Disjunctive Normal Form (DNF) – as well as under constraints on the dependency sets. For DNF matrices, we obtain a clear classification: 2-DQBF is PSPACE-complete, while 3-DQBF is NEXP-hard, even with disjoint dependencies. For CNF matrices, the picture is more nuanced: we show that the complexity of k-DQBF ranges from NL-complete for 2-DQBF with disjoint dependencies to NEXP-complete for 6-DQBF with arbitrary dependencies.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases Dependency quantified Boolean formulas, complexity, completeness, conjunctive normal form, disjunctive normal form

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.10

Funding We would like to acknowledge the generous support of Royal Society International Exchange Grant no. EC\R3\233183, the National Science and Technology Council of Taiwan under grant NSTC 111-2923-E-002-013-MY3, and the NTU Center of Data Intelligence: Technologies, Applications, and Systems under grant NTU-113L900903.

1 Introduction

Propositional satisfiability (SAT) solving has made significant progress over the past 30 years [4, 10]. Thanks to clever algorithms and highly optimised solvers, SAT has become a powerful tool for solving hard combinatorial problems in many areas, including verification, planning, and artificial intelligence [5]. Modern solvers can handle very large formulas efficiently, making SAT a practical choice in many settings.

However, for problems beyond NP, such as variants of reactive synthesis, direct encodings in propositional logic often grow exponentially with the input and quickly become too large to fit in memory. This has led to growing interest in more expressive logics, such as Quantified Boolean Formulas (QBF) and Dependency Quantified Boolean Formulas (DQBF) [17]. DQBF extends QBF by allowing explicit control over the dependency sets: each existential variable


© Che Cheng, Long-Hin Fung, Jie-Hong Roland Jiang, Friedrich Slivovsky, and Tony Tan;
licensed under Creative Commons License CC-BY 4.0



can be assigned its own set of universal variables it depends on. A model of a DQBF assigns to each existential variable a Skolem function that maps assignments of its dependency set to truth values. From a game-theoretic point of view, a DQBF model is a collection of sets of local strategies – one set for each existential variable – that may observe only part of the universal assignment. This makes DQBF more succinct than QBF and particularly well-suited for applications such as synthesis and verification, where components often make decisions based on partial information. Unfortunately, this added expressiveness comes at a cost: DQBF satisfiability is NEXP-complete, and only a few tractable fragments are known [7, 8, 6, 18, 15]. One notable tractable case involves CNF matrices with dependency sets that are either pairwise disjoint or identical; such formulas can be rewritten into satisfiability-equivalent Σ_3-QBFs [18].

Building on these ideas, we apply similar restrictions on the dependency sets to refine a recent classification of the complexity of DQBF with k existential variables (k-DQBF) [14]. For DNF matrices, this restriction has no effect, since the proofs in [14] for the PSPACE-hardness of 2-DQBF and NEXP-hardness of 3-DQBF can be carried over to formulas with pairwise disjoint dependency sets.

For CNF matrices, the situation is more subtle. For $k \geqslant 3$ and even non-constant k with disjoint dependencies, we extend the strategy of [18] to split clauses containing variables with incomparable dependency sets, but instead of reducing it to a QBF, we directly construct an NP algorithm to establish the NP membership. This technique can be extended to the case where any two dependency sets are either disjoint or comparable, and the size blow-up remains polynomial for constant k. The resulting DQBF only has existential variables with empty dependency sets, and its satisfiability can be checked in NP.

When arbitrary dependencies are allowed in CNF matrices, we prove that 3-DQBF is Π_2^{P}-hard. Further, a variant of Tseitin transformation lets us convert a k-DQBF with an arbitrary matrix into a $(k+3)$-DQBF with CNF matrix, yielding PSPACE-hardness of 5-DQBF and NEXP-hardness of 6-DQBF with CNF matrices.

As for the satisfiability problem of 2-DQBF, [13] shows that it reduces to detecting contradicting cycles in a succinctly represented implication graph, making it PSPACE-complete. For CNF matrices and disjoint dependencies, we show that the fully expanded graph has a simple structure, allowing satisfiability tests in NL. Consequently, the satisfiability of 2-DQBF with CNF matrices and unrestricted dependencies is in coNP – one can guess an assignment to the shared universal variables and solve the resulting instance with disjoint dependencies in NL. We also prove the NL- and coNP-hardness of the two problems via a reduction from 2-SAT and 3-DNF tautology, respectively.

Our results, summarised in Table 1, help map out the complexity of natural fragments of DQBF and show how both the formula structure and dependency restrictions play a key role in determining tractability.

2 Preliminaries

In this section, we define the notation used throughout this paper and recall the necessary technical background. All logarithms have base 2. For a positive integer m, $[m]$ denotes the set of integers $\{1, \ldots, m\}$.

Boolean values TRUE and FALSE are denoted by $\top$ and $\bot$, respectively. Boolean connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$, and $\oplus$ are interpreted as usual. A literal ℓ is a Boolean variable v or its negation $\neg v$. We write $\mathrm{var}(v) = \mathrm{var}(\neg v) = v$ for the variable of a literal and $\mathrm{sgn}(v) = \top$ and $\mathrm{sgn}(\neg v) = \bot$ to denote its sign. We also write $v \oplus \bot$ and $v \oplus \top$ to denote the literals v and $\neg v$, respectively.

Table 1 Summary of the complexity results.

k	k-DQBF$^{\mathsf{d}}_{\mathsf{cnf}}$	k-DQBF$^{\mathsf{de}}_{\mathsf{cnf}}$	k-DQBF$^{\mathsf{dec}}_{\mathsf{cnf}}$, k-DQBF$^{\mathsf{ds}}_{\mathsf{cnf}}$	k-DQBF$_{\mathsf{cnf}}$	k-DQBF$^{\mathsf{d}}_{\mathsf{dnf}}$
1	–	–	–	L (Theorem 19)	coNP-c (Theorem 4)
2	NL-c (Theorem 8)	NL-c (Corollary 17)	NL-c (Corollary 17)	coNP-c (Theorem 20)	PSPACE-c (Theorem 4)
3, 4	NP-c (Theorem 14)	NP-c (Corollary 17)	NP-c (Corollary 17)	Π^{P}_2-h (Theorem 21)	NEXP-c (Theorem 4)
5				PSPACE-h (Theorem 23)	
6+				NEXP-c (Theorem 23)	
Non-const.		Σ^{P}_3-c [18]	NEXP-c [18]		

Note: "-c" denotes "-complete", "-h" denotes "-hard", and "non-const." denotes "non-constant."

A clause is a disjunction of literals, and a cube is a conjunction of literals. A Boolean formula φ is in conjunctive normal form (CNF) if it is a conjunction of clauses and in disjunctive normal form (DNF) if it is a disjunction of cubes. We view a clause or a cube as a set of literals and a formula in CNF (respectively, DNF) as a set of clauses (respectively, cubes) whenever appropriate. We sometimes write a clause in the form of $Q \to C$, where Q is a cube and C is a clause, and similarly a DNF formula in the form of $\varphi \to \psi$, where φ is in CNF and ψ is in DNF.

We say that two sets of clauses A and B are *variable-disjoint* if for any clause $C_1 \in A$ and $C_2 \in B$, C_1 and C_2 do not share a common variable. For variable-disjoint sets A and B, we write $A \times B$ to denote the set of clauses $\{(C_1 \vee C_2) \mid C_1 \in A, C_2 \in B\}$. We generalise this notion to $A_1 \times A_2 \times \cdots \times A_n$ for pairwise variable-disjoint sets $A_1, \ldots, A_n$.

We write $\bar{v} = (v_1, \ldots, v_n)$ to denote a vector of n Boolean variables with $|\bar{v}| \coloneqq n$ denoting its length.[1] An *assignment* on $\bar{v}$ is a function from $\bar{v}$ to $\{\top, \bot\}$. We often identify an assignment on $\bar{v}$ with a vector $\bar{a} = (a_1, \ldots, a_n) \in \{\top, \bot\}^n$, denoted $\bar{a}^{\bar{v}}$, which maps each v_i to a_i. When $\bar{v} \subseteq \bar{u}$, we write $\bar{a}^{\bar{u}}(\bar{v})$ to denote the vector of Boolean values $(\bar{a}^{\bar{u}}(v))_{v \in \bar{v}}$. When $\bar{v}$ is clear from the context, we will simply write $\bar{a}$ instead of $\bar{a}^{\bar{v}}$.

Two assignments $\bar{a}^{\bar{u}}$ and $\bar{b}^{\bar{v}}$ are *consistent*, denoted by $\bar{a}^{\bar{u}} \simeq \bar{b}^{\bar{v}}$, if $\bar{a}^{\bar{u}}(v) = \bar{b}^{\bar{v}}(v)$ for every $v \in \bar{u} \cap \bar{v}$. When $\bar{a}^{\bar{u}}$ and $\bar{b}^{\bar{v}}$ are consistent, we write $(\bar{a}^{\bar{u}}, \bar{b}^{\bar{v}})$ to denote the union $\bar{a}^{\bar{u}} \cup \bar{b}^{\bar{v}}$. Given a Boolean formula ϕ over the variables $\bar{u}, \bar{v}$ and an assignment $\bar{a}^{\bar{v}}$, we denote by $\phi[\bar{a}^{\bar{v}}]$ the induced formula over the variables $\bar{u}$ obtained by assigning the variables in $\bar{v}$ with Boolean values according to the assignment $\bar{a}^{\bar{v}}$.

For a positive integer m and a vector of variables $\bar{u}$ of length $n > \log m$, by abuse of notation, we write $\bar{u} = m$ to denote the cube $\bigwedge_{i \in [n]} u_i \leftrightarrow a_i$, where $(a_1, \ldots, a_n)$ is the n-bit binary representation of m.

[1] To avoid clutter, we always assume a vector of variables $\bar{v} = (v_1, \ldots, v_n)$ do not contain duplicate entries, which can be viewed as a set $\{v_1, \ldots, v_n\}$. We will thus use set-theoretic operations on such vectors as on sets.

2.1 DQBF and Its Subclasses

We consider Dependency Quantified Boolean Formulas (DQBF) of the form

$$\Phi = \forall \bar{x}, \exists y_1(D_1), \ldots, y_k(D_k).\, \varphi\,, \tag{1}$$

where $\bar{x} = (x_1, \ldots, x_n)$, $D_i \subseteq \bar{x}$ is the *dependency set* of the existential variable y_i for every $i \in [k]$, and φ is a quantifier-free Boolean formula over the variables $\bar{x} \cup \bar{y}$ called the *matrix* of Φ. We write $\mathrm{dep}(v) \coloneqq D_i$ if $v = y_i$ and $\mathrm{dep}(v) \coloneqq \{x_i\}$ if $v = x_i$. We extend this notation to literals and clauses by letting $\mathrm{dep}(\ell) \coloneqq \mathrm{dep}(\mathrm{var}(\ell))$ for a literal ℓ and $\mathrm{dep}(C) \coloneqq \bigcup_{\ell \in C} \mathrm{dep}(\ell)$ for a clause C.

We say that Φ is satisfiable if for every $i \in [k]$ there is a Boolean formula f_i using only variables in D_i such that by replacing each y_i with f_i, the formula φ becomes a tautology. In this case, we call the sequence $f_1, \ldots, f_k$ a model of Φ and refer to each individual f_i as a Skolem function for y_i.

We define the subclasses $k\text{-DQBF}^{\alpha}_{\beta}$ of DQBF, where $k \geqslant 1$ indicates the number of existential variables, $\alpha \in \{\mathsf{d}, \mathsf{de}, \mathsf{dec}, \mathsf{ds}\}$ indicates the condition on the dependency sets, and $\beta \in \{\mathsf{cnf}, \mathsf{dnf}\}$ indicates the form of the matrix.

For the dependency set annotation α, we define:
DQBF$^{\mathsf{d}}$ For every $i \neq j$, $D_i \cap D_j = \emptyset$,
DQBF$^{\mathsf{de}}$ For every $i \neq j$, $D_i \cap D_j = \emptyset$ or $D_i = D_j$,
DQBF$^{\mathsf{dec}}$ For every $i \neq j$ with $|D_i| \leqslant |D_j|$, $D_i \cap D_j = \emptyset$, $D_i = D_j$, or $D_j = \bar{x}$, and
DQBF$^{\mathsf{ds}}$ For every $i \neq j$ with $|D_i| \leqslant |D_j|$, $D_i \cap D_j = \emptyset$ or $D_i \subseteq D_j$.
The letters d, e, c, and s denote *disjoint*, *equal*, *complete*, and *subset*, respectively. Note that the dependency sets of a $\text{DQBF}^{\mathsf{ds}}$ formula form a *laminar set family*. The classification of different dependency structures is inspired by [18], but we specify the condition that the formula is in CNF explicitly in our notation. That is, $\text{DQBF}^{\mathrm{de}}$ and $\text{DQBF}^{\mathrm{dec}}$ in [18] correspond to $\text{DQBF}^{\mathsf{de}}_{\mathsf{cnf}}$ and $\text{DQBF}^{\mathsf{dec}}_{\mathsf{cnf}}$ in our notation, respectively.

Note that $\text{DQBF}^{\mathsf{d}} \subseteq \text{DQBF}^{\mathsf{de}} \subseteq \text{DQBF}^{\mathsf{dec}} \subseteq \text{DQBF}^{\mathsf{ds}}$. The first two inclusions are trivial, and the last one comes from the observation that both $D_i = D_j$ and $D_j = \bar{x}$ are special cases of $D_i \subseteq D_j$.

When k, α, or β is missing, it means that the corresponding restriction is dropped. For instance, $3\text{-DQBF}_{\mathsf{dnf}}$ denotes the class of DQBF with 3 existential variables, arbitrary dependency structure, and matrix in DNF, while DQBF^{d} denotes the class of DQBF with the dependency structure specified by d and an arbitrary Boolean formula as the matrix. We denote by $\mathsf{sat}(k\text{-DQBF}^{\alpha}_{\beta})$ the satisfiability problem for the class $k\text{-DQBF}^{\alpha}_{\beta}$.

▶ Remark 1. For every $\alpha \in \{\mathsf{d}, \mathsf{de}, \mathsf{dec}, \mathsf{ds}\}$ and $\beta \in \{\mathsf{cnf}, \mathsf{dnf}\}$, checking whether a DQBF formula Φ is in the class $\text{DQBF}^{\alpha}_{\beta}$ can be done deterministically in space logarithmic in the length of Φ. To do so, we iterate through all the variables to check whether it satisfies the conditions set by α. In each iteration, it suffices to store $O(1)$ number of indices of the variables, which requires only logarithmic space.

2.2 Manipulation of $\text{DQBF}_{\mathsf{cnf}}$

We recall two operations for manipulating $\text{DQBF}_{\mathsf{cnf}}$ formulas, namely *universal reduction* [3, 12] and *resolution-based variable elimination* [20].

▶ **Lemma 2** (Universal reduction [3, 12]). *Let $\Phi = \forall \bar{x}, \exists y_1(D_1), \ldots, \exists y_k(D_k).\, \varphi$ be a $\text{DQBF}_{\mathsf{cnf}}$ formula, $C \in \varphi$ be a clause, $\ell \in C$ be a universal literal, and let $C' \coloneqq C \setminus \{\ell\}$. If $\ell \notin \mathrm{dep}(C')$, then Φ is equisatisfiable to*

$$\Phi' \coloneqq \forall \bar{x}, \exists y_1(D_1), \ldots, \exists y_k(D_k).\, \varphi \cup \{C'\} \setminus \{C\}\,.$$

Using universal reduction, we assume that $\bigcup_{i\in[k]} D_i = \bar{x}$ for every $\mathrm{DQBF}_{\mathsf{cnf}}$ formula, since any universal variable not in $\bigcup_{i\in[k]} D_i$ can be universally-reduced from every clause.

For variable elimination by resolution, we only need a weaker version, which is sufficient for our purpose.

▶ **Lemma 3** (Variable elimination by resolution [20]). *Let* $\Phi = \forall\bar{x}, \exists y_1(D_1), \ldots, \exists y_k(D_k).\, \varphi$ *be a* $\mathrm{DQBF}_{\mathsf{cnf}}$ *formula. We partition* φ *into three sets:*

- $\varphi^{y_1} \coloneqq \{C \in \varphi \mid y_1 \in C\}$,
- $\varphi^{\neg y_1} \coloneqq \{C \in \varphi \mid \neg y_1 \in C\}$, *and*
- $\varphi^{\emptyset} \coloneqq \varphi \setminus (\varphi^{y_1} \cup \varphi^{\neg y_1})$.

If for every $C \in \varphi^{y_1}$ *we have* $\mathrm{dep}(C) \subseteq \mathrm{dep}(y_1)$, *or for every* $C \in \varphi^{\neg y_1}$ *we have* $\mathrm{dep}(C) \subseteq \mathrm{dep}(y_1)$, *then* Φ *is equisatisfiable to*

$$\forall\bar{x}, \exists y_2(D_2), \ldots, \exists y_k(D_k).\, \varphi^{\emptyset} \cup \{C \otimes_{y_1} C' \mid C \in \varphi^{y_1}, C' \in \varphi^{\neg y_1}\},$$

where $C \otimes_v C'$ *denotes the resolution of* C *and* C' *w.r.t. the pivot* v, *i.e.,* $C \otimes_v C' = (C \setminus \{v\}) \cup (C' \setminus \{\neg v\})$.

The intuition is that y_1 can "see" every assignment that may force it to be assigned to $\top$ (respectively, $\bot$), and thus if all resolvents are satisfied, then there must be a Skolem function for y_1 that satisfies the clauses in $\varphi^{y_1} \cup \varphi^{\neg y_1}$. Note that the number of clauses after removing y_1 is at most $|\varphi|^2$.

2.3 Universal Expansion of k-DQBF

Consider a k-DQBF formula $\Phi \coloneqq \forall\bar{x}, \exists y_1(D_1), \ldots, \exists y_k(D_k).\, \varphi$. Let $\bar{y} = (y_1, \ldots, y_k)$. Given an assignment $\bar{a}$ on $\bar{x}$ and $\bar{b}$ on $\bar{y}$, for every $i \in [k]$, let $\bar{a}_i$ be the restriction of $\bar{a}$ to D_i and b_i be the restriction of $\bar{b}$ to y_i. We can expand Φ into an equisatisfiable k-CNF formula $\exp(\Phi)$ by instantiating each y_i into exponentially many *instantiated variables* of the form $Y_{i,\bar{a}_i}$ [2, 6, 12]. Formally,

$$\exp(\Phi) \coloneqq \bigwedge_{(\bar{a},\bar{b}):\varphi[\bar{a},\bar{b}]=\bot} \mathcal{C}_{\bar{a},\bar{b}},$$

where $\mathcal{C}_{\bar{a},\bar{b}} \coloneqq \bigvee_{i\in[k]} Y_{i,\bar{a}_i} \oplus b_i$. Intuitively, in the expansion $\exp(\Phi)$, the Boolean variable $Y_{i,\bar{a}_i}$ represents the value of a candidate Skolem function $f_i(\bar{a}_i)$ for y_i. The universal expansion shows that the satisfiability of Φ can be reduced to a Boolean satisfiability problem (with exponential blow-up). Moreover, if the assignment $(\bar{a}, \bar{b})$ falsifies the matrix φ, then a satisfying assignment of $\exp(\Phi)$ must assign $Y_{i,\bar{a}_i}$ to $\neg b_i$ for some $i \in [k]$.

3 Complexity of sat(k-DQBF$^{\mathsf{d}}_{\mathsf{dnf}}$)

Having defined the various subclasses of DQBF, we can refine previous results by stating them more precisely. In this section, we consider the case where the matrix is in DNF.

By combining the DNF version of Tseitin transformation [9, Proposition 1] and the results in [14], we can show that restricting to DNF matrix and pairwise-disjoint dependency sets does not affect the complexity of $\mathsf{sat}(k\text{-DQBF})$.

▶ **Theorem 4.** $\mathsf{sat}(k\text{-DQBF}^{\mathsf{d}}_{\mathsf{dnf}})$ *is* coNP-, PSPACE-, *and* NEXP-*complete when* $k = 1$, $k = 2$, *and* $k \geqslant 3$, *respectively.*

See the appendix for the detailed proof. Since $k\text{-DQBF}^{\mathsf{d}}_{\mathsf{dnf}} \subseteq k\text{-DQBF}^{\mathsf{de}}_{\mathsf{dnf}} \subseteq k\text{-DQBF}^{\mathsf{dec}}_{\mathsf{dnf}} \subseteq k\text{-DQBF}^{\mathsf{ds}}_{\mathsf{dnf}} \subseteq k\text{-DQBF}_{\mathsf{dnf}} \subseteq k\text{-DQBF}$, we have the following corollary:

▶ **Corollary 5.** $\mathsf{sat}(k\text{-DQBF}^{\alpha}_{\mathsf{dnf}})$ *and* $\mathsf{sat}(k\text{-DQBF}_{\mathsf{dnf}})$ *is* coNP-, PSPACE-, *and* NEXP-*complete when* $k = 1$, $k = 2$, *and* $k \geqslant 3$, *respectively, for every* $\alpha \in \{\mathsf{de}, \mathsf{dec}, \mathsf{ds}\}$.

4 Complexity of $\mathsf{sat}(k\text{-DQBF}^{\alpha}_{\mathsf{cnf}})$

In this section, we consider the complexity of $\mathsf{sat}(k\text{-DQBF}^{\alpha}_{\mathsf{cnf}})$ and $\mathsf{sat}(\text{DQBF}^{\alpha}_{\mathsf{cnf}})$, with a focus on the case where $\alpha = \mathsf{d}$. We first prove an important property of the expansion of $\text{DQBF}^{\mathsf{d}}_{\mathsf{cnf}}$ formulas in Section 4.1 Then, in Sections 4.2 and 4.3 we show that $\mathsf{sat}(k\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$ is of the same complexity as k-SAT for $k \geqslant 2$,[2] and that $\mathsf{sat}(\text{DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$ is of the same complexity as SAT. This shows that, in stark contrast to the DNF case in the previous section, with pairwise disjoint dependency sets and with CNF matrix, the exponential gap between SAT and DQBF disappears. Finally, we discuss other dependency structures in Section 4.4.

4.1 Universal Expansion of $\text{DQBF}^{\mathsf{d}}_{\mathsf{cnf}}$

In this section, we show a useful property of the expansion of $\text{DQBF}^{\mathsf{d}}_{\mathsf{cnf}}$ formulas. We fix a $k\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}}$ formula:

$$\Phi = \forall \bar{x}, \exists y_1(D_1), \ldots, \exists y_k(D_k).\ \bigwedge_{j \in [m]} C_j. \tag{2}$$

Let $\bar{y} = (y_1, \ldots, y_k)$. Given an assignment $\bar{a}$ on $\bar{x}$ and $\bar{b}$ on $\bar{y}$, for every $i \in [k]$, let $\bar{a}_i$ be the restriction of $\bar{a}$ to D_i and b_i be the restriction of $\bar{b}$ to y_i.

Recall that for a DQBF formula Φ, each instantiated clause in $\exp(\Phi)$ corresponds to a falsifying assignment of the matrix of Φ. For a formula in CNF, the set of falsifying assignments can be represented by the union of the set of falsifying assignments of each individual clause. This allows us to represent the instantiated clauses in $\exp(\Phi)$ as the union of polynomially many sets when Φ is a $\text{DQBF}^{\mathsf{d}}_{\mathsf{cnf}}$ formula. Moreover, the disjoint dependency structure allows us to further represent each of these sets as the Cartesian product of variable-disjoint sets of instantiated literals. To formally state the property, we first define some notation.

For a clause C_j in Φ, we write $C_j^i(\Phi)$ to denote the subset of C_j within y_i's dependency set, $\mathcal{L}_{i,j}(\Phi)$ the set of instantiated literals $Y_{i,\bar{a}_i} \oplus b_i$ where the assignment $(\bar{a}_i, b_i)$ falsifies C_j^i, and $\mathfrak{C}_j(\Phi)$ the set of instantiated clauses $\mathcal{C}_{\bar{a},\bar{b}}$ where $(\bar{a}, \bar{b})$ falsifies $\neg C_j$. We now formally define these sets.

▶ **Definition 6.** *Let* Φ *be a* $k\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}}$ *formula as in* (2). *For every* $j \in [m]$ *and* $i \in [k]$, *we define the sets* $C_j^i(\Phi)$, $\mathcal{L}_{i,j}(\Phi)$ *and* $\mathfrak{C}_j(\Phi)$*:*

- $C_j^i(\Phi) \coloneqq \{\ell \in C_j \mid \mathrm{var}(\ell) \in D_i \cup \{y_i\}\}$.
- $\mathcal{L}_{i,j}(\Phi) \coloneqq \{Y_{i,\bar{a}_i} \oplus b_i \mid (\bar{a}_i, b_i) \simeq \neg C_j^i\}$.
- $\mathfrak{C}_j(\Phi) \coloneqq \{\mathcal{C}_{\bar{a},\bar{b}} \mid (\bar{a}, \bar{b}) \simeq \neg C_j\}$.

When Φ *is clear from the context, we simply write* C_j^i, $\mathcal{L}_{i,j}$ *and* $\mathfrak{C}_j$.

We remark that $(\bar{a}, \bar{b}) \simeq \neg C_j$ if and only if $(\bar{a}, \bar{b})$ falsifies C_j, and similarly $(\bar{a}_i, b_i) \simeq \neg C_j^i$ if and only if $(\bar{a}_i, b_i)$ falsifies C_j^i. Note also that $\exp(\Phi) = \bigwedge_{j \in [m]} \bigwedge_{C \in \mathfrak{C}_j} C$ and that the sets $\mathcal{L}_{1,j}, \ldots, \mathcal{L}_{k,j}$ are pairwise variable-disjoint.

[2] There is no dependency structure for $k = 1$.

We now state the property formally.

▶ **Lemma 7.** *Let Φ be as in Equation (2). For every $j \in [m]$, $\mathfrak{C}_j = \mathcal{L}_{1,j} \times \cdots \times \mathcal{L}_{k,j}$.*

Proof. We fix an arbitrary $j \in [m]$. We first prove the "$\subseteq$" direction. Let $\mathcal{C}_{\bar{a},\bar{b}}$ be a clause in $\mathfrak{C}_j$. That is, $(\bar{a},\bar{b})$ is an assignment that falsifies C_j. Let $\bar{a}_i$ be the restriction of $\bar{a}$ on D_i and b_i be the restriction of $\bar{b}$ on y_i, for every $i \in [k]$. By definition, $\mathcal{C}_{\bar{a},\bar{b}} = \bigvee_{i\in[k]} Y_{i,\bar{a}_i} \oplus b_i$. Since $(\bar{a},\bar{b})$ falsifies C_j, it is consistent with the cube $\neg C_j$. Hence, for every $i \in [k]$, each $\bar{a}_i, b_i$ is consistent with the cube $\neg C^i_j$. By definition, the literal $Y_{i,\bar{a}_i} \oplus b_i$ belongs to $\mathcal{L}_{i,j}$, for every $i \in [k]$.

Next, we prove the "$\supseteq$" direction. Let $C \coloneqq (L_1 \vee \cdots \vee L_k) \in \mathcal{L}_{1,j} \times \cdots \times \mathcal{L}_{k,j}$. By definition, for every $i \in [k]$, there is assignment $(\bar{a}_i, b_i)$ such that L_i is the literal $Y_{i,\bar{a}_i} \oplus b_i$ and $(\bar{a}_i, b_i)$ is consistent with the cube $\neg C^i_j$. Due to the disjointness of the dependency sets, all the assignments $(\bar{a}_i, b_i)$'s are pairwise consistent. Let $(\bar{a},\bar{b})$ be their union $\bigcup_{i\in[k]}(\bar{a}_i, b_i)$.[3] Since each $(\bar{a}_i, b_i)$ is consistent with $\neg C^i_j$, $(\bar{a},\bar{b})$ is consistent with all of $\neg C^1_j, \ldots, \neg C^k_j$. Therefore, $(\bar{a},\bar{b})$ is a falsifying assignment of C_j. By definition, the clause $\mathcal{C}_{\bar{a},\bar{b}} = \bigvee_{i\in[k]} Y_{i,\bar{a}_i} \oplus b_i$ is in $\mathfrak{C}_j$. ◀

4.2 2-DQBF$^{\mathsf{d}}_{\mathsf{cnf}}$

In this section we will show that $\mathsf{sat}(2\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$ is NL-complete, as stated formally in Theorem 8.

▶ **Theorem 8.** $\mathsf{sat}(2\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$ *is* NL*-complete.*

Before we proceed to the formal proof, we first review some notation and terminology. Recall that the expansion of a 2-DQBF formula (even when the matrix is in an arbitrary form) is a 2-CNF formula, which can be viewed as a directed graph, called the *implication graph* (of the 2-CNF formula) [1]. The vertices in the implication graph are the literals, and for every clause $(\ell \vee \ell')$ in the formula, there are two edges, $(\neg\ell \to \ell')$ and $(\neg\ell' \to \ell)$.

The following notion of a disimplex will be useful.

▶ **Definition 9** (Disimplex [11])**.** *Given two sets of vertices $\mathcal{A}, \mathcal{B}$, the* disimplex *from $\mathcal{A}$ to $\mathcal{B}$ is the directed graph $K(\mathcal{A},\mathcal{B}) \coloneqq (\mathcal{A} \cup \mathcal{B}, \mathcal{A} \times \mathcal{B})$.*

In other words, a disimplex $K(\mathcal{A},\mathcal{B})$ is a complete directed bipartite graph where all the edges are oriented from $\mathcal{A}$ to $\mathcal{B}$.

The rest of this subsection is devoted to the proof of Theorem 8. For the rest of this subsection, we fix a 2-DQBF$^{\mathsf{d}}_{\mathsf{cnf}}$ formula $\Phi = \forall \bar{z}_1, \bar{z}_2, \exists y_1(\bar{z}_1), \exists y_2(\bar{z}_2).\ \bigwedge_{j\in[m]} C_j$. We will simply write C^i_j, $\mathcal{L}_{i,j}$ and $\mathfrak{C}_j$ to denote the sets $C^i_j(\Phi)$, $\mathcal{L}_{i,j}(\Phi)$ and $\mathfrak{C}_j(\Phi)$ defined in Definition 6. For a set $\mathcal{L}$ of literals, we denote by $\widehat{\mathcal{L}}$ the set of negated literals in $\mathcal{L}$, i.e., $\widehat{\mathcal{L}} \coloneqq \{\neg L \mid L \in \mathcal{L}\}$.

We first show that the implication graph of $\exp(\Phi)$ is a finite union of disimplices, and that the length of any shortest path between two vertices is bounded above by $2m$.

▶ **Lemma 10.** *Let $G = (\mathcal{V},\mathcal{E})$ be the implication graph of* $\exp(\Phi)$*. The set of edges $\mathcal{E}$ can be represented as*

$$\mathcal{E} = \bigcup_{j\in[m]} (\widehat{\mathcal{L}}_{1,j} \times \mathcal{L}_{2,j}) \cup (\widehat{\mathcal{L}}_{2,j} \times \mathcal{L}_{1,j}),$$

[3] Note that, as stated in Section 2.2, we assume that $\bigcup_{i\in[k]} D_i = \bar{x}$.

which is the union of the edge sets of m pairs of disimplices. Moreover, for every two vertices $L, L' \in \mathcal{V}$, if L' is reachable from L, then there exists a path from L to L' of length at most $2m$.

Proof. By definition,

$$\mathcal{E} = \left\{(\neg Y_{1,\bar{z}_1} \oplus b_1, Y_{2,\bar{z}_2} \oplus b_2), (\neg Y_{2,\bar{z}_2} \oplus b_2, Y_{1,\bar{z}_1} \oplus b_1) \,\middle|\, \varphi[\bar{a}_1^{\bar{z}_1}, \bar{a}_2^{\bar{z}_2}, b_1^{y_1}, b_2^{y_2}] = \bot\right\}.$$

Since any assignment that falsifies φ must falsify some clause C_j in φ, we have

$$\mathcal{E} = \bigcup_{j\in[m]} \bigcup_{\mathcal{C}_{\bar{a},\bar{b}} \in \mathfrak{C}_j} \{(\neg Y_{1,\bar{z}_1} \oplus b_1, Y_{2,\bar{z}_2} \oplus b_2), (\neg Y_{2,\bar{z}_2} \oplus b_2, Y_{1,\bar{z}_1} \oplus b_1)\}.$$

By Lemma 7, we have $\mathfrak{C}_j = \{(L_1 \vee L_2) \mid L_1 \in \mathcal{L}_{1,j}, L_2 \in \mathcal{L}_{2,j}\}$ for every $j \in [m]$. Therefore,

$$\mathcal{E} = \bigcup_{j\in[m]} \left(\widehat{\mathcal{L}}_{1,j} \times \mathcal{L}_{2,j}\right) \cup \left(\widehat{\mathcal{L}}_{2,j} \times \mathcal{L}_{1,j}\right).$$

For the second part of the proof, assume, for the sake of contradiction, that $P = (L_0, \ldots, L_n)$ is a shortest path from L to L' with $n > 2m$. Then, by the pigeonhole principle, there must be some $0 \leqslant i_1 < i_2 < n$ such that (L_{i_1}, L_{i_1+1}) and (L_{i_2}, L_{i_2+1}) belongs to the same disimplex $K \subseteq \mathcal{E}$, and thus $(L_{i_1}, L_{i_2+1}) \in K \subseteq \mathcal{E}$. We can then construct a shorter path $P' = (L_0, \ldots, L_{i_1}, L_{i_2+1}, \ldots, L_n)$ from L to L', which contradicts with the assumption that P is a shortest path. ◀

Proof of Theorem 8. For the NL membership, we devise an algorithm by checking the unsatisfiability of $\exp(\Phi)$ directly on these disimplices. We present an NL algorithm that checks the unsatisfiability of $\exp(\Phi)$ by looking for cycles containing both an instantiated literal and its negation in the implication graph $G = (\mathcal{V}, \mathcal{E})$ of $\exp(\Phi)$.[4]

A naïve idea is to first non-deterministically guess a literal L and the paths P from L to $\neg L$ and P' from $\neg L$ to L. However, since $|\mathcal{V}|$ is exponential in $|\bar{x}|$, representing a literal $L \in \mathcal{V}$ takes linear space. We instead make use of Lemma 10 and guess the disimplex each edge of P, P' belongs in, denoted by the sequences $(K(\mathcal{A}_1, \mathcal{B}_1), \ldots, K(\mathcal{A}_n, \mathcal{B}_n))$ and $(K(\mathcal{A}'_1, \mathcal{B}'_1), \ldots, K(\mathcal{A}'_{n'}, \mathcal{B}'_{n'}))$ with $n, n' \in [2m]$, where each $\mathcal{A}, \mathcal{B}$ is of the form $\mathcal{L}_{i,j}$ or $\widehat{\mathcal{L}}_{i,j}$. We then check if

- for every step $j \in [n-1]$, whether there exists some $L_j \in \mathcal{B}_j \cap \mathcal{A}_{j+1}$,
- for every step $j' \in [n'-1]$, whether there exists some $L'_{j'} \in \mathcal{B}'_{j'} \cap \mathcal{A}'_{j'+1}$, and
- whether there exists some $L_0 \in \mathcal{A}_1 \cap \widehat{\mathcal{B}}_n \cap \widehat{\mathcal{A}}'_1 \cap \mathcal{B}'_{n'}$.

We reject if one of the checks fails, and accept if all checks succeed. In the latter case, there are paths $P = (L_0, L_1, \ldots, L_{n-1}, \neg L_0)$ and $P' = (\neg L_0, L'_1, L'_2, \ldots, L'_{n'-1}, L_0)$.

In particular, $\mathcal{L}_{i,j} \cap \mathcal{L}_{i',j'}$ is non-empty if and only if $i = i'$ and C_j^i and $C_{j'}^{i'}$ are consistent. The consistency check can be done by keeping two pointers to the position in the clause using $\log(|\bar{x}| + 2)$ bits per pointer. This can easily be generalised to check the intersection of any constant number of $\mathcal{L}_{i,j}$'s. For $\widehat{\mathcal{L}}_{i,j}$, simply replace C_j^i with the clause $\hat{C}_j^i$ with the sign of y_i flipped if a literal of y_i is present, i.e.,

$$\hat{C}_j^i \coloneqq \left(C_j^i \setminus \{y_i, \neg y_i\}\right) \cup \left(\neg C_j^i \cap \{y_i, \neg y_i\}\right).$$

[4] Recall that a 2-SAT formula φ is unsatisfiable if and only if there is a cycle containing both a literal and its negation in the implication graph of φ.

For the hardness proof, we provide a reduction from 2-SAT to 2-DQBF$^{\mathsf{d}}_{\mathsf{cnf}}$. Let $\varphi = \bigwedge_{j\in[m]}(\ell_{j,1} \vee \ell_{j,2})$ be a 2-CNF formula over the variables $\bar{x} = (x_1, \ldots, x_n)$. The idea is to encode the assignment of $\bar{x}$ with a function $f : [n] \to \{\bot, \top\}$ where $f(i)$ represents the assignment of x_i.

In the following, for a literal ℓ, let $\mathrm{ind}(\ell)$ denote the index i where $x_i = \mathrm{var}(\ell)$. We construct the 2-DQBF$^{\mathsf{d}}_{\mathsf{cnf}}$ formula

$$\Psi := \forall \bar{u}_1, \forall \bar{u}_2, \exists y_1(\bar{u}_1), \exists y_2(\bar{u}_2).\, \psi\,,$$

where $\bar{u}_1$ and $\bar{u}_2$ have length $O(\log n)$ for representing the variables in $\bar{x}$ and ψ is a CNF formula that states the following.

- $((\bar{u}_1 = i) \wedge (\bar{u}_2 = i)) \to (y_1 \leftrightarrow y_2)$ for every $i \in [n]$.
- $((\bar{u}_1 = \mathrm{ind}(\ell_{j,1})) \wedge (\bar{u}_2 = \mathrm{ind}(\ell_{j,2}))) \to ((y_1 \leftrightarrow \mathrm{sgn}(\ell_{j,1})) \vee (y_2 \leftrightarrow \mathrm{sgn}(\ell_{j,2})))$ for every $j \in [m]$.

The first item states that the Skolem functions for y_1 and y_2 must be the same. The second item implies that $\bar{a}^{\bar{x}}$ is a satisfying assignment of φ if and only if the function $f_{\bar{a}}$ is a Skolem function for Ψ by encoding the assignment $\bar{a}^{\bar{x}}$ as a function $f_{\bar{a}} \colon [n] \to \{\top, \bot\}$, where $f_{\bar{a}}(i) = \bar{a}^{\bar{x}}(x_i)$. ◀

4.3 k-DQBF$^{\mathsf{d}}_{\mathsf{cnf}}$: $k \geqslant 3$ and Non-Constant k

For $k \geqslant 3$ and even arbitrary DQBF$^{\mathsf{d}}_{\mathsf{cnf}}$, we show that it is NP-complete. Let Φ be as in Equation (2). To show the NP membership, we first show that for every $j \in [m]$, some y_i is responsible for satisfying all the clauses in $\mathfrak{C}_j$.

▶ **Lemma 11.** *Let Φ be as in Equation (2) and let $\bar{Y}$ be the vector of variables in $\exp(\Phi)$. For every $j \in [m]$ and every assignment $\bar{a}$ on $\bar{Y}$, $\bar{a}$ satisfies the CNF formula $\bigwedge_{\mathcal{C}\in\mathfrak{C}_j} \mathcal{C}$ if and only if $\bar{a}$ satisfies the cube $\bigwedge_{L\in\mathcal{L}_{i,j}} L$ for some $i \in [k]$.*

Proof. We first prove the "if" direction. Let $\bar{a}$ be an assignment on $\bar{Y}$. If $\bar{a}$ satisfies the cube $\bigwedge_{L\in\mathcal{L}_{i,j}} L$, then, for every clause $\mathcal{C} \in \mathfrak{C}_j$, by Lemma 7, there exists some $L \in \mathcal{L}_{i,j} \cap \mathcal{C}$ that is satisfied by $\bar{a}$. Thus, $\mathcal{C}$ is satisfied by L.

For the "only if" direction, assume that $\bar{a}$ does not satisfy the cube $\bigwedge_{L\in\mathcal{L}_{i,j}} L$ for every $i \in [k]$. That is, for every $i \in [k]$, there exists some $L_i \in \mathcal{L}_{i,j}$ such that $\bar{a}(\mathrm{var}(L_i)) \neq \mathrm{sgn}(L_i)$. It follows that the clause $\left(\bigvee_{i\in[k]} L_i\right) \in \mathfrak{C}_j$ is falsified by $\bar{a}$, and thus $\bar{a}$ does not satisfy $\bigwedge_{\mathcal{C}\in\mathfrak{C}_j} \mathcal{C}$. ◀

▶ Remark 12. Recall that $\exp(\Phi) = \bigwedge_{j\in[m]} \bigwedge_{\mathcal{C}\in\mathfrak{C}_j} \mathcal{C}$. Thus, Lemma 11 can be reformulated as follows. For every assignment $\bar{a}^{\bar{Y}}$, $\bar{a}^{\bar{Y}}$ satisfies $\exp(\Phi)$ if and only if there is a function $\xi : [m] \to [k]$ such that for every $j \in [m]$, $\bar{a}^{\bar{Y}}$ satisfies the cube $\bigwedge_{L\in\mathcal{L}_{\xi(j),j}} L$. Intuitively, the function ξ is the mapping that maps index j to index i in the statement in Lemma 11. This formulation will be useful later on.

The next lemma shows the NP membership of $\mathsf{sat}(\mathrm{DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$.

▶ **Lemma 13.** $\mathsf{sat}(\mathrm{DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$ *is in* NP.

Proof. Consider a DQBF$^{\mathsf{d}}_{\mathsf{cnf}}$ formula:

$$\Phi = \forall \bar{z}_1, \ldots, \forall \bar{z}_k, \exists y_1(\bar{z}_1), \ldots, \exists y_k(\bar{z}_k).\ \bigwedge_{j\in[m]} C_j$$

with k existential variables and m clauses.

By the reformulation of Lemma 11 in Remark 12, an assignment $\bar{a}$ on $\bar{Y}$ satisfies $\exp(\Phi)$ if and only if there exists a mapping $\xi\colon [m] \to [k]$ such that $\bar{a}^{\bar{Y}}$ satisfies $\bigwedge_{j\in[m]} \bigwedge_{L\in\mathcal{L}_{\xi(j),j}} L$, or equivalently, if there exists a partition $\{S_i\}_{i\in[k]}$ of $[m]$ such that for each $i \in [k]$, the following QBF Φ_i is satisfiable:

$$\Phi_i = \forall \bar{z}_i, \exists y_i.\ \bigwedge_{j\in S_i} C^i_j\,.$$

Note that since Φ_i contains only one existential variable and it depends on all universal variables, checking the satisfiability of Φ_i is in P using Lemma 3.[5] An NP algorithm guesses the partition $\{S_i\}_{i\in[k]}$ and verifies that Φ_i is satisfiable for every $i \in [k]$. ◀

▶ **Theorem 14.** $\mathsf{sat}(k\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$ *for every* $k \geqslant 3$ *and* $\mathsf{sat}(\text{DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$ *are* NP*-complete.*

Proof. By Lemma 13, $\mathsf{sat}(\text{DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$ is in NP. Since $k\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}} \subseteq \text{DQBF}^{\mathsf{d}}_{\mathsf{cnf}}$, $\mathsf{sat}(k\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$ is also in NP for every constant k.

For the NP-hardness, a reduction from 3-SAT to $3\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}}$ can be done analogous to that of Theorem 8. A complete proof can be found in Section B.2. Since adding more existential variables only increases the complexity, $\mathsf{sat}(k\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$ for every $k \geqslant 3$ and $\mathsf{sat}(\text{DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$ are also NP-hard. ◀

4.4 k-DQBF$^{\alpha}_{\mathsf{cnf}}$: Different Dependency Structure

It has been shown in [18] that $\mathsf{sat}(\text{DQBF}^{\mathsf{de}}_{\mathsf{cnf}})$ is Σ^{P}_3-complete and $\mathsf{sat}(\text{DQBF}^{\mathsf{dec}}_{\mathsf{cnf}})$ is NEXP-complete. Since $\text{DQBF}^{\mathsf{dec}}_{\mathsf{cnf}} \subseteq \text{DQBF}^{\mathsf{ds}}_{\mathsf{cnf}} \subseteq \text{DQBF}$ and $\mathsf{sat}(\text{DQBF})$ is also NEXP-complete, we know $\mathsf{sat}(\text{DQBF}^{\mathsf{ds}}_{\mathsf{cnf}})$ is NEXP-complete. In this section, we show a surprising result that, when k is a constant, $\mathsf{sat}(k\text{-DQBF}^{\alpha}_{\mathsf{cnf}})$ has the same complexity as k-SAT and $\mathsf{sat}(k\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$ for every $\alpha \in \{\mathsf{de}, \mathsf{dec}, \mathsf{ds}\}$. Since $k\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}} \subseteq k\text{-DQBF}^{\mathsf{de}}_{\mathsf{cnf}} \subseteq k\text{-DQBF}^{\mathsf{dec}}_{\mathsf{cnf}} \subseteq k\text{-DQBF}^{\mathsf{ds}}_{\mathsf{cnf}}$, it suffices to show the results for $\mathsf{sat}(k\text{-DQBF}^{\mathsf{ds}}_{\mathsf{cnf}})$.

We start with $\mathsf{sat}(2\text{-DQBF}^{\mathsf{ds}}_{\mathsf{cnf}})$.

▶ **Theorem 15.** $\mathsf{sat}(2\text{-DQBF}^{\mathsf{ds}}_{\mathsf{cnf}})$ *is* NL*-complete.*

Proof. Since $2\text{-DQBF}^{\mathsf{ds}}_{\mathsf{cnf}} \supseteq 2\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}}$, the hardness follows from Theorem 8. For NL membership, consider a $2\text{-DQBF}^{\mathsf{ds}}_{\mathsf{cnf}}$ formula $\Phi \coloneqq \forall \bar{x}, \exists y_1(D_1), \exists y_2(D_2).\, \varphi$. First, we check whether D_1 and D_2 are disjoint using only logarithmic space. (See Remark 1.) If D_1 and D_2 are disjoint, we use the algorithm from Theorem 8 to determine its satisfiability. Otherwise, without loss of generality, we may assume that $D_1 \subseteq D_2$. We will show that this case can be decided in deterministic logarithmic space. Indeed, in this case Φ is a standard QBF and we can perform a level-ordered Q-resolution proof [16]. Since there are only two existential variables, any proof uses at most four clauses, and we can simply iterate through all 4-tuples of clause indices and check whether Q-resolution can be performed.

In the following, we give an alternative proof that works directly on the semantics of QBF. To ease notation, we write $\bar{z}_1 \coloneqq D_1$, $\bar{z}_2 \coloneqq D_2 \setminus D_1$, and $\bar{z}_3 \coloneqq \bar{x} \setminus D_2$. Note that Φ is equivalent to a QBF

$$\begin{aligned}
\Psi &= \forall \bar{z}_1, \exists y_1, \forall \bar{z}_2, \exists y_2, \forall \bar{z}_3.\, \varphi \\
&= \forall \bar{z}_1, \exists y_1, \forall \bar{z}_2.\, (\forall \bar{z}_3.\, \varphi[\bot^{y_2}]) \vee (\forall \bar{z}_3.\, \varphi[\top^{y_2}]) \\
&= \forall \bar{z}_1.\, (\forall \bar{z}_2.\, (\forall \bar{z}_3.\, \varphi[(\bot^{y_1}, \bot^{y_2})]) \vee (\forall \bar{z}_3.\, \varphi[(\bot^{y_1}, \top^{y_2})])) \\
&\qquad \vee (\forall \bar{z}_2.\, (\forall \bar{z}_3.\, \varphi[(\top^{y_1}, \bot^{y_2})]) \vee (\forall \bar{z}_3.\, \varphi[(\top^{y_1}, \top^{y_2})]))\,,
\end{aligned}$$

[5] In fact, it is in L, as shown later in Theorem 19.

which is false if and only if there are assignments $\bar{a}^{\bar{z}_1}$, $\bar{b}^{\bar{z}_2}$, and $\bar{c}^{\bar{z}_2}$ such that

$$\forall \bar{z}_3.\, \varphi[(\bot^{y_1}, \bot^{y_2}, \bar{a}^{\bar{z}_1}, \bar{b}^{\bar{z}_2})] \vee \forall \bar{z}_3.\, \varphi[(\bot^{y_1}, \top^{y_2}, \bar{a}^{\bar{z}_1}, \bar{b}^{\bar{z}_2})] \vee \forall \bar{z}_3.\, \varphi[(\top^{y_1}, \bot^{y_2}, \bar{a}^{\bar{z}_1}, \bar{c}^{\bar{z}_2})]$$
$$\vee \forall \bar{z}_3.\, \varphi[(\top^{y_1}, \top^{y_2}, \bar{a}^{\bar{z}_1}, \bar{c}^{\bar{z}_2})]$$

is false. Since each of the four disjuncts is still in CNF, the formula is false if and only if each disjunct has a falsified clause. This is equivalent to finding four clauses $C_1, C_2, C_3, C_4 \in \varphi$ such that

- the clauses C_1, C_2, C_3, C_4 are consistent on the variables in $\bar{z}_1$,
- the clauses C_1, C_2 are consistent on the variables in $\bar{z}_2$,
- the clauses C_3, C_4 are consistent on the variables in $\bar{z}_2$, and
- $\neg C_1$, $\neg C_2$, $\neg C_3$, and $\neg C_4$ are consistent with $\neg y_1 \wedge \neg y_2$, $\neg y_1 \wedge y_2$, $y_1 \wedge \neg y_2$, and $y_1 \wedge y_2$, respectively.

To find such clauses, we can iterate through all 4-tuples of clause indices and check whether the properties hold. ◀

Next, we show that for every $k \geqslant 3$, $\mathsf{sat}(k\text{-DQBF}^{\mathsf{ds}}_{\mathsf{cnf}})$ is NP-complete, just like k-SAT.

▶ **Theorem 16.** *For every constant $k \geqslant 3$, $\mathsf{sat}(k\text{-DQBF}^{\mathsf{ds}}_{\mathsf{cnf}})$ is* NP*-complete.*

Before we present the proof of Theorem 16, we note that since $k\text{-DQBF}^{\mathsf{de}}_{\mathsf{cnf}} \subseteq k\text{-DQBF}^{\mathsf{dec}}_{\mathsf{cnf}} \subseteq k\text{-DQBF}^{\mathsf{ds}}_{\mathsf{cnf}}$, we obtain the following results as a corollary of Theorems 8 and 14–16.

▶ **Corollary 17.** $\mathsf{sat}(k\text{-DQBF}^{\mathsf{de}}_{\mathsf{cnf}})$ *and* $\mathsf{sat}(k\text{-DQBF}^{\mathsf{dec}}_{\mathsf{cnf}})$ *are* NL*-complete when $k = 2$ and* NP*-complete when $k \geqslant 3$.*

The rest of this section is devoted to the proof of Theorem 16.

Proof of Theorem 16. We will consider the membership proof. The hardness follows from Theorem 14. We fix a $k\text{-DQBF}^{\mathsf{ds}}_{\mathsf{cnf}}$ formula:

$$\Phi := \forall \bar{x}, \exists y_1(D_1), \ldots, \exists y_k(D_k).\ \bigwedge_{j \in [m]} C_j. \tag{3}$$

Without loss of generality, we may assume that no existential variable has an empty dependency set, since our NP algorithm can guess an assignment to such variables at the outset. By Lemma 2, we may also assume that every universal variable appears in some dependency set. We say that a dependency set D_i is maximal if there is no j where $D_i \subsetneq D_j$. An existential variable y_i is maximal if its dependency set is maximal.

To decide the satisfiability of Φ, our algorithm works by recursion on the number of existential variables. The base case is when there is only one existential variable. This case can be decided in polynomial time and, in fact, in deterministic logspace. See, e.g., Theorem 19.

For the induction step, we pick a maximal variable y_t. There are two cases.

Case 1: $D_t = \bar{x}$. We apply Lemma 3 and eliminate y_t, resulting in a formula with one less existential variable and $O(m^2)$ clauses. We then proceed recursively.

Case 2: $D_t \neq \bar{x}$. We deal with this case by generalising the technique in Lemma 13.

Let $\{i_1, \ldots, i_p\} = \{i \mid D_i \subseteq D_t\}$ and $\{i'_1, \ldots, i'_q\} = \{i' \mid D_{i'} \cap D_t = \emptyset\}$. For each $j \in [m]$, we partition C_j into two clauses:

$$C_j^{+t} := \{\ell \mid \mathrm{dep}(\ell) \subseteq D_t\}$$
$$C_j^{-t} := C_j \setminus C_j^{+t}$$

Intuitively, C_j^{+t} is the subclause of C_j that includes all the literals with dependency sets inside D_t. On the other hand, C_j^{-t} is the subclause that contains the rest of the literals. Due to the laminar structure of the dependency sets and that y_t is a maximal variable, $C_j^{-t} = \{\ell \mid \mathrm{dep}(\ell) \cap D_t = \emptyset\}$.

For a function $\xi : [m] \to \{+t, -t\}$, we define two formulas:

$$\Phi_{+t,\xi} := \forall \bar{x}, \exists y_{i_1}(D_{i_1}), \ldots, \exists y_{i_p}(D_{i_p}). \bigwedge_{j:\xi(j)=+t} C_j^{+t}$$

$$\Phi_{-t,\xi} := \forall \bar{x}, \exists y_{i'_1}(D_{i'_1}), \ldots, \exists y_{i'_q}(D_{i'_q}). \bigwedge_{j:\xi(j)=-t} C_j^{-t}$$

We have the following lemma.

▶ **Lemma 18.** *Φ is satisfiable if and only if there is a function $\xi : [m] \to \{+t, -t\}$ such that $\Phi_{+t,\xi}$ and $\Phi_{-t,\xi}$ are both satisfiable.*

Note that guessing ξ requires m bits. The algorithm guesses the function ξ and verifies recursively that both $\Phi_{+t,\xi}$ and $\Phi_{-t,\xi}$ are satisfiable. Since the algorithm terminates after k steps, and k is a constant, and the number of clauses constructed in each recursive step is at most quadratically many, each step can be done in polynomial time. ◀

The proof of Lemma 18 is a generalisation of Lemma 11. Let $\bar{Y}$ be the vector of variables in the expansion $\exp(\Phi)$. We can show that an assignment $\bar{a}$ on $\bar{Y}$ satisfies $\exp(\Phi)$ if and only if it satisfies $\exp(\Phi_{+t,\xi})$ and $\exp(\Phi_{-t,\xi})$ for some function ξ. A detailed proof can be found in the appendix.

5 Complexity of sat(k-DQBF$_{\mathsf{cnf}}$)

In this section, we remove the constraint on the dependency structure and consider k-DQBF$_{\mathsf{cnf}}$. The case $k = 1$ can be solved very efficiently.

▶ **Theorem 19.** sat(1-DQBF$_{\mathsf{cnf}}$) *is in* L.

Proof. Let $\Phi = \forall \bar{z}_1, \bar{z}_2, \exists y(\bar{z}_1). \bigwedge_{j\in[m]} C_j$. Similar to the proof of Theorem 15, to show unsatisfiability, it suffices to find $C_1, C_2 \in \varphi$ such that

- C_1, C_2 are consistent on the variables in $\bar{z}_1$, and
- $\neg C_1$ and $\neg C_2$ are consistent with $\neg y$ and y, respectively.

The correctness follows from the same reasoning. ◀

Next, we consider the case where $k = 2$.

▶ **Theorem 20.** sat(2-DQBF$_{\mathsf{cnf}}$) *is* coNP*-complete.*

Proof. For membership, we give an NP algorithm for checking unsatisfiability. Let $\Phi := \forall \bar{x}, \exists y_1(D_1), \exists y_2(D_2). \varphi$ be be a 2-DQBF$_{\mathsf{cnf}}$ formula. Let $\bar{z} = D_1 \cap D_2$. Note that for every assignment $\bar{a}$ on $\bar{z}$, the induced formula $\Phi[\bar{a}]$ is a 2-DQBF$^{\mathsf{d}}_{\mathsf{cnf}}$ formula, the satisfiability of which can be decided in polynomial time by Theorem 8. Therefore, to decide whether Φ is unsatisfiable, we can guess an assignment $\bar{a}$ on $\bar{z}$ and accept if and only if $\Phi[\bar{a}]$ is not satisfiable.

For hardness, we provide a reduction from the 3-DNF tautology problem to sat(2-DQBF$_{\mathsf{cnf}}$). Let $\varphi = \bigvee_{j\in[m]} Q_j$ be a 3-DNF formula over the variables $\bar{x} = (x_1, \ldots, x_n)$, where each $Q_j = (\ell_{j,1} \wedge \ell_{j,2} \wedge \ell_{j,3})$ is a 3-literal cube. We construct the following 2-DQBF$_{\mathsf{cnf}}$ formula:

$$\Psi = \forall \bar{x}, \forall \bar{u}_1, \forall \bar{u}_2, \exists y_1(\bar{x}, \bar{u}_1), \exists y_2(\bar{x}, \bar{u}_2). \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4\,,$$

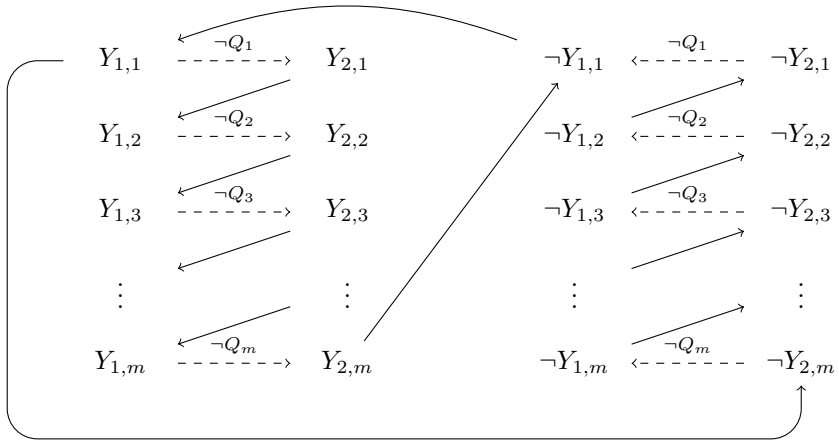


Figure 1 The implication graph $G_{\bar{a}}$. Each dashed edge $\overset{\neg Q_i}{\dashrightarrow}$ is present if and only if $\bar{a}^{\bar{x}}$ falsifies Q_i.

where $\bar{u}_1, \bar{u}_2$ have length $O(\log m)$ for representing the numbers in $[m]$ and $\psi_1, \ldots, \psi_4$ are as follows.

$$\psi_1 := (\bar{u}_1 = 1) \to y_1$$
$$\psi_2 := \bigwedge_{j \in [m-1]} \Big(((\bar{u}_1 = j+1) \wedge (\bar{u}_2 = j)) \to (y_2 \to y_1) \Big)$$
$$\psi_3 := \big((\bar{u}_1 = 1) \wedge (\bar{u}_2 = m)\big) \to (y_2 \to \neg y_1)$$
$$\psi_4 := \bigwedge_{j \in [m]} \bigwedge_{i \in [3]} \Big(((\bar{u}_1 = j) \wedge (\bar{u}_2 = j) \wedge \neg \ell_{j,i}) \to (y_1 \to y_2) \Big)$$

We claim that φ is a tautology if and only if Ψ is satisfiable. To see this, we fix an arbitrary assignment $\bar{a}$ on $\bar{x}$ and consider the induced formula $\Psi[\bar{a}]$. Note that $\Psi[\bar{a}]$ is a 2-DQBF$^{\mathsf{d}}_{\mathsf{cnf}}$ formula with universal variables $\bar{u}_1, \bar{u}_2$. Since $|\bar{u}_1| = |\bar{u}_2| = \log m$, the expansion $\exp(\Psi[\bar{a}])$ is a 2-CNF formula with $2m$ variables $Y_{1,1}, \ldots, Y_{1,m}, Y_{2,1}, \ldots, Y_{2,m}$. Here we abuse the notation and write $Y_{i,j}$ instead of $Y_{i,\bar{a}}$ where $\bar{a}$ is the binary representation of j.

It can be easily verified that the implication graph $G_{\bar{a}}$ of the expansion $\exp(\Psi[\bar{a}])$ is as shown in Figure 1, where a dashed edge $\overset{\neg Q_j}{\dashrightarrow}$ is present if and only if $\bar{a}$ falsifies the cube Q_j. Indeed, ψ_1 states that the edge $\neg Y_{1,1} \to Y_{1,1}$ is present. ψ_2 states that the edges $Y_{2,i} \to Y_{1,i+1}$ and $\neg Y_{1,i+1} \to \neg Y_{2,i}$ are present for every $i \in [m-1]$. ψ_3 states that the edges $Y_{2,m} \to \neg Y_{1,1}$ and $Y_{1,1} \to \neg Y_{2,m}$ are present. Finally, ψ_4 states that the dashed edges $Y_{1,j} \overset{\neg Q_j}{\dashrightarrow} Y_{2,j}$ and $Y_{1,j} \overset{\neg Q_j}{\dashrightarrow} Y_{2,j}$ are present if $\bar{a}^{\bar{x}}$ falsifies Q_j, for every $j \in [m]$. This implies that $\bar{a}^{\bar{x}}$ falsifies all cubes in φ if and only if there exists a cycle in $G_{\bar{a}}$. Since a cycle in $G_{\bar{a}}$ (if exists) contains contradicting literals, $\bar{a}^{\bar{x}}$ falsifies all cubes in φ if and only if $\Psi[\bar{a}]$ is not satisfiable. Since the assignment $\bar{a}$ is arbitrary, φ is a tautology if and only if Ψ is satisfiable. ◀

Next, we consider k-DQBF$_{\mathsf{cnf}}$. Note that 3-DQBF$_{\mathsf{cnf}}$ subsumes both 3-DQBF$^{\mathsf{d}}_{\mathsf{cnf}}$ and 2-DQBF$_{\mathsf{cnf}}$. Thus, sat(3-DQBF$_{\mathsf{cnf}}$) is is both NP-hard and coNP-hard. We improve these results by showing that it is Π^{P}_2-hard.

▶ **Theorem 21.** sat(3-DQBF$_{\mathsf{cnf}}$) *is* Π^{P}_2*-hard.*

Proof. We show this by reduction from Π_2-QBF in 3-CNF. The reduction is similar to the one in Theorem 14, except that the existential variables are allowed to depend on common variables, thereby enabling the encoding of a two-level QBF.

Let $\varphi = \forall \bar{z}, \exists \bar{x}.\ \bigwedge_{j\in[m]}(\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$ be a Π_2-QBF in 3-CNF. Let $\bar{x} = (x_1, \ldots, x_n)$ and for a literal ℓ, let $\mathrm{ind}(\ell)$ denote the index i where $x_i = \mathrm{var}(\ell)$.

We construct the following 3-DQBF$_{\mathsf{cnf}}$ formula:

$$\Psi \coloneqq \forall \bar{z}, \forall \bar{u}_1, \forall \bar{u}_2, \forall \bar{u}_3, \exists y_1(\bar{z}, \bar{u}_1), \exists y_2(\bar{z}, \bar{u}_2), \exists y_3(\bar{z}, \bar{u}_3).\ \psi_1 \wedge \psi_2 \wedge \psi_3\,,$$

where $\bar{u}_1, \bar{u}_2, \bar{u}_3$ have length $\log n$ for representing the variables in $\bar{x}$ and each ψ_1, ψ_2, ψ_3 is the following CNF formula:

$$\psi_1 \coloneqq \bigwedge_{i\in[n]} \Big(\big((\bar{u}_1 = i) \wedge (\bar{u}_2 = i)\big) \to (y_1 \leftrightarrow y_2) \Big)$$

$$\psi_2 \coloneqq \bigwedge_{i\in[n]} \Big(\big((\bar{u}_1 = i) \wedge (\bar{u}_3 = i)\big) \to (y_1 \leftrightarrow y_3) \Big)$$

$$\psi_3 \coloneqq \bigwedge_{j\in[m]} \Big((\sigma_{j,1} \wedge \sigma_{j,2} \wedge \sigma_{j,3}) \to (\tau_{j,1} \vee \tau_{j,2} \vee \tau_{j,3}) \Big)\,,$$

where for every $i \in [3]$, $\sigma_{j,i}$ and $\tau_{j,i}$ are defined as

$$(\sigma_{j,i}, \tau_{j,i}) = \begin{cases} (\bar{u}_i = \mathrm{ind}(\ell_{j,i}), y_i \leftrightarrow \mathrm{sgn}(\ell_{j,i})) & \text{if } \mathrm{var}(\ell_{j,i}) \text{ is existential in } \varphi \\ (\top, \ell_{j,i}) & \text{otherwise.} \end{cases}$$

The intuition is that a vector of Skolem functions $f_1, \ldots, f_n$ for $\bar{x}$ in φ can be encoded as a single function $g : \{\top, \bot\}^{|\bar{z}|} \times [n] \to \{\top, \bot\}$, which then corresponds to a Skolem function for each y_i in Ψ. The formulas ψ_1 and ψ_2 state that for every Skolem functions (g_1, g_2, g_3) for Ψ, all functions g_1, g_2, g_3 must be the same. Then, ψ_3 states that the Skolem functions $f_1, \ldots, f_n$ for φ (if exists) must satisfies all the clauses $C_1, \ldots, C_m$. Therefore, φ is a true QBF if and only if Ψ is satisfiable. ◀

We next show a Tseitin-like transformation that requires only three additional existential variables. This is in contrast with the standard Tseitin transformation that requires as many existential variables as the length of the input formula. Our new Tseitin-like transformation leads to new hardness results for $\mathsf{sat}(5\text{-DQBF}_{\mathsf{cnf}})$ and $\mathsf{sat}(6\text{-DQBF}_{\mathsf{cnf}})$.

▶ **Lemma 22** (Reformulated Tseitin Transformation)**.** *Given a k-DQBF formula, we can construct in polynomial time an equisatisfiable $(k+3)$-DQBF$_{\mathsf{cnf}}$ formula.*

Proof. Let $\Phi = \forall \bar{x}, \exists y_1(\bar{z}_1), \ldots, \exists y_k(\bar{z}_k).\ \varphi$ be a k-DQBF formula where φ is a circuit with gates $g_1, \ldots, g_m$. We assume, without loss of generality, that

$$g_i = \begin{cases} x_i & \text{for every } 1 \leqslant i \leqslant n \\ y_{i-n} & \text{for every } n+1 \leqslant i \leqslant n+k \\ f_i(g_{l_i}, g_{r_i}) & \text{for every } n+k+1 \leqslant i \leqslant m, \end{cases}$$

where $l_i, r_i \in [i-1]$ are the indices of the two fan-ins of the gate g_i implementing the Boolean function f_i, and that g_m corresponds to the primary output of the circuit.

We construct the $(k+3)$-DQBF$_{\mathsf{cnf}}$ formula

$$\Psi \coloneqq \forall \bar{x}, \forall \bar{u}_1, \forall \bar{u}_2, \forall \bar{u}_3, \exists y_1(\bar{z}_1), \ldots, \exists y_k(\bar{z}_k), \exists t_1(\bar{x}, \bar{u}_1), \exists t_2(\bar{x}, \bar{u}_2), \exists t_3(\bar{x}, \bar{u}_3).\ \psi\,,$$

where $\bar{u}_1, \bar{u}_2, \bar{u}_3$ have length $\log m$ for representing the numbers in $[m]$ and ψ is a CNF formula with clauses encoding

- $((\bar{u}_1 = i) \wedge (\bar{u}_2 = i)) \to (t_1 \leftrightarrow t_2)$ for every $i \in [m]$,
- $((\bar{u}_1 = i) \wedge (\bar{u}_3 = i)) \to (t_1 \leftrightarrow t_3)$ for every $i \in [m]$,
- $(\bar{u}_1 = i) \to (t_1 \leftrightarrow x_i)$ for every $i \in [n]$,
- $(\bar{u}_1 = n + i) \to (t_1 \leftrightarrow y_i)$ for every $i \in [k]$,
- $((\bar{u}_1 = i) \wedge (\bar{u}_2 = l_i) \wedge (\bar{u}_3 = r_i)) \to (t_1 \leftrightarrow f_i(t_2, t_3))$ for every $n + k + 1 \leqslant i \leqslant m$,
- $(\bar{u}_1 = m) \to t_1$.

The intuitive meaning is as follows. The first two items state that the Skolem functions for t_1, t_2, t_3 must be the same. The next three state that the Skolem function g for t_1 encodes the values of the gates, where $g(\bar{x}, \bar{u}_1)$ is the value of gate j when $\bar{u}_1$ is the binary representation of j. The last item encodes that the output of the circuit must be true. The correctness proof is routine and hence omitted. ◀

In particular, by combining Lemma 22 with the fact that sat(2-DQBF) and sat(3-DQBF) are PSPACE-complete and NEXP-complete [14], we establish the hardness of $\mathsf{sat}(5\text{-DQBF}_{\mathsf{cnf}})$ and $\mathsf{sat}(6\text{-DQBF}_{\mathsf{cnf}})$.

▶ **Theorem 23.** $\mathsf{sat}(5\text{-DQBF}_{\mathsf{cnf}})$ *is* PSPACE*-hard and* $\mathsf{sat}(6\text{-DQBF}_{\mathsf{cnf}})$ *is* NEXP*-complete.*

6 Conclusions and Future Work

While $\mathsf{sat}(k\text{-DQBF}^{\mathsf{d}}_{\mathsf{dnf}})$ is as hard as $\mathsf{sat}(k\text{-DQBF})$, we observe a range of differing complexity results in the CNF case. For the case of $\mathsf{sat}(k\text{-DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$, we show that it is in fact as easy as k-SAT – exponentially easier than $\mathsf{sat}(k\text{-DQBF})$. Generalising the results in [18], we also show that $\mathsf{sat}(\text{DQBF}^{\mathsf{d}}_{\mathsf{cnf}})$ is NP-complete and that $\mathsf{sat}(k\text{-DQBF}^{\alpha}_{\mathsf{cnf}})$ has the same complexity as k-SAT for $\alpha \in \{\mathsf{d}, \mathsf{de}, \mathsf{dec}, \mathsf{ds}\}$. For the case of $k\text{-DQBF}_{\mathsf{cnf}}$, we show that it is only coNP-complete when $k = 2$ (whereas sat(2-DQBF) is PSPACE-complete) and of the same NEXP-complete complexity as sat(DQBF) when $k \geqslant 6$. These results show that, when parametrising DQBF with the number of existential variables, it is more natural to consider DNF as the normal form for the matrix, analogous to how CNF is considered the standard form for SAT.

The exact complexity of $\mathsf{sat}(k\text{-DQBF}_{\mathsf{cnf}})$ is yet to be discovered for $k = 3$, 4, and 5. In particular, the best-known membership result is still that they are in NEXP. We leave this for future work.

References

1 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.

2 V. Balabanov and J.-H. R. Jiang. Reducing satisfiability and reachability to DQBF. In *QBF Workshop*, 2015.

3 Valeriy Balabanov, Hui-Ju Katherine Chiang, and Jie-Hong R. Jiang. Henkin quantifiers and Boolean formulae: A certification perspective of DQBF. *Theoretical Computer Science*, 523:86–100, 2014. doi:10.1016/j.tcs.2013.12.020.

4 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*, volume 3545, pages 72–87. CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3545/paper6.pdf.

5 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185. IOS Press, 2009.

6 Uwe Bubeck. *Model-Based Transformations for Quantified Boolean Formulas*. PhD thesis, University of Paderborn, 2010.

7 Uwe Bubeck and Hans Kleine Büning. Dependency quantified Horn formulas: Models and complexity. In Armin Biere and Carla P. Gomes, editors, *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121, pages 198–211. Springer, 2006. doi:10.1007/11814948_21.

8 Uwe Bubeck and Hans Kleine Büning. Rewriting (dependency-)quantified 2-CNF with arbitrary free literals into existential 2-HORN. In Ofer Strichman and Stefan Szeider, editors, *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference*, volume 6175, pages 58–70. Springer, 2010. doi:10.1007/978-3-642-14186-7_7.

9 Fa-Hsun Chen, Shen-Chang Huang, Yu-Cheng Lu, and Tony Tan. Reducing NEXP-complete problems to DQBF. In Alberto Griggio and Neha Rungta, editors, *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design*, pages 199–204, 2022. doi:10.34727/2022/isbn.978-3-85448-053-2_26.

10 Johannes Klaus Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. The silent (r)evolution of SAT. *Commun. ACM*, 66(6):64–72, 2023. doi:10.1145/3560469.

11 Ana Paulina Figueroa and Bernardo Llano. An infinite family of self-diclique digraphs. *Appl. Math. Lett.*, 23(5):630–632, 2010. doi:10.1016/j.aml.2010.01.026.

12 Andreas Fröhlich, Gergely Kovásznai, Armin Biere, and Helmut Veith. iDQ: Instantiation-based DQBF solving. In *Proceedings of the Fifth Pragmatics of SAT workshop*, volume 27, pages 103–116, 2014. doi:10.29007/1s5k.

13 Long-Hin Fung, Che Cheng, Yu-Wei Fan, Tony Tan, and Jie-Hong Roland Jiang. 2-DQBF solving and certification via property-directed reachability analysis. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*, pages 197–207. IEEE, 2024. doi:10.34727/2024/isbn.978-3-85448-065-5_25.

14 Long-Hin Fung and Tony Tan. On the complexity of k-DQBF. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, volume 271, pages 10:1–10:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SAT.2023.10.

15 Robert Ganian, Tomás Peitl, Friedrich Slivovsky, and Stefan Szeider. Fixed-parameter tractability of dependency QBF with structural parameters. In Diego Calvanese, Esra Erdem, and Michael Thielscher, editors, *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020*, pages 392–402, 2020. doi:10.24963/kr.2020/40.

16 Mikolás Janota and João Marques-Silva. Expansion-based QBF solving versus Q-resolution. *Theor. Comput. Sci.*, 577(C):25–42, 2015. doi:10.1016/j.tcs.2015.01.048.

17 G. Peterson, J. Reif, and S. Azhar. Lower bounds for multiplayer noncooperative games of incomplete information. *Computers & Mathematics with Applications*, 41(7):957–992, 2001. doi:10.1016/S0898-1221(00)00333-3.

18 Christoph Scholl, Jie-Hong Roland Jiang, Ralf Wimmer, and Aile Ge-Ernst. A PSPACE subclass of dependency quantifed Boolean formulas and its effective solving. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1584–1591. AAAI Press, 2019. doi:10.1609/aaai.v33i01.33011584.

19 G. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part II*, 1968.

20 Ralf Wimmer, Karina Gitina, Jennifer Nist, Christoph Scholl, and Bernd Becker. Preprocessing for DQBF. In Marijn Heule and Sean A. Weaver, editors, *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, volume 9340, pages 173–190, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-24318-4_13.

21 Lintao Zhang. Solving QBF with combined conjunctive and disjunctive normal form. In *Proceedings of the 21st National Conference on Artificial Intelligence*, pages 143–150. AAAI Press, 2006. URL: http://www.aaai.org/Library/AAAI/2006/aaai06-023.php.

A Additional Background

We provide additional background knowledge in this section.

A.1 Tseitin Transformation

Tseitin transformation is a standard technique to turn an arbitrary Boolean satisfiability problem into an equisatisfiable one in 3-CNF form [19]. It can be directly lifted to QBF and DQBF by allowing the Tseitin variables to depend on every universal variable.

Given a DQBF

$$\Phi = \forall \bar{x}, \exists y_1(\bar{z}_1), \ldots, \exists y_k(\bar{z}_k).\, \varphi\,,$$

where φ is a circuit with gates $g_1, \ldots, g_m$, we assume, without loss of generality, that

$$g_i = \begin{cases} x_i & \text{for every } 1 \leqslant i \leqslant n \\ y_{i-n} & \text{for every } n+1 \leqslant i \leqslant n+k \\ f_i(g_{l_i}, g_{r_i}) & \text{for every } n+k+1 \leqslant i \leqslant m, \end{cases}$$

where $l_i, r_i \in [i-1]$ are the indices of the two fanins of the gate g_i implementing the Boolean function f_i.

The core idea of Tseitin transformation is that we can introduce a fresh variable t_i for every gate g_i and encode locally the relation between the inputs and the output of the gate. The formula ψ_G encoding these constraints is a CNF formula encoding

- $t_i \leftrightarrow x_i$ for every $1 \leqslant i \leqslant n$,
- $t_i \leftrightarrow y_{i-n}$ for every $n+1 \leqslant i \leqslant n+k$, and
- $t_i \leftrightarrow f_i(t_{l_i}, t_{r_i})$ for every $n+k+1 \leqslant i \leqslant m$.

We then have Φ is equisatisfiable to

$$\Psi_1 \coloneqq \forall \bar{x}, \exists y_1(\bar{z}_1), \ldots, \exists y_k(\bar{z}_k), \exists \bar{t}(\bar{x}).\, \psi_G \wedge t_m\,.$$

To transform it to DNF form, as noted in [9], Φ is equisatisfiable to

$$\Psi_2 \coloneqq \forall \bar{x}, \forall \bar{t}, \exists y_1(\bar{z}_1), \ldots, \exists y_k(\bar{z}_k).\, \psi_G \to t_m\,.$$

Note that the matrix of the formula is in DNF form. In the context of QBF, it can be thought of as applying the Tseitin transformation on $\neg\varphi$ and then negating the resulting existential formula [21]. We refer to this as the *DNF version* of Tseitin transformation.

B Omitted Proofs

We fill in the omitted proofs in the main text in this section.

B.1 Complexity of sat(k-DQBF$^{\mathsf{d}}_{\mathsf{dnf}}$)

Proof of Theorem 4. Since we are considering subclasses of k-DQBF, it suffices to show the hardness part.

First, observe that the DNF version of the Tseitin transformation (see Section A.1) preserves both the number of existential variables and the dependency structure. Therefore, we have that sat(k-DQBF$^{\alpha}_{\mathsf{dnf}}$) is as hard as sat(k-DQBF$^{\alpha}$) for every combination of α and k. In addition, observe that the formula constructed to show the PSPACE- and NEXP-hardness of sat(2-DQBF) and sat(3-DQBF) in [14, Theorems 4 and 5] are in fact 2-DQBF$^{\mathsf{d}}$ and 3-DQBF$^{\mathsf{d}}$, respectively. Thus, we have sat(k-DQBF$^{\mathsf{d}}_{\mathsf{dnf}}$) is coNP-, PSPACE-, and NEXP-complete for $k = 1$, $k = 2$, and $k \geqslant 3$, respectively. ◀

B.2 Hardness of sat(3-DQBF$^{\mathsf{d}}_{\mathsf{cnf}}$)

Hardness Proof of Theorem 14. Let $\varphi = \bigwedge_{j\in[m]}(\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$ be a 3-CNF formula over the variables $\bar{x} = (x_i)_{i\in[n]}$. We again write $\mathrm{ind}(\ell) \coloneqq i$ if $\mathrm{var}(\ell) = x_i$.

We construct the following 3-DQBF$^{\mathsf{d}}_{\mathsf{cnf}}$ formula

$$\Psi \coloneqq \forall \bar{u}_1, \forall \bar{u}_2, \forall \bar{u}_3, \exists y_1(\bar{u}_1), \exists y_2(\bar{u}_2), \exists y_3(\bar{u}_3).\, \psi\,,$$

where $\bar{u}_1$, $\bar{u}_2$, and $\bar{u}_3$ are of length $O(\log n)$ for representing the variables in $\bar{x}$ and ψ is a CNF formula that states the following.

- $(\bar{u}_1 = i) \wedge (\bar{u}_2 = i) \to (y_1 \leftrightarrow y_2)$ for every $i \in [n]$,
- $(\bar{u}_1 = i) \wedge (\bar{u}_3 = i) \to (y_1 \leftrightarrow y_3)$ for every $i \in [n]$, and
- $(\bar{u}_1 = \mathrm{ind}(\ell_{j,1})) \wedge (\bar{u}_2 = \mathrm{ind}(\ell_{j,2})) \wedge (\bar{u}_3 = \mathrm{ind}(\ell_{j,3})) \to (y_1 \leftrightarrow \mathrm{sgn}(\ell_{j,1})) \vee (y_2 \leftrightarrow \mathrm{sgn}(\ell_{j,2})) \vee (y_3 \leftrightarrow \mathrm{sgn}(\ell_{j,3}))$ for every $j \in [m]$.

The first two items state that the Skolem functions for y_1, y_2, and y_3 must be the same. The third item implies that $\bar{a}^{\bar{x}}$ is a satisfying assignment of φ if and only if the function $f_{\bar{a}}$ is a Skolem function for Ψ by encoding $\bar{a}^{\bar{x}}$ as a function $f_{\bar{a}} \colon [n] \to \{\top, \bot\}$, where $f_{\bar{a}}(i) = \bar{a}^{\bar{x}}(x_i)$. ◀

B.3 Proof of Lemma 18

Recall that we fix a k-DQBF$^{\mathsf{ds}}_{\mathsf{cnf}}$ formula:

$$\Phi \coloneqq \forall \bar{x}, \exists y_1(D_1), \ldots, \exists y_k(D_k).\ \bigwedge_{j\in[m]} C_j. \tag{4}$$

Let $\bar{y} = (y_1, \ldots, y_k)$. We recall some of the notation used in the main text.

- y_t is a maximal variable where $D_t \neq \bar{x}$.
- $\{i_1, \ldots, i_p\} = \{i \mid D_i \subseteq D_t\}$ and $\{i'_1, \ldots, i'_q\} = \{i' \mid D_{i'} \cap D_t = \emptyset\}$.
- For each $j \in [m]$:

$$C_j^{+t} \coloneqq \{\ell \mid \mathrm{dep}(\ell) \subseteq D_t\}$$
$$C_j^{-t} \coloneqq C_j \setminus C_j^{+t}$$

- For a function $\xi : [m] \to \{+t, -t\}$, we define two formulas:

$$\Phi_{+t,\xi} \coloneqq \forall \bar{x}, \exists y_{i_1}(D_{i_1}), \ldots, \exists y_{i_p}(D_{i_p}).\ \bigwedge_{j \text{ s.t. } \xi(j)=+t} C_j^{+t}$$
$$\Phi_{-t,\xi} \coloneqq \forall \bar{x}, \exists y_{i'_1}(D_{i'_1}), \ldots, \exists y_{i'_q}(D_{i'_q}).\ \bigwedge_{j \text{ s.t. } \xi(j)=-t} C_j^{-t}$$

Finally, we recall Lemma 18.

▶ **Lemma 18.** *Φ is satisfiable if and only if there is a function $\xi : [m] \to \{+t, -t\}$ such that $\Phi_{+t,\xi}$ and $\Phi_{-t,\xi}$ are both satisfiable.*

To prove Lemma 18, we will need additional notation and terminology. Let $S_t \coloneqq \{y_i \mid D_i \subseteq D_t\}$. Note that $y_t \in S_t$. To ease notation, we write $D_t^c \coloneqq \bar{x} \setminus D_t$ and $S_t^c \coloneqq \bar{y} \setminus S_t$. That is, D_t^c is the complement of D_t w.r.t. $\bar{x}$ and S_t^c is the complement of S_t w.r.t. $\bar{y}$. In the following, we will drop the subscript t in D_t, S_t, D_t^c, S_t^c and simply write D, S, D^c, S^c.

For an assignment $(\bar{a}^D, \bar{b}^S)$, we define the clause $\mathrm{cl}(\bar{a}^D, \bar{b}^S)$:

$$\mathrm{cl}(\bar{a}^D, \bar{b}^S) \coloneqq \bigvee_{i \in S} Y_{i,\bar{a}_i} \oplus b_i, \qquad \text{where each } \bar{a}_i = \bar{a}^D(D_i) \text{ and } b_i = \bar{b}^S(y_i).$$

Similarly, for an assignment $(\bar{a}^{D^c}, \bar{b}^{S^c})$, we define the clause $\mathrm{cl}(\bar{a}^{D^c}, \bar{b}^{S(y)^c})$:

$$\mathrm{cl}(\bar{a}^{D^c}, \bar{b}^{S^c}) \coloneqq \bigvee_{y_i \in S^c} Y_{i,\bar{a}_i} \oplus b_i, \qquad \text{where each } \bar{a}_i = \bar{a}^{D^c}(D_i) \text{ and } b_i = \bar{b}^{S^c}(y_i).$$

We now generalise Definition 6 to the laminar case.

▶ **Definition 24.** *Let Φ be as in Equation (4). For every $j \in [m]$, we define the sets:*

$$\mathcal{L}^*_{+t,j}(\Phi) \coloneqq \{\mathrm{cl}(\bar{a}^D, \bar{b}^S) \mid (\bar{a}^D, \bar{b}^S) \simeq \neg C_j^{+t}\}$$
$$\mathcal{L}^*_{-t,j}(\Phi) \coloneqq \{\mathrm{cl}(\bar{a}^{D^c}, \bar{b}^{S^c}) \mid (\bar{a}^{D^c}, \bar{b}^{S^c}) \simeq \neg C_j^{-t}\}$$
$$\mathfrak{C}_j(\Phi) \coloneqq \{\mathcal{C}_{\bar{a},\bar{b}} \mid (\bar{a}^{\bar{x}}, \bar{b}^{\bar{y}}) \simeq \neg C_j\}$$

The following lemma is a generalisation of Lemma 7 to the laminar case.

▶ **Lemma 25.** *Let Φ be as in Equation (4). Then, for every $j \in [m]$, $\mathfrak{C}_j = \mathcal{L}^*_{+t,j}(\Phi) \times \mathcal{L}^*_{-t,j}(\Phi)$.*

Proof. The proof is a straightforward generalisation of Lemma 7. For the sake of completeness, we present it here.

We fix an arbitrary $j \in [m]$. We first prove the "$\subseteq$" direction. Let $\mathcal{C}_{\bar{a},\bar{b}}$ be a clause in $\mathfrak{C}_j$. That is, $(\bar{a}^{\bar{x}}, \bar{b}^{\bar{y}})$ is an assignment that falsifies C_j. By definition, $\mathcal{C}_{\bar{a},\bar{b}} = \bigvee_{i \in [k]} Y_{i,\bar{a}_i} \oplus b_i$. Since $(\bar{a}, \bar{b})$ falsifies C_j, it is consistent with the cube $\neg C_j$.

Let $\bar{a}_t = \bar{a}^{\bar{x}}(D)$ and $\bar{b}_t = \bar{b}^{\bar{y}}(S)$. Let $\bar{a}_0 = \bar{a}^{\bar{x}}(D^c)$ and $\bar{b}_0 = \bar{b}^{\bar{y}}(S^c)$. Both are consistent with the cubes $\neg C_j^{+t}$ and $\neg C_j^{-t}$, respectively. By definition, the clause $\mathrm{cl}(\bar{a}_t^D, \bar{b}_t^S)$ is in $\mathcal{L}^*_{+t,j}(\Phi)$ and the clause $\mathrm{cl}(\bar{a}_t^{D^c}, \bar{b}_t^{S^c})$ is in $\mathcal{L}^*_{-t,j}(\Phi)$. The inclusion follows since

$$\mathcal{C}_{\bar{a},\bar{b}} = \mathrm{cl}(\bar{a}_t^D, \bar{b}_t^S) \vee \mathrm{cl}(\bar{a}_t^{D^c}, \bar{b}_t^{S^c})$$

Next, we prove the "$\supseteq$" direction. Let $C \in \mathcal{L}^*_{+t,j}(\Phi) \times \mathcal{L}^*_{-t,j}(\Phi)$. Let $C \coloneqq B_1 \vee B_2$, where $B_1 \in \mathcal{L}^*_{+t,j}(\Phi)$ and $B_2 \in \mathcal{L}^*_{-t,j}(\Phi)$. By definition,

- there is assignment $(\bar{a}_1^D, \bar{b}_1^S)$ such that B_1 is the clause $\mathrm{cl}(\bar{a}_1^D, \bar{b}_1^S)$,
- there is assignment $(\bar{a}_2^{D^c}, \bar{b}_2^{S^c})$ such that B_2 is the clause $\mathrm{cl}(\bar{a}_2^{D^c}, \bar{b}_2^{S^c})$.

Since the dependency sets of the variables in S are disjoint with the dependency sets of the variables in S^c , the assignments $(\bar{a}_1^D, \bar{b}_1^S)$ and $(\bar{a}_2^{D^c}, \bar{b}_2^{S^c})$ are consistent. Let $(\bar{a}^{\bar{x}}, \bar{b}^{\bar{y}})$ be their union, which is consistent with $\neg C_j^{+t} \wedge \neg C_j^{-t}$. Therefore, $(\bar{a}^{\bar{x}}, \bar{b}^{\bar{y}})$ is a falsifying assignment of C_j. By definition, the clause $\mathcal{C}_{\bar{a},\bar{b}} = B_1 \vee B_2$ and it is in $\mathfrak{C}_j$. ◀

Now, Lemma 18 follows from the following lemma, which is the generalisation of Lemma 11.

▶ **Lemma 26.** *Let Φ be as in Equation (4) and let $\bar{Y}$ be the vector of variables in* $\exp(\Phi)$*. For every assignment $\bar{a}^{\bar{Y}}$, $\bar{a}^{\bar{Y}}$ satisfies* $\exp(\Phi)$ *if and only if it satisfies* $\exp(\Phi_{+t,\xi})$ *and* $\exp(\Phi_{-t,\xi})$ *for some function $\xi : [m] \to \{+t, -t\}$.*

Proof. The proof is similar to Lemma 26. For completeness, we reprove it here. We observe that:

$$\exp(\Phi) = \bigwedge_{j \in [m]} \bigwedge_{C \in \mathfrak{C}_j} C$$
$$= \bigwedge_{j \in [m]} \bigwedge_{(C_1, C_2) \in \mathcal{L}^*_{+t,j}(\Phi) \times \mathcal{L}^*_{-t,j}(\Phi)} C_1 \vee C_2$$

The second equality comes from Lemma 25. Thus, $\exp(\Phi)$ is satisfiable iff there is a function $\xi : [m] \to \{+t, -t\}$ such that

$$\left(\bigwedge_{j:\xi(j)=+t} \bigwedge_{C_1 \in \mathcal{L}^*_{+t,j}(\Phi)} C_1 \right) \wedge \left(\bigwedge_{j:\xi(j)=-t} \bigwedge_{C_2 \in \mathcal{L}^*_{-t,j}(\Phi)} C_2 \right)$$

is satisfiable. The first part of the conjunction is precisely $\exp(\Phi_{+t,\xi})$ and the second part is precisely $\exp(\Phi_{-t,\xi})$. ◀

Better Extension Variables in DQBF via Independence

Leroy Chew
TU Wien, Austria

Tomáš Peitl
TU Wien, Austria

Abstract

We show that extension variables in (D)QBF can be generalised by conditioning on universal assignments. The benefit of this is that the dependency sets of such conditioned extension variables can be made smaller to allow easier refutations. This simple modification instantly solves many challenges in p-simulating the QBF expansion rule, which cannot be p-simulated in proof systems that have strategy extraction [13]. Simulating expansion is even more crucial in DQBF, where other methods are incomplete. In this paper we provide an overview of the strength of this new independent extension rule. We find that a new version of Extended Frege called IndExtFrege + ∀red can p-simulate a multitude of difficult QBF and DQBF techniques, even techniques that are difficult to approach with eFrege + ∀red. We show five p-simulations, that IndExtFrege + ∀red p-simulates QRAT, DQBF-IR-calc, IR($\mathcal{D}^{\text{rrs}}$)-calc, Fork-Resolution and DQRAT which together underpin most DQBF solving and preprocessing techniques. The p-simulations work despite these systems using complicated rules and our new extension rule being relatively simple. Moreover, unlike recent p-simulations by eFrege + ∀red we can simulate the proof rules line by line, which allows us to mix QBF rules more easily with other inference steps.

2012 ACM Subject Classification Theory of computation → Proof complexity

Keywords and phrases DQBF, QBF, Proof Systems, Dependency Schemes, RAT, Extended Frege, Skolem functions

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.11

Related Version *Full Version*: https://arxiv.org/abs/2505.20069 [15]

Funding *Leroy Chew*: Funding by FWF ESPRIT grant number ESP-197.

Acknowledgements Thanks to Marijn Heule and Mikoláš Janota for their discussions on this topic. Thanks to the anonymous reviewers for the valuable feedback.

1 Introduction

Proof systems that allow extension variables are very powerful [17]. We know that in propositional logic, extended resolution can p-simulate a multitude of disparate techniques, and lower bounds to extended resolution remain an open problem. This is remarkable because resolution itself is a weak proof system with various lower bounds, but simply adding the extension rule upgrades it to be amongst the most powerful propositional proof systems and is equivalent to the checking format DRAT [31]. An extension rule for variable v takes the form:

$v \leftrightarrow b(X)$: b is a Boolean function, X is a set of existing variables

Quantified Boolean Formulas (QBF) also use extension variables [19] and these too can be very powerful. QBFs list all variables in a quantifier prefix. This defines whether a variable is existential (∃) or universal (∀). The order defines dependencies so variables only depend on other variables to their left. When used in refutations, extension variables must


© Leroy Chew and Tomáš Peitl;
licensed under Creative Commons License CC-BY 4.0



be existential, otherwise it is too easy to violate extension clauses. The dependency set of the new variables (which is the same as the quantification order in QBF) must be careful not to introduce falsity into the formula. In fact one of the earliest approaches was to say that every new variable is quantified rightmost to conservatively give it the entire dependency set. The drawback with this approach is that it limits the power of inferences from using extension variables [8]. In the current alternative, sometimes known as "strong extension", we place $\exists v$ immediately after all the variables X used in b. eFrege + $\forall$red uses this strong notion, and it is so powerful no unconditional lower bound can be found, unless either a long standing proof complexity or circuit complexity open problem is solved [7]. Some problems do emerge when looking at conditional lower bounds, though. The Select family of formulas can be shown by the QBF proof system QRAT to be equivalent to the law of non-contradiction on QBFs, but do not have short proofs in QBF Extended Frege unless PSPACE $\subseteq$ P/poly [13]. QRAT manages this through a combination of extension variables and an explicit rule that calculates so-called spurious dependencies in quantified variables, when nominal dependencies can be ignored when making inferences. Note how different this is to propositional logic, in most cases where we would take a basic proof system and add even a complicated rule we would still be simulated by extended resolution. Based on our observations in this paper we find that we can do better with extension variables. Instead of extensions being pure definitions, they are now under conditions.

$$\alpha \to (v \leftrightarrow b(X)) : \alpha \text{ is a partial assignment of the universal variables}$$

The utility of this is not immediately obvious because v is weaker than it could be, however precisely because this is weaker now the dependency of v on the variables of α is no longer necessary for soundness. Because we can remove arbitrary dependencies, the natural class for this proof system is for Dependency QBF (DQBF). But we can still use this rule for QBF, in fact it adds substantial clarity to existing QBF proof systems.

Our main contribution is that we propose a line based proof system, that each new line addition preserves satisfiability, so when arriving at the falsum symbol 0 we know we started with an unsatisfiable DQBF. Unlike many QBF systems we are not able to construct countermodels from following the proof steps (unless P = PSPACE). We then show how it can p-simulate existing QBF and DQBF proof system rules. Figures 1 and 2 shows the known p-simulations in DQBF and QBF proof systems *after* considering the work in this paper[1]. New results are presented in **bold**. The existing results are from papers that span over decades worth of work, and transitively show our new proof system p-simulates all systems in both figures [1, 2, 3, 4, 6, 7, 9, 10, 12, 14, 16, 15, 18, 22, 23, 24, 25, 26, 28, 30].

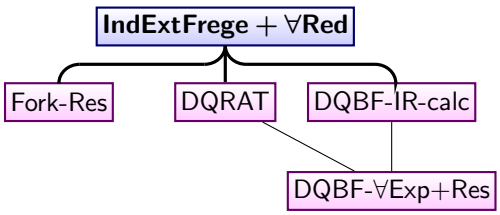


Figure 1 The p-simulation structure of refutational S-form DQBF proof systems.

[1] The p-simulation of G was discovered afterwards and is proven in the full version of this paper [15]

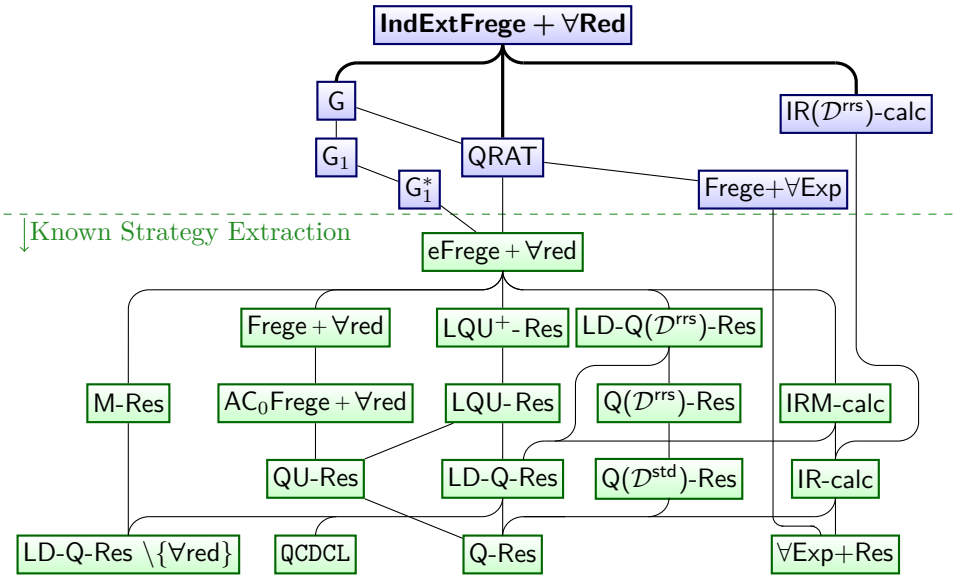


Figure 2 The p-simulation structure of refutational QBF proof systems.

1.1 Related Work

The original Extended Q-Resolution was defined by Jussila et al. [19] and discussed the weak and strong extension variables. QRAT by Biere, Heule and Seidl generalises extension variables in its clause additions. In the original paper [18] and later in Kiesl and Seidl's paper [22], they use similar clause additions to our independent extension definitions. Unlike our definitions, the new variables have a larger dependency set, but get round this by detecting spurious dependencies with an Extended Universal Reduction rule (EUR). These QRAT extension variables can capture the annotated variables used in the QBF proof system ∀Exp+Res, but there has not been success making the same method work for the more general QBF proof system IR-calc [11]. Our approach works backwards, instead of using the presence of resolution path schemes to derive expansions, we use expansions to show the validity of resolution path schemes.

Blinkhorn proposed a DQBF proof system based on generalising the QBF proof system QRAT to DQBF [10]. This system too allows the addition of new variables v, but when using the RAT addition rules to make definition clauses the RAT addition rule only allows the variables in the dependency set of v to be used unlike in our case. Therefore we suspect their proof system is weaker. Rabe proposed a DQBF proof system called Fork-Resolution that has extension variables but only in the case of clause-splitting [26].

Chew and Clymo showed that QRAT's strongest rule violated the property of strategy extraction unless P = PSPACE [13]. Later, Chew and Heule showed that the QBF sequent calculus G p-simulates QRAT [14]. The sequent calculus G [24] creates quantified variables of QBF witnesses so it has a witnessing scheme that works to simulate EUR. We use the same witnessing scheme in our work to simulate EUR with our DQBF system, hence showing strategy extraction, even for the QBF fragment, is impossible unless P = PSPACE.

The discussion around new variables comes after Reichl and Slivovsky's successful DQBF solver `Pedant` [27] introduces the similar *arbiter variables*, with a few more limitations.

1.2 Organisation

We define the necessary preliminaries of QBF and DQBF in Section 2. As an example we include the DQBF complete refutation system DQBF-IR-calc. In Section 3 we define our new extension rule and show its soundness, firstly we integrate it into a full proof Frege system that we call IndExtFrege + ∀red in Section 3.2, and later an equivalent resolution system in Section 3.3. In order to show completeness we p-simulate the complete DQBF-IR-calc proof system. Section 4 demonstrates the advantages of using IndExtFrege + ∀red on (D)QBFs, with more p-simulations. In Section 4.1 we give a definition of the clausal proof system Fork-Resolution for DQBF and show how to p-simulate that in IndExtFrege + ∀red. In Section 4.2 we give a definition of the QBF interference based proof system QRAT and show how IndExtFrege + ∀red can p-simulate it. In Section 4.3 we do the same for the QBF proof system IR($\mathcal{D}^{\mathsf{rrs}}$)-calc which uses the reflexive resolution path dependency scheme.

2 Preliminaries

We assume a countably infinite set $\mathcal{V}$ of propositional *variables* is given. A *literal* is either a variable $x \in \mathcal{V}$ or its negation ($\neg x$), also written as $\bar{x}$, whereby $\bar{\bar{x}} = x$. A *(propositional) formula* is defined recursively: (1) literals are formulas; and (2) if ϕ and ψ are formulas, then $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \to \psi$, $\phi \leftrightarrow \psi$ and $\bar{\phi}$ are also formulas. A *circuit* is like a formula, but its recursive structure is a DAG (directed acyclic graph) instead of a tree: subformulas can be reused. Every formula is a circuit, but transforming a circuit into a formula involves potentially exponential duplication of nodes. The set of variables of a circuit or formula is defined recursively as $\mathrm{var}(\phi \circ \psi) = \mathrm{var}(\phi) \cup \mathrm{var}(\psi)$ for any operator $\circ$, $\mathrm{var}(\bar{\phi}) = \mathrm{var}(\phi)$, and $\mathrm{var}(x) = x$ if x is a variable. We use $\sqcup$ to denote the disjoint union, i.e. the union of two sets that are known to be disjoint. A *clause* is a finite set of literals, semantically interpreted as their disjunction (equivalently, a formula consisting only of literals and "or" connectives). A clause C is *tautological*, if $\{x, \bar{x}\} \in C$ for some variable x. A clause is *unit* if only contains a single literal, a clause is the *empty clause* if it contains no literals. The negation $\bar{C}$ of a clause C can be viewed as a conjunction of unit clauses $\bigwedge_{l \in C} \{\bar{l}\}$.

A *(partial) assignment* to a circuit ϕ is a mapping $\alpha : V \subseteq \mathrm{var}(\phi) \to \{0, 1\}$. A partial assignment α is *complete* if it assigns every variable, i.e. $V = \mathrm{var}(\phi)$. We can write an assignment α as a function, a conjunction of literals or sequence of literals i.e. $x\bar{y}\bar{z}$. As a function an assignment is extended to circuits by $\alpha(\phi \wedge \psi) = \alpha(\phi) \cdot \alpha(\psi)$, $\alpha(\phi \vee \psi) = \max(\alpha(\phi), \alpha(\psi))$ and $\alpha(\bar{\phi}) = 1 - \alpha(\phi)$. α *satisfies* (is a *model* of) a circuit ϕ if $\alpha(\phi) = 1$. A circuit is *satisfiable* if it has a model, and *unsatisfiable* otherwise. Circuit ϕ is a *tautology* if $\bar{\phi}$ is unsatisfiable. Circuit ϕ *entails* a circuit ψ, written $\phi \models \psi$, if every complete assignment $\alpha : \mathrm{var}(\phi) \to \{0, 1\}$ that satisfies ϕ also satisfies ψ. Two circuits ϕ and ψ are *(logically) equivalent*, written $\phi \equiv \psi$, if $\mathrm{var}(\phi) = \mathrm{var}(\psi)$, and $\phi \models \psi$ and $\psi \models \phi$. Two circuits ϕ and ψ are *equisatisfiable*, written $\phi \cong \psi$, if they are both satisfiable or both unsatisfiable.

A formula is in *conjunctive normal form (CNF)* if it is a (finite) conjunction of clauses. Any circuit ϕ can be transformed into a logically equivalent CNF using distributivity and De Morgan's rules, but the resulting size may be exponential. Allowing *extension variables*, i.e. variables that do not occur in ϕ, it is possible to transform any circuit ϕ into $\mathrm{CNF}(\phi)$, an equisatisfiable CNF in linear time [29]. This translation has the additional property that if α is a model of $\mathrm{CNF}(\phi)$, then $\alpha|_{\mathrm{var}(\phi)}$ is a model of ϕ and vice versa, if α is a model of ϕ, then there exists a model β of $\mathrm{CNF}(\phi)$ such that $\beta(x) = \alpha(x)$ for all $x \in \mathrm{var}(\phi)$.

2.1 Proof systems

A *proof system* as defined by Cook and Reckhow [17] for some non-empty language $\mathcal{L}$ is a polynomial-time computable function on strings whose range is exactly $\mathcal{L}$. Intuitively f maps proofs to valid theorems, non-proofs can be mapped to some arbitrary known element in $\mathcal{L}$. Soundness comes from well-definition, and completeness is from surjectivity. Many proof systems are line-based where a finite set of rules govern the derivation of valid inferences in the language until a conclusive line is derived.

Proof complexity measures the *sizes* of proofs, i.e. number of characters in the proof string. In a line based proof system, where we can distinguish individual lines we can measure the proof *length*- the number of lines. Where lines are clauses, the clause *width* is the number of literals in the clause. Given a proof system g for language $\mathcal{L}$ we say that proof system f *p-simulates* g if there is a polynomial time procedure p that maps g proofs to f proofs such that $f(p(\pi)) = g(\pi)$ for all g-proofs. We do not require f to necessarily be a proof system for $\mathcal{L}$, but a proof system for $\mathcal{L}' \supseteq \mathcal{L}$. In this paper we do this when we p-simulate QBF proof systems with DQBF proof systems.

Frege systems are "text-book" style line-based proof systems for propositional logic. They consist of a finite, sound and complete set of axioms and rules where any variable can be substituted with any formula. Fig. 3 gives an example of a Frege system.

$$\frac{}{1} \qquad \frac{}{x_1 \to (x_2 \to x_1)} \qquad \frac{}{((x_1 \to 0) \to 0) \to x_1}$$

$$\frac{}{(x_1 \to (x_2 \to x_3)) \to ((x_1 \to x_2) \to (x_1 \to x_3))} \qquad \frac{x_1 \qquad x_1 \to x_2}{x_2}$$

$$\frac{}{(x_1 \to 0) \to \neg x_1} \qquad \frac{}{\neg x_1 \to (x_1 \to 0)} \qquad \frac{}{(x_1 \vee x_2) \to (x_2 \vee x_1)}$$

$$\frac{}{x_1 \to (x_1 \vee x_2)} \qquad \frac{}{(x_1 \vee x_2) \to (\neg x_1 \to x_2)}$$

$$\frac{}{(x_1 \to x_2) \to (\neg x_1 \vee x_2)} \qquad \frac{}{(x_1 \wedge x_2) \to x_1}$$

$$\frac{}{(x_1 \wedge x_2) \to x_2} \qquad \frac{}{x_1 \to (x_2 \to x_1 \wedge x_2)}$$

Figure 3 A Frege system for connectives $0, 1, \neg, \to \vee, \wedge$.

The rules will depend on the connectives included, but Cook and Reckhow [17] showed all Frege systems are p-equivalent. For example a Frege system that uses $\vee, \neg, \wedge$ can adopt the following laws without changing the proof complexity:

$$\frac{C}{C \vee p}\ (\text{Weak}) \qquad \frac{C \vee p \qquad C \vee \neg p}{C}\ (\text{Res}) \qquad \frac{C \vee p \qquad C \vee q}{C \vee (p \wedge q)}\ (\text{Distr})$$

2.2 (Dependency) Quantified Boolean Formulas

A *quantified Boolean formula (QBF)* is a propositional formula equipped with Boolean quantifiers: $\forall$ and $\exists$. $\forall x \phi(x) \equiv \phi(0) \wedge \phi(1)$ and $\exists x \phi(x) \equiv \phi(0) \vee \phi(1)$. A QBF in prenex form $\Pi\phi$ contains a propositional *matrix* ϕ which is quantifier-free and a *prefix* $\Pi = \mathcal{Q}_1 x_1 \ldots \mathcal{Q}_k x_k$

where $\mathcal{Q}_i \in \{\forall, \exists\}$ for $1 \leq i \leq k$. A closed QBF requires every variable to be bound to some quantifier in the prefix. We will mainly work with closed prenex QBFs in this paper. For variable x we use $x \in \exists$ to denote that x is existentially quantified somewhere in the prefix, or $x \in \forall$ to denote that x is universally quantified in the prefix. The quantifier order matters, we say that existential variable x depends on u if u in quantified left on x in the prefix. In this way we can build a dependency set D_x of each existentially quantified variable x containing exactly the universal variables that are left of x.

A *Skolem function* for existential variable x is a Boolean function $f_x : D_x \to \{0, 1\}$. This allows us to use alternative semantics to define the truth of a QBF: that a closed prefix QBF is true if and only if there is a set of Skolem functions, one for each existential variable x, such that for every complete assignment to all the universal variables the universal assignment completed with the values of the Skolem functions under that assignment, form a satisfying assignment to the propositional matrix. We call such a set of Skolem functions *winning*. Dually, a closed QBF is false if and only if there is a set of *Herbrand functions*, one for each universal variable y, such that for every complete assignment to all the existential variables the Herbrand functions falsify the propositional matrix.

A Dependency Quantified Boolean Formula (QBF) can be defined and an S-form DQBF uses this notion of Skolem functions as its main semantics. An S-form DQBF $\Pi\phi$ has a prefix $\Pi = \forall u_1 \dots u_p \exists x_1(D_{x_1}) \dots x_q(D_{x_q})$, here the quantifier order does not matter as the dependency sets are explicitly given. Each D_x can be any arbitrary subset of $\{u_1 \dots u_p\}$. A DQBF is true if and only if there is a set of Skolem functions, one for each existential variable x, such that for every complete assignment to all the universal variables the universal assignment completed with the values of the Skolem functions under that assignment, form a satisfying assignment to the proposition matrix. We sometimes write $\forall U \exists E \phi$ for an arbitrary S-form DQBF, where U is the set of universal variables, E the set of existential variables each with their own unspecified dependency set and ϕ a propositional matrix containing no quantifiers. We define D_u for a universal variable to be $\{u\}$. We can also define the dependency set of a clause C as $(\bigcup_{\text{var}(y) \in C} D_y)$ including universal literals.

QBF is a PSPACE-complete language and DQBF is NEXPTIME-complete. We can demonstrate a DQBF is true by exhibiting its Skolem functions as circuits and showing the matrix with the Skolem functions substituted in is a propositional tautology. To show a DQBF is false we can use a DQBF proof system such as DQBF-IR-calc [3] given in Figure 4. We will define a new refutational proof system for DQBF in the next section.

3 IndExtFrege + ∀red, an S-form DQBF proof system

3.1 Independent Extension

Consider the refutational proof system Frege + ∀red in QBF [7], composed of Frege rules and a ∀red rule. Frege rules allow for propositional line based inference and the reduction rule allows a universal variable u to be replaced by a constant 0 or 1, given below:

$$\frac{\Pi\phi \wedge L(u)}{\Pi\phi \wedge L(u) \wedge L(0)} \qquad \frac{\Pi\phi \wedge L(u)}{\Pi\phi \wedge L(u) \wedge L(1)}$$

No variable can appear to the right of u in L. A sound DQBF version exists.

▶ **Lemma 1** (∀red soundness). *Suppose $\Pi\phi \wedge L(u)$ is a true S-form DQBF, and L contains no existential variables x such that u is in the dependency set of x. Then the S-form DQBF $\Pi\phi \wedge L(u) \wedge L(0)$ is true and the S-form DQBF $\Pi\phi \wedge L(u) \wedge L(1)$ is also true.*

Proof. An S-form DQBF is true if and only if it has a set of satisfying Skolem functions, a function σ_x for each of its existential variables x. Suppose $\Pi\phi \wedge L(u)$ is satisfied by the set $\{\sigma_x \mid x \in \exists\}$. We will show that $\Pi\phi \wedge L(u) \wedge L(0)$ is satisfied by the same functions. Consider an arbitrary universal assignment α, ϕ is satisfied by assumption and so is $L(u)$. If $\alpha(u)$ is 0 then $L(u) = L(0)$ is satisfied. Otherwise consider β which is identical to α except on u. $\alpha(u) = 1$ and $\beta(u) = 0$, but the outputs of Skolem functions of the existential variables in L remain unaffected by changing between α and β, only the variable u is affected and so $L(0)$ is satisfied. Therefore $\Pi\phi \wedge L(u) \wedge L(0)$ is satisfied by the same set of Skolem functions as $\Pi\phi \wedge L(u)$. The case with $L(1)$ is symmetrical. ◀

Universal reduction in $L(u)$ is blocked when there are existential variables in L that depend on u. Extension variables are also existential and can end up blocking reduction through excessive dependency. We define extension variables that *conditionally* represent Boolean circuits, for smaller dependency sets. We give two versions, one for conjunction and one for disjunction. We could instead use a single rule based on a functionally complete connective such as NAND (which we do in Section 3.3), but our definitions fit more nicely into the proofs of this paper. Let α be a conjunction of universal literals, and Y is a set of literals, both existential and universal.

$$\frac{\Pi\phi}{\Pi\exists v(D_v)\phi \wedge (\alpha \rightarrow (v \leftrightarrow \bigwedge_{y\in Y} y))} \qquad \frac{\Pi\phi}{\Pi\exists v(D_v)\phi \wedge (\alpha \rightarrow (v \leftrightarrow \bigvee_{y\in Y} y))}$$

The extension variable v is a new variable not appearing in Π, nor in ϕ. D_v is calculated as the union over all $D_{\mathrm{var}(y)}$ for $y \in Y$ and we then subtract the domain of α. This means v is independent of every variable in α and even if some variable x in Y does depend on some variable u, that dependence will be removed if $u \in \alpha$. In the earlier extension rule [19, 7], the variables that extension variable was defined on coincided with its dependency set. In our new rule v can be defined on variables and not receive its full dependency set. Having a smaller dependency set means that v prevents fewer reduction steps. For why this is permissible, we can think of the equational part of the definition only applying once α is already set, therefore there is no scenario of the α variables where v is required to consider a different input value for these variables, other than in the situation where it must consider α.

The downside of this definition is that substituting a Boolean function b for its extension variable v adds condition α. But because we subtract the domain of α, v no longer blocks the reduction of variables from α so in many cases we can reduce these variables.

▶ **Lemma 2.** *Suppose $\Pi\phi$ is a true S-form DQBF, and $\Pi\exists v(D_v)\phi \wedge (\alpha \rightarrow (v \leftrightarrow b(Y)))$ is constructed according to the Independent Extension rule, where b is a Boolean function. Then $\Pi\exists v(D_v)\phi \wedge (\alpha \rightarrow (v \leftrightarrow b(Y)))$ is a true S-form DQBF.*

Proof. $\Pi\phi$ is a true S-form DQBF, therefore it has Skolem functions σ_x for each existential variable x. This set of Skolem functions satisfies all lines in ϕ, but $\alpha \rightarrow (v \leftrightarrow b(Y))$ may or may not be satisfied. To make sure it is satisfied we use a Skolem function σ_v for v. We apply substitution of α to Y to get $\alpha(Y)$ which assigns some variables to constants. Notice those variables in the domain of α are now constant and no longer in the domain of $\alpha(Y)$. The free variables of $\alpha(Y)$ are those in Y but not α. We define $\sigma_v(D_v) = b(\alpha(Y))$, which works because $b(\alpha(Y))$'s dependency set is $(\bigcup_{y\in Y} D_y) \setminus D_\alpha$. Note that v does not appear in ϕ, so ϕ remains satisfied.

If α is not satisfied then $\alpha \rightarrow (v \leftrightarrow b(Y))$ is automatically satisfied. If α is satisfied then $Y = \alpha(Y)$. If $b(Y)$ is true then $b(\alpha(Y))$ is true and so σ_v is true which makes v true, so $v \leftrightarrow b(Y)$ is satisfied. Likewise if $b(Y)$ is false, then $b(\alpha(Y))$ is false so σ_v and thus v are false, therefore $v \leftrightarrow b(Y)$ is satisfied. ◀

Axiom rule: $\mathsf{axiom}(\phi)$

$$\frac{}{\{a^{\tau|_{D_a}} \mid a \in C, \mathrm{var}(a) \in \exists\}}$$

- C is a clause in the matrix ϕ.
- $\tau|_{D_a} := \{\neg l \mid l \in C, \mathrm{var}(l) \in \forall, \mathrm{var}(l) \in D_a\}$.

Instantiation rule: $\mathsf{inst}(C, \beta)$

$$\frac{C}{\{a^{(\alpha \circ \beta)|_{D_a}} \mid a^\alpha \in C\}}$$

- β is a partial assignment to universal variables.
- $\alpha \circ \beta$ is $\{l \mid (l \in \alpha) \vee (l \in \beta \wedge \bar{l} \notin \alpha)\}$.
- $(\alpha \circ \beta)|_{D_a} := \{l \in \alpha \circ \beta \mid \mathrm{var}(l) \in D_a\}$.

Resolution rule: $\mathsf{res}(C_1, C_2, x^\tau)$

$$\frac{C_1 \qquad C_2}{(C_1 \cup C_2) \setminus \{x^\tau, \neg x^\tau\}}$$

- $x^\tau \in C_1$ and $\neg x^\tau \in C_2$.

Figure 4 Proof rules of DQBF-IR-calc [3] .

3.2 A Sound and Complete Proof System

We define our new proof system IndExtFrege + ∀red in Figure 5. The proof system works as a refutational proof typically does, starting with a DQBF $\forall U \exists E \phi$. ϕ is a propositional formula. For QBFs, IndExtFrege + ∀red automatically generalises eFrege + ∀red, but since we also desire DQBF completeness only p-simulation of a complete DQBF proof system suffices, here we choose DQBF-IR-calc given in Figure 4. DQBF-IR-calc works by removing all universal literals from the formula and replacing each existential literal with an annotated literal, the annotations are partial universal assignments. The idea is that you can remove universal quantifiers by expansion, but you create multiple copies of the inner existential variables, so the annotations track which expansions have led to this literal. A p-simulation works because these annotated variables can be defined by independent extension, where they drop the dependence on universal variables that have been expanded on.

▶ **Theorem 3.** *There is an $O(w^2 l)$ IndExtFrege + ∀red p-simulation of DQBF-IR-calc. Where l is the number of lines in π, w is the size of the largest clause $w = \max_{C \in \pi}(\sum_{l^\alpha \in C} 1 + |\alpha|)$ (π being the DQBF-IR-calc proof).*

Proof. DQBF-IR-calc uses existential annotated variables. For each annotated variable x^α appearing in the DQBF-IR-calc proof we introduce definition clauses $\bar{\alpha} \vee x^\alpha \vee \bar{x}$ and $\bar{\alpha} \vee \bar{x}^\alpha \vee x$ based on $\alpha \to (x^\alpha \leftrightarrow x)$. We therefore add $O(wl)$ many clauses each of width $O(w)$.
Axiom: in DQBF-IR-calc an axiom involves some instantiation of a clause C. τ is the partial assignment that contradicts the universal literals in C. We replace each existential literal l with $l^{\tau|_{D_l}}$ where $\tau|_{D_l}$ restricts τ's domain to variables in $D_{\mathrm{var}(l)}$. We obtain this by resolving with $\bar{\tau}|_{D_l} \vee l^{\tau|_{D_l}} \vee \bar{l}$. We accumulate universal literals from $\bar{\tau}$ in our axiom, but these can be reduced now there are no existential literals that block these literals.
Res: The resolution step is easy to p-simulate as Frege p-simulates resolution.
Inst: The final rule allows us to instantiate to increase the universal annotation uniformly everywhere in a clause. One $\mathsf{inst}(C, \beta)$ of size $O(w)$ can be simulated by lines that total size $O(w^2)$. Instantiation may replace literal x^α with $x^{\alpha \sqcup \beta'}$ for some $\beta' \subseteq \beta$. We already have definitions $\bar{\alpha} \vee \bar{x}^\alpha \vee x$, $\bar{\alpha} \vee \bar{\beta}' \vee x^{\alpha \sqcup \beta'} \vee \bar{x}$. Resolving over x gets us $\bar{\alpha} \vee \bar{\beta}' \vee x^{\alpha \sqcup \beta'} \vee \bar{x}^\alpha$. Neither x^α nor $x^{\alpha \sqcup \beta'}$ have α's variables in its dependency set, so we can now reduce the literals of $\bar{\alpha}$ to get $\bar{\beta}' \vee x^{\alpha \sqcup \beta'} \vee \bar{x}^\alpha$.

All rules and axioms, for any Frege system of choice.

$$\frac{}{L}\ \text{(Axiom)}$$

L is a conjunct in the propositional matrix ϕ.

$$\frac{}{(\alpha \to (v \leftrightarrow \bigwedge_{y\in Y} y))}\ \text{(IndExt-}\wedge\text{)} \qquad \frac{}{(\alpha \to (v \leftrightarrow \bigvee_{y\in Y} y))}\ \text{(IndExt-}\vee\text{)}$$

v is a fresh $\exists$ variable, α is a conjunction of $\forall$ literals. $D_v = (\bigcup_{y\in Y} D_y) \setminus D_\alpha$.

$$\frac{L(u)}{L(0)}\ \text{(0-red)} \qquad \frac{L(u)}{L(1)}\ \text{(1-red)}$$

u is a $\forall$ variable. There is no $\exists$ variable x in $\mathrm{var}(L)$ such that $u \in D_x$.
As an additional rule, the prefix Π may be weakened to Π' to add a new variable that does not appear in the matrix.

Figure 5 Proof rules of IndExtFrege + ∀red.

We use these clauses to instantiate via resolving $O(w)$ many times. β' is necessarily a sub assignment of β, but we may accumulate any of the literals of $\bar{\beta}$ in our clause. Instantiation means that there will be no existential variable remaining that will depend on the variables of β. And so all universal literals that accumulate can be reduced. The lines are of $O(w)$ size and we involve $O(w)$ many of them to simulate this line. ◀

▶ **Example 4.** Let our DQBF prefix be $\forall uvw \exists a(u,v) b(w)$. Suppose we have the instantiation step $\mathsf{inst}(a^u \vee b, v\bar{w})$. In a p-simulated DQBF-IR-calc proof we already have some clauses that define annotated variables a^u, a^{uv}, $b^{\bar{w}}$. We can resolve $(\bar{u} \vee \bar{a}^u \vee a)$ and $(\bar{u} \vee \bar{v} \vee a^{uv} \vee \bar{a})$ to get $(\bar{u} \vee \bar{v} \vee \bar{a}^u \vee a^{uv})$, now we reduce $u = 1$ to get $(\bar{v} \vee \bar{a}^u \vee a^{uv})$. Using $(\bar{v} \vee \bar{a}^u \vee a^{uv})$ and $a^u \vee b$ we get $\bar{v} \vee a^{uv} \vee b$, we resolve again with $(w \vee b^{\bar{w}} \vee \bar{b})$ to get $\bar{v} \vee w \vee a^{uv} \vee b^{\bar{w}}$. We then reduce with $v = 1, w = 0$ to get $a^{uv} \vee b^{\bar{w}}$ which is exactly what $\mathsf{inst}(a^u \vee b, v\bar{w})$ becomes under our prefix.

Notice that unlike previous simulations of IR-calc such as the simulation of IR-calc by eFrege + ∀red [16], we are not formalising the strategy, but going line-by-line and replicating each line and its original semantic meaning. In QBF IR-calc relies on the base propositional inference rule being as weak as resolution for strategy extraction to be possible, but here we can use stronger forms of inference on instantiated clauses.

▶ **Corollary 5.** IndExtFrege + ∀red *is refutationally complete for S-form DQBFs.*

Proof. Technically, simulating DQBF-IR-calc only shows this for DQBF with CNF matrices. Any propositional formula can be transformed into a logically equivalent CNF through enumerating all falsifying assignments to ϕ. This can be done with Frege rules and the refutation can then proceed by simulating DQBF-IR-calc. ◀

▶ **Theorem 6.** IndExtFrege + ∀red *is a sound refutational proof system.*

Proof. We claim that if DQBF $\forall U \exists E \phi$ is true and a number of lines $L_1 \dots L_n$ are derived by IndExtFrege + ∀red from $\forall U \exists E \phi$, then $\forall U' \exists E' \phi \wedge L_1 \wedge \dots \wedge L_n$ is also a true DQBF, where E' and U' extend the prefix only to include the new variables added by prefix weakening or

the Independent Extension rules from $L_1 \dots L_n$. We can prove that if $\forall U' \exists E' \phi \wedge L_1 \wedge \dots \wedge L_n$ has a set of Skolem functions that satisfies all conjuncts then $\forall U'' \exists E'' \phi \wedge L_1 \wedge \dots \wedge L_{n+1}$ has a set of Skolem functions that satisfies all conjuncts (where E'' and U'' extend E' and U', respectively to include the new variables of L_{n+1}). Axiom and Frege rules and reduction rules preserve winning Skolem functions. For Axiom and Frege this is easy to see. Since these rules preserve models in propositional logic they preserve whether a Skolem function satisfies all lines. For reduction, if every line is satisfied by the Skolem functions, this includes $L(u)$, and this does not change under both values of u and factoring in these values does not necessitate updating any of the Skolem functions of the variables of L because they do not depend on u (Lemma 1). The IndExt axioms preserve DQBF truth by Lemma 2. ◀

In the following example, we demonstrate that IndExtFrege + ∀red is conditionally strictly stronger than eFrege + ∀red.

▶ **Example 7.** Let X be the set of variables $\{x_1, \dots, x_{2n}\}$ and $\phi(X)$ a CNF in the variables of X. Then $\Pi\phi(X)$ with prefix $\Pi = \forall x_1 \exists x_2 \forall x_3 \dots \exists x_{2n}$ is a closed PCNF. We also define a second set of mirrored variables $X' = \{x'_1, \dots, x'_{2n}\}$. We can define two false QBF families:

$\mathsf{Duality}(\Pi\phi) = \exists x'_1 \forall x_1 \exists x_2 \forall x'_2 \dots \exists x'_{2n-1} \forall x_{2n-1} \exists x_{2n} \forall x'_{2n} \phi(X) \wedge \neg\phi(X')$,

$\mathsf{Select}(\Pi\phi) = \forall u \exists x'_1 \forall x_1 \exists x_2 \forall x'_2 \dots \exists x_{2n} \forall x'_{2n} (\phi(X) \vee u) \wedge (\neg\phi(X') \vee \neg u)$.

Due to the prefix ordering, Duality has an easy strategy for each variable, but Select has a PSPACE-hard strategy for its first variable u. It was shown [13] that using the easy strategy Duality always has a short proof in eFrege + ∀red. Because of strategy extraction, Select cannot have short proofs in eFrege + ∀red unless PSPACE ⊂ P/poly, but they have short proofs in IndExtFrege + ∀red.

For each variable x_i we create new variable y_i, if x_i is existential in Π then we use definition $\neg u \to (y_i \leftrightarrow x_i)$ if x_i is universal then we use definition $u \to (y_i \leftrightarrow x'_i)$. These variables have a dependency subset of x_i and x'_i, respectively. Importantly, they are independent of u, much like in the Duality formula where there is no u variable. Now we replace all existential variables x_i and x'_i in $(\phi(X) \vee u) \wedge (\neg\phi(X') \vee \neg u)$ with y_i variables. This gives us two conjuncts $(\phi(Y) \vee u)$ and $(\neg\phi(Y') \vee \neg u)$ where $Y = \{y_i \mid \exists x_i \in \Pi\} \cup \{x_i \mid \forall x_i \in \Pi\}$ and $Y' = \{y_i \mid \exists x'_i \in \Pi'\} \cup \{x'_i \mid \forall x'_i \in \Pi'\}$. The conditional part of the definition is absorbed by the $\vee u$ and $\vee \neg u$ part, respectively. In each conjunct, u can be reduced due to independence from all the existential variables. Therefore we have $\phi(Y) \wedge \neg\phi(Y')$, and structurally we have the $\mathsf{Duality}(\Pi\phi)$ formula, so we simply proceed with the short eFrege + ∀red proof to get a short proof. Because these proofs are short and uniform, IndExtFrege + ∀red cannot have polynomial time strategy extraction unless P = PSPACE and so eFrege + ∀red cannot p-simulate IndExtFrege + ∀red unless P = PSPACE.

3.3 A Resolution Version

We can better demonstrate the simplicity of the new rule by defining an equivalent resolution system that uses only four clausal rules. We give the proof system in Figure 6.

▶ **Theorem 8.** IndExtQURes *and* IndExtFrege + ∀red *are p-equivalent.*

Proof. ($\Leftarrow$) We p-simulate each individual rule. (Ax) in IndExtQURes is a straightforward applications of (Ax) in IndExtFrege + ∀red. (Red) in IndExtQURes can be p-simulated using $(0 - \mathsf{red})$ or $(1 - \mathsf{red})$, depending on whether the reduced literal was positive of negative, respectively. Using Frege we can say a literal equal to 0 or $\neg 1$ is equivalent to it being removed. It is well-known that the resolution rule is p-simulated by Frege. For the (IndExt) rule we us the (IndExt-$\vee$) rule in Figure 5 on $Y = \{\bar{y}_1, \bar{y}_2\}$. Then each of the three clauses

$$\frac{}{L}\ (\text{Ax}) \qquad \frac{C \vee u}{C}\ (\text{Red}) \qquad \frac{E \vee \neg x \qquad F \vee x}{E \vee F}\ (\text{Res})$$

L is a clause in the propositional matrix ϕ. u is a $\forall$ literal. There is no $\exists$ literal l in C such that $\text{var}(u) \in D_{\text{var}(l)}$, and there is no $\bar{u} \in C$.

$$\frac{}{(\bar{\alpha} \vee v \vee y_1), (\bar{\alpha} \vee v \vee y_2), (\bar{\alpha} \vee \bar{v} \vee \bar{y}_1 \vee \bar{y}_2)}\ (\text{IndExt})$$

v is a fresh $\exists$ variable, α is a conjunction of $\forall$ literals. $D_v = (D_{y_1} \vee D_{y_2}) \setminus D_\alpha$.

As an additional rule, the prefix Π may be weakened to Π' to add a new variable that does not appear in the matrix.

Figure 6 Proof rules of IndExtQURes.

for (IndExt) in Figure 6 is a propositional implicant of the formula derived by (IndExt-$\vee$) from Figure 5 and can be derived using a short Frege proof. Finally whenever the empty clause $\bot$ is derived in IndExtQURes, the final steps either use a resolution step or a reduction step. In either case we can p-simulate and derive 0 instead.

($\Rightarrow$) we interpret every line L_i in IndExtFrege + $\forall$red as an extension variable l_i that is built as the circuit for the formula in the line. All Frege rules and axioms can be p-simulated by the well know p-simulation of Ext Frege by Ext Res. The axiom rule for IndExtFrege + $\forall$red will technically have to be p-simulated by deriving a singleton clause with the variable l representing the disjunction, but every literal x in the axiom clause will be used in a definition clause $\neg x \vee l$, and so we resolve away the literals until we get singleton l.

For (IndExt-$\wedge$) and (IndExt-$\vee$), consider Figure 5 where we define a new variable v we represent the $\bigwedge_{y \in Y} y$ and $\bigvee_{y \in Y} y$ formulas from Figure 5 with an extension variable p, we take the variables y_1 and y_2 from Figure 6 to both be $\bar{p}$ and then we define v using (IndExt) (using the same α). Once we define extension variable q with $l \leftrightarrow (\alpha \rightarrow (v \leftrightarrow w))$ we can resolve to get singleton l.

Suppose we perform a reduction from $L(u)$ to $L(0)$ in IndExtFrege + $\forall$red. Let us label $L(u)$ as p and $L(0)$ as q. $L(u) \wedge \bar{u} \rightarrow L(0)$ is an obvious propositional tautology and we use the p-simulation of Ext Frege by Ext Res to derive clause $\bar{p} \vee u \vee q$, as we do not have weakening in IndExtQURes, we may obtain a stronger clause, which is just as useful. Once we obtain singleton p we resolve it to get $q \vee u$. u is not blocked by q and so we reduce to get q, or even the empty clause (which saves us from all subsequent lines). This works symmetrically for $(1 - \text{red})$ as it does for $(0 - \text{red})$

And finally if we derive l which represents the empty clause from IndExtFrege + $\forall$red, we simply resolve with $\neg l$ which is part of the definition of l to get the empty clause in IndExtQURes. ◀

4 P-simulations

In order to show completeness we have already shown that IndExtFrege + $\forall$red p-simulates DQBF-IR-calc. We can show more p-simulations that demonstrate the power of this new extension rule, including Fork Resolution, QRAT, and $\mathcal{D}^{\text{rrs}}$ based systems. With these p-simulations we show that IndExtFrege + $\forall$red can capture the vast majority of (D)QBF solving and preprocessing techniques.

These p-simulations also demonstrate that IndExtFrege + ∀red is powerful. A proof system is roughly as powerful as the most expressive object it can "cut" on. Resolution cuts on literals, Frege cuts on propositional formulas, bounded-depth Frege cuts on bounded depth formulas and G cuts on QBFs. The extension variables in eFrege and Ext Res cut on propositional circuits, as is the case in eFrege + ∀red. IndExtFrege + ∀red allows extension variables to be more expressive and by combining them we can express PSPACE-hard objects, and we make use of this in the p-simulations of this section.

4.1 P-Simulation of Fork Resolution

In this section we show how IndExtFrege + ∀red can p-simulate another DQBF proof system that has a different notion of extension variables.

The Fork Resolution proof system is sound and complete for DQBFs that have a CNF matrix. Its main novelty is a Fork Extension rule [26], which is used for splitting clauses.

$$\frac{C_1 \cup C_2}{e \vee C_1 \quad \bar{e} \vee C_2}\ (\text{F-Ext}) \qquad \frac{x \vee C_1 \qquad \bar{x} \vee C_2}{C_1 \cup C_2}\ (\text{Res}) \qquad \frac{C \vee u}{C}\ (\forall\text{red})$$

e is a new $\exists$-variable and has dependency set $(\bigcup_{\text{var}(y)\in C_1} D_y) \cap (\bigcup_{\text{var}(y)\in C_2} D_y)$. Fork Resolution also uses a resolution rule and a reduction rule: provided $\bar{u}$ is not in C and var(u) is not in the dependency set of C.

Given an instance of the Fork Extension on $C_1 \cup C_2$, the idea is to get an extension variable x that is equivalent to $\forall\{u \in U \mid u \in (\bigcup_{y\in C_2} D_y), u \notin \bigcup_{y\in C_1} D_y)\}C_2$. In other words x is true if C_2 is true under all assignments to the specific universal variables that govern C_2 but not C_1.

It should be clear why this should mean $C_1 \cup C_2 \to \bar{x} \vee C_2$, because x is a stronger version of C_2. $C_1 \cup C_2$ also implies $x \vee C_1$ is true because if a Skolem function always satisfies $C_1 \cup C_2$, then C_2 must be satisfied, whenever C_1 is falsified regardless of the values of any remaining dependencies of C_2.

▶ **Lemma 9.** IndExtFrege + ∀red *can p-simulate the Fork Extension rule (F-Ext).*

Proof. We order all universal variables in the *marginal* dependency set of C_2, $\{u_1, \ldots, u_k\} = \{u \in U \mid u \in (\bigcup_{y\in C_2} D_y), u \notin (\bigcup_{y\in C_1} D_y)\}$. We define:

$$e_0 \leftrightarrow C_2,\ \ u_{i+1} \to (e_i^{u_{i+1}} \leftrightarrow e_i),\ \ \bar{u}_{i+1} \to (x_i^{\bar{u}_{i+1}} \leftrightarrow e_i),\ \ e_{i+1} \leftrightarrow (e_i^{u_{i+1}} \wedge e_i^{\bar{u}_{i+1}})$$

And we can use the extension clauses that give these definitions. We make the induction hypothesis that there is a short proof of $e_i \vee C_1$ and of $\bar{e}_i \vee C_2$ in IndExtFrege + ∀red. Starting from $i = 0$ and incrementing until $i = k$.

Base Case. We resolve off all the C_2 literals in $C_1 \vee C_2$ to get $C_1 \vee e_0$. $\bar{e}_0 \vee C_2$ is part of the definition of $e_0 = C_2$.

Inductive Step. We start with $C_1 \vee e_i$ and $\bar{e}_i \vee C_2$. Resolving $\bar{u}_{i+1} \vee e_i \vee \bar{e}_i^{u_{i+1}}$ with $u_{i+1} \vee e_i \vee \bar{e}_i^{\bar{u}_{i+1}}$ gets us $e_i \vee \bar{e}_i^{u_{i+1}} \vee \bar{e}_i^{\bar{u}_{i+1}}$ and then with $\bar{e}_{i+1} \vee e_i^{u_{i+1}}$ and $\bar{e}_{i+1} \vee e_i^{\bar{u}_{i+1}}$, we get that $e_i \vee \bar{x}_{i+1}$. We can then, with a resolution step, get $\bar{e}_{i+1} \vee C_2$

In $C_1 \vee e_i$ we replace literal e_i with $\bar{u}_{i+1} \vee e_i^{u_{i+1}}$. Since $\bar{u}_{i+1}$ is not blocked by any literal in C_1 and not blocked by $e_i^{u_{i+1}}$ it can be reduced to derive $C_1 \vee e_i^{u_{i+1}}$, $C_1 \vee x_i^{\bar{u}_{i+1}}$ is derived in the same way. We resolve both with $e_{i+1} \vee \bar{e}_i^{u_{i+1}} \vee \bar{e}_i^{\bar{u}_{i+1}}$ to get $C_1 \vee e_{i+1}$

This DAG-like process is linear in the number of induction steps, potentially multiplied by the width of $C_1 \cup C_2$. To finalise, we take $e = e_k$, and necessarily it is not dependent on variables not in the dependency set of C_2 because of the initial definition of e_0 and any variable not in the dependency set of C_1 is removed by the time we get to e_k. ◀

▶ **Corollary 10.** IndExtFrege + ∀red *p-simulates Fork Resolution.*

Unlike in eFrege + ∀red where new variables are essentially propositional circuits, here we show how a new variable can be efficiently constructed to resemble a QBF properly. IndExtFrege + ∀red shares the ability to cut over a QBF with G, which goes some way to explain why IndExtFrege + ∀red is so powerful.

4.2 P-simulation of QRAT

In this section with show our DQBF proof system IndExtFrege + ∀red can p-simulate the QBF proof system QRAT [18], which governs many preprocessing steps in QBF.

4.2.1 Definition of QRAT

In propositional logic, interference based proof systems modify a CNF into another equisatisfiable CNF without preserving models. Resolution, is not an interference based proof system because it preserves logical equivalence. An example of an interference based proof system is DRAT [31], where clause addition can destroy an existing model, but only when there is a guarantee that another model exists and is preserved. QRAT has similar rules to DRAT, but since QRAT is a QBF proof system, quantifier order must be respected. Since QRAT works in QBF, the notion of preserving individual models, is understood as preserving Skolem functions. QRAT does not preserve Skolem functions.

▶ **Definition 11** ([10])**.** *Fix a QBF prefix* Π*. Now consider a clause* D *and a literal* l *(not necessarily in* D*) we define,* $O_D = \{k \in D \mid \mathrm{lv}(k) \leq \mathrm{lv}(l), \mathrm{var}(k) \neq \mathrm{var}(l)\}$*, where* $\mathrm{lv}(k) \leq \mathrm{lv}(l)$ *indicates that* k *has a lower quantifier level than* l*, i.e* k *is left of* l *or, because we include equality, both are part of the same quantifier block (i.e. same dependency set).* O_D *is called the* outer clause*. For a DQBF we define the outer clause* O_D *differently for existential and universal literals* l*.*

When l *is existential:* $O_D := \{x \in D \mid x \in \exists, D_{\mathrm{var}(x)} \subseteq D_{\mathrm{var}(l)}, \mathrm{var}(x) \neq \mathrm{var}(l)\} \cup \{x \in D \mid x \in \forall, \mathrm{var}(x) \in D_{\mathrm{var}(l)}\}$

When l *is universal we define a set of existential variables* $\mathcal{S} = \{x \in \exists \mid \mathrm{var}(l) \in D_x\}$ *from that we define a set of universal variables* $T = \bigcap_{x \in S} D_x \setminus \{\mathrm{var}(l)\}$*, and we can then define the outer clause of* D*:* $O_D := \{x \in D \mid x \in \exists, D_{\mathrm{var}(x)} \subseteq T\} \cup \{x \in D \mid x \in \forall, \mathrm{var}(x) \in T\}$*.*

The outer clause is a concept used to evaluate whether new clauses can be added, as it is the critical part that decides soundness. This is in the same vein as how extension clauses rely on the outer variables of the definition.

For some rules in QRAT, they can only be applied soundly if some semantic implication is known, but for a proof system we need to know if these implications are true in polynomial time. To do this we use the sound but incomplete unit propagation procedure.

▶ **Definition 12.** $\phi \vdash_1 \bot$ *means we arrive at the empty clause through the following process: For every clause in* ϕ *of size one (in other words a unit clause),* $\{l\}$ *we add* l *to a partial assignment* α *(*$\mathrm{var}(l)$ *is assigned the polarity of literal* l*). Now we remove literals from clauses in* ϕ *whenever they conflict with* α*. This may produce more unit clauses, or even an empty clause, we repeat this process until no more unit clauses are produced.*

When it comes to simulate QRAT, fortunately unit propagation is something that can be easy to p-simulate in a sequence of rules. QRAT's first rule uses unit propagation very directly to infer clauses propositionally, in this rule we do not have to consider quantifier type or order.

▶ **Definition 13** (Asymmetric Tautology Addition (ATA)). *Let ϕ be a CNF with Π a prefix. Let C be a clause not in ϕ. Let Π' be a prefix including the variables of C and ϕ, $\Pi \subset \Pi'$.*

Suppose $\phi \wedge \bar{C} \vdash_1 \bot$. Then we can infer $\Pi'\phi \wedge C$ from $\Pi\phi$.

The next rule, QRATA deals with adding or removing a clause, but this time the Skolem function for a particular existential literal l changes as a result of this rule. This means that QRATA preserves truth but does not necessarily preserve the strategies. QRATA is a generalisation of the extension rule in eFrege + $\forall$red, and as such we have to respect the quantification order.

▶ **Definition 14** (Quantified Resolution Asymmetric Tautology Addition (QRATA)). *Let $\Pi\phi$ be a PCNF with closed prefix Π and CNF matrix ϕ. Let C be a clause not in ϕ. Let Π_1 and Π_2 be disjoint prefixes and x a variable such that $\Pi \subseteq \Pi_1\exists x\Pi_2$ and a literal l with $\mathrm{var}(l) = x$. The difference in prefix is simply to allow new variables coming from $C \vee l$. For every clause $D \in \phi$ with $\bar{l} \in D$ if $\phi \wedge \bar{C} \wedge \bar{l} \wedge \bar{O}_D \vdash_1 \bot$, then we can derive $\Pi_1\exists x\Pi_2\phi \wedge (C \vee l)$ from $\Pi\phi$.*

The next two rules QRATU and EUR remove a universal literal from a clause, with side conditions which we will soon introduce.

$$\frac{\Pi_1\forall u\Pi_2\phi \wedge (C \vee l)}{\Pi_1\forall u\Pi_2\phi \wedge C}\ (\mathrm{QRATU}) \qquad \frac{\Pi_1\forall u\Pi_2\phi \wedge (C \vee l)}{\Pi_1\forall u\Pi_2(\phi \wedge C)}\ (\mathrm{EUR})$$

For QRATU the condition is similar to that of QRATA but instead of adding a blocked clause over an existential literal, it removes a blocked universal literal. EUR has a different condition based on resolution paths.

▶ **Definition 15** (Quantified Resolution Asymmetric Tautology Universal (QRATU)). *Let $\Pi_1\forall u\Pi_2\phi$ be a PCNF with closed prefix $\Pi_1\forall u\Pi_2$ and CNF matrix ϕ. Let $C \vee l$ be a clause with universal literal l, with $\mathrm{var}(l) = u$. If for every $D \in \phi$ with $\bar{l} \in D$, $\phi \wedge \bar{C} \wedge \bar{O}_D \vdash_1 \bot$, then we can derive $\Pi_1\forall u\Pi_2\phi \wedge C$ from $\Pi_1\forall u\Pi_2\phi \wedge (C \vee l)$.*

For EUR, we consider potential resolution steps connecting clauses to one another. Clauses being in different connected components indicate independence from one another and we can expand on this idea to calculate when a universal literal is locally pure and can only be considered one polarity.

▶ **Definition 16.** *Consider a CNF ϕ and subset χ of clauses in ϕ and a subset $\mathcal{S}$ of variables. $\mathfrak{L}(\phi, \chi, \mathcal{S})$ lists the $\mathcal{S}$-*literals on the resolutions paths from χ *and $\mathfrak{C}(\phi, \chi, \mathcal{S})$ lists* the clauses on the resolution paths from χ. *These are found using an iterative procedure until reaching a fix-point.*

Initialisation. *We start with the clauses in χ and the $\mathcal{S}$ literals in those clauses. $\mathfrak{L}(\phi, \chi, \mathcal{S}) \leftarrow \{l \mid$ there is $C \in \chi$ s.t. $l \in C, \mathrm{var}(l) \in \mathcal{S}\}$ and $\mathfrak{C}(\phi, \chi, \mathcal{S}) \leftarrow \chi$.*

Adding a clause. *If there if some $D \in \phi$ such that $\bar{p} \in D$ and $p \in \mathfrak{L}(\phi, \chi, \mathcal{S})$, then we can update $\mathfrak{L}(\phi, \chi, \mathcal{S})$ and $\mathfrak{C}(\phi, \chi, \mathcal{S})$. $\mathfrak{L}(\phi, \chi, \mathcal{S}) \leftarrow \mathfrak{L}(\phi, \chi, \mathcal{S}) \cup \{q \in D \mid q \neq \bar{p}, \mathrm{var}(q) \in \mathcal{S}\}$ and $\mathfrak{C}(\phi, \chi, \mathcal{S}) \leftarrow \mathfrak{C}(\phi, \chi, \mathcal{S}) \cup \{D\}$. We continue this until we reach fix-point, in other words for all $p \in \mathfrak{L}(\phi, \chi, \mathcal{S})$ if $D \in \phi$ and $\bar{p} \in D$, then $\{q \in D \mid q \neq \bar{p}, \mathrm{var}(q) \in \mathcal{S}\} \subset \mathfrak{L}(\phi, \chi, \mathcal{S})$ and $D \in \mathfrak{C}(\phi, \chi, \mathcal{S})$. Fix-point is reached in polynomial time.*

▶ **Definition 17.** *Let $\Pi_1\forall u\Pi_2\phi$ be a PCNF with closed prefix $\Pi_1\forall u\Pi_2$ and CNF matrix ϕ. Let $C \vee l$ be a clause with universal literal l, with $\mathrm{var}(l) = u$.*

If the resolution path $\mathfrak{C}(\phi \wedge C, C, \mathcal{S})$ contains no clause D such that $\bar{l} \in D$, when $\mathcal{S}$ is the set of existential variables right of l in the prefix (i.e. in Π_2), then we can derive $\Pi_1\forall u\Pi_2(\phi \wedge C)$ from $\Pi_1\forall u\Pi_2\phi \wedge (C \vee l)$.

The refutational system QRAT includes ATA, QRATA, QRATU, EUR rules as well as arbitrary clause deletion and that the prefix allows for new variables to be added.

4.2.2 Interference-based Reasoning via Independent Extension Variables

Our p-simulation is somewhat surprising, as QRAT includes a complicated Extended Universal Reduction rule that performs reductions according to global resolution path connectivity between clauses. On the surface level IndExtFrege + ∀red would seem to be ill-equipped to talk globally about a formula, as it does not have any global conditions. In fact IndExtFrege + ∀red is monotonic in the sense that no deletion rule is necessary. The way we p-simulate clause deletion is simply to ignore clauses that would have been in the QRAT proof. This has a small technicality, QRAT can eliminate a variable and then introduce it again with a different meaning, but in IndExtFrege + ∀red we have to take care to create new variables when we refresh the meanings of variables. In fact when we use an interference based rule such as QRATA, QRATU or EUR we create a new copy of the formula.

We begin with p-simulations of DQBF generalisations of ATA and QRATA, in fact we do not need the independence power of the new Independent Extension rule for this.

▶ **Lemma 18** ([21]). *Suppose we have a CNF ϕ and a clause $C \notin \phi$. If $\phi \wedge \bar{C} \vdash_1 \bot$ then there is a short resolution + weakening proof starting from ϕ to $\phi \wedge C$.*

Proof. Any simplification of a clause can be derived via resolution. If we have a unit clause x and clause $C \vee \bar{x}$, then the clause C can be derived in a single step. The number of literals removed before the empty clause is bounded above by the total number of individual literals. Hence the short proof. ◀

▶ **Lemma 19.** *Suppose we have a DQBF: $\forall U \exists E \phi$. Let l be a literal of some existential variable in E and C be a clause not in ϕ, where C contains literal l.*

With respect to existential literal/variable l, recall the DQBF definition of an outer clause O_K of some clause K, according to Definition 11. If for every clause $D \in \phi$ with $\bar{l} \in D$ satisfies: $\phi \wedge \bar{C} \wedge \bar{O}_D \wedge \bar{l} \vdash_1 \bot$, then using a polynomial bounded (in the size of $\forall U \exists E \psi$) number of rules of IndExtFrege + ∀red *we can extend $\exists E$ to $\exists E'$, which includes a new variable l' such that $D_{l'} \subseteq D_{\mathrm{var}(l)}$ and derive the clauses of $(\phi \wedge C)'$ which is a copy of $\phi \wedge C$ where every l literal in $\mathcal{S}$ is replaced by l' and every $\bar{l}$ literal in $\mathcal{S}$ is replaced by $\bar{l}'$.*

Proof. The aim is to replace l with the substitution $l' = l \vee \bigwedge_{D \in \phi}^{\bar{l} \in D} O_D$ and this will respect all existing clauses and add C'. l' has the same dependency set as l because of the care taken when selecting literals for the outer clause. We can prove the clauses of ϕ' by cases:

1. Clauses $K \in \phi$ that do not contain l nor $\bar{l}$ can remain unchanged in ϕ'.
2. Clauses $K \in \phi$ that contain l but not $\bar{l}$ can be weakened to $K \vee \bigwedge_{D \in \phi}^{\bar{l} \in D} O_D$ and the replacement of $l \vee \bigwedge_{D \in \phi}^{\bar{l} \in D} O_D$ with l' gains us $K' = K \setminus \{l\} \cup \{l'\}$.
3. Clauses $K \in \phi$ that contain $\bar{l}$ but not l have the property that the tautology $O_K \vee \bar{O}_K$ implies $K \setminus \{\bar{l}\} \vee \bigvee_{D \in \phi}^{\bar{l} \in D} \bar{O}_D$. Combine with a conjunction of K we get $K \setminus \{\bar{l}\} \vee (\bar{l} \wedge \bigvee_{D \in \phi}^{\bar{l} \in D} \bar{O}_D)$ and now can replace $(\bar{l} \wedge \bigvee_{D \in \phi}^{\bar{l} \in D} \bar{O}_D)$ with $\bar{l}'$ to get $K' = (K \setminus \{\bar{l}\}) \cup \{\bar{l}'\}$.
4. Clauses $K \in \phi$ that contain both l and $\bar{l}$ then since $l' \vee \bar{l}'$ is also tautological we can derive $K' = K \setminus \{l, \bar{l}\} \cup \{l', \bar{l}'\}$.

We use the short proof of $\phi \to C \vee l \vee \bigwedge_{D \in \phi}^{\bar{u} \in D} O_D$ from unit propagation to derive $C \vee l \vee \bigwedge_{D \in \phi}^{\bar{u} \in D} O_D$ and then replace $l \vee \bigwedge_{D \in \phi}^{\bar{u} \in D} O_D$ with l' to get $C' \vee l'$. ◀

Instead of p-simulating EUR and QRATU separately, what we will show here is a simulation of a powerful mixed reduction rule that acts like a combination of QRATU and EUR and even works for DQBF. QRATU and EUR will both be special cases when the prefix gives a QBF.

▶ **Lemma 20.** *Suppose we have a DQBF: $\forall U \exists E \psi$. Let u be a literal of some universal variable in U and $C \vee u$ be a clause in ψ, i.e $\psi = \phi \wedge (C \vee u)$. Recall Definition 11 for the definition of $\mathcal{S}$ Let $\chi_0 = \mathfrak{C}(\psi, C \vee u, \mathcal{S})$, the set of clauses reachable from $C \vee u$ via resolution path in $\mathcal{S}$.*

If for every clause $D \in \chi_0$ with $\bar{u} \in D$ satisfies: $\phi \wedge \bar{C} \wedge \bar{O}_D \vdash_1 \bot$, then using a polynomial bounded (in the size of $\forall U \exists E \psi$) number of rules of IndExtFrege *we can extend $\exists E$ to $\exists E'$, which includes for every variable x in $\mathcal{S}$ a variable x' such that $D_{x'} \subseteq D_x$ and derive the clauses of $(\phi \wedge C)'$ which is a copy of $\phi \wedge C$ where every x in $\mathcal{S}$ is replaced by x'.*

As specified in the lemma the proof is done by replacing literals with different extension variables. The origins of the replacement scheme used here trace back to the simulation of QRAT by G [14], although this simulation here simulates a more powerful rule and works in the DQBF setting. This is despite being seemingly handicapped by only being able to create definitions that are existential variables, whereas G can use complicated functions to witness universal variables. To overcome this we hide functions that previously witness universal variables into new definitions of the existential variables that depend on them.

Proof. We start by introducing the extensions $\bar{u} \to (x^{\bar{u}} \leftrightarrow x)$ and $u \to (x^u \leftrightarrow x)$ for every $x \in \mathcal{S}$. Let $\mathcal{O} = \bigvee_{D \in \chi_0}^{\bar{u} \in D} \bar{O}_D$. For each literal in $L = \mathfrak{L}(\psi, C \vee u, \mathcal{S})$ and their complement we now define $l^* = \left(\bar{u} \vee \bar{\mathcal{O}} \to l^{\bar{u}}\right) \wedge (u \wedge \mathcal{O} \to l^u)$. In other words, l^* is $l^{\bar{u}}$ except in the circumstance that u is true and some outer clause is falsified, and in that case and only that case we have l^* takes the value from l^u. The definition is consistent with negation (i.e. $\neg l^* = (\neg l)^*$). The dependency set of l^* is no greater than that of l.

One may notice that if there were Skolem functions that satisfied $\forall U \exists E \psi$, these Skolem functions extended to the new $*$-variables would also satisfy χ_0^*, which replaces all x variables with x^* in χ_0. This is because if u is false every l^* plays as $l^{\bar{u}}$ which is how l plays anyway. If u is true and some O_D for $D \in \chi_0, \bar{u} \in D$ is falsified, all l^* variables play consistently as l^u which is consistent with l and u and thus satisfies all clauses. If u is true and every O_D for $D \in \chi_0, \bar{u} \in D$ is satisfied, l^* plays as $l^{\bar{u}}$ which will still satisfy every clause in χ_0 except these D, but each of them is already satisfied from their outer clause.

We have to actually construct χ_0^* using rules from IndExtFrege. There are four types of clauses K in χ_0^*. For each we have to construct K^* which replaces every S-literal l with l^* in K: **1.** K contains no u literal. **2.** K contains a positive u literal only. **3.** K that contains a negative u literal only. **4.** K contains a tautology over u.

1. For each $\mathcal{S}$-literal l of K we can use definition $\bar{u} \to (l^{\bar{u}} \leftrightarrow l)$. For K we replace each of these literals l for $l^{\bar{u}} \vee u$ via resolving with the definitions. At the end of this process we can denote this clause as $K^{\bar{u}} \vee u$. There is no literal that blocks u from reduction here so we can derive $K^{\bar{u}}$. We do this similarly with definitions $u \to (l^u \leftrightarrow l)$ to create K^u. $(u \wedge \mathcal{O}) \vee K^* \vee \bar{K}^{\bar{u}}$ is a tautology once the definition of l^* is considered which can be proven in a linear size Frege proof. Likewise $\bar{u} \vee \bar{\mathcal{O}} \vee K^* \vee \bar{K}^u$ is a tautology which can be proven in a linear size Frege proof. And by resolving disjuncts together we are left with K^*.

$$\dfrac{\dfrac{\dfrac{K^{\bar{u}} \vee u}{K^{\bar{u}}} \qquad (u \wedge \mathcal{O}) \vee K^* \vee \bar{K}^{\bar{u}}}{(u \wedge \mathcal{O}) \vee K^*} \qquad \dfrac{\dfrac{K^u \vee \bar{u}}{K^u} \qquad \bar{u} \vee \bar{\mathcal{O}} \vee K^* \vee \bar{K}^u}{\bar{u} \vee \bar{\mathcal{O}} \vee K^*}}{K^*}$$

2. For K, we replace each of these literals l for $l^{\bar{u}} \vee u$. At the end of this process we can denote this clause as $K^{\bar{u}}$, we do not have to include the extra u literal as that is already included. We do this similarly with definitions $u \to (l^u \leftrightarrow l)$ to create $K^u \vee \bar{u}$. The $\bar{u}$ need not be reduced as it will be absorbed into the $\bar{u}$ disjunct later. $(u \wedge \mathcal{O}) \vee K^* \vee \bar{K}^{\bar{u}}$ is a tautology once the definition of l^* is considered which can be proven in a linear size Frege proof. Likewise $(u \vee \bar{\mathcal{O}}) \vee K^* \vee \bar{K}^u$ is a tautology which can be proven in a linear size Frege proof. And by resolving disjuncts together we are left with K^*.
3. We can derive $(u \wedge \bigvee_{D \in \chi_0}^{\bar{u} \in D} \bar{O}_D) \vee K^*$ simply because $\bar{O}_K \vee O_K$ and $u \vee \bar{u}$ are tautological clauses and that K^* which replaces every S-literal l with l^*, contains $\bar{u}$ and O_K as subclauses. The clause is derived using distributivity.
   We replace each of these literals l for $l^u \vee \bar{u}$. At the end of this process we can denote this clause as K^u we do not have to include the extra $\bar{u}$ literal as that is already included. $u \vee \bar{\mathcal{O}} \vee K^* \vee \bar{K}^u$ is a tautology which can be proven in a linear size Frege proof. K^* can be derived by resolving disjuncts.

$$\dfrac{\dfrac{\dfrac{\bar{O}_K \vee O_K}{\mathcal{O} \vee K^*} \qquad \dfrac{\dfrac{u \vee \bar{u}}{u \vee K^*} \qquad \dfrac{K^u \qquad \bar{u} \vee \bar{\mathcal{O}} \vee K^* \vee \bar{K}^{\bar{u}}}{\bar{u} \vee \bar{\mathcal{O}} \vee K^*}}{\bar{\mathcal{O}} \vee K^*}}{K^*}$$

4. K^* is a tautological clause.

We are not yet done, because we have only derived clauses for χ_0 not the whole of ϕ. To do this we need a slightly better replacement scheme than l to l^*. To start with, for $\mathcal{S}$ variables x where neither x nor $\bar{x}$ appear in L, we can use trivial replacement $x' = x$. Now consider $L = \mathfrak{L}(\psi, C \vee u, \mathcal{S})$. For any literal l such that $l, \bar{l} \in L$ then we use replacement l to l^*. For any literal $l \in L$ such that $\bar{l} \notin L$, we use replacement l to l' where $l' = l^* \vee l$. Note that l' and l have the same dependency set.

Consider χ_0^* and weaken every l^* literal such that $l \in L$ and $\bar{l} \notin L$ to $l \vee l^*$. Now consider all clauses in $\chi_1 = \mathfrak{C}(\psi, \phi \setminus \chi_0, \mathcal{S})$ and for each clause weaken every l literal such that $l \in L$ and $\bar{l} \notin L$ to $l \vee l^*$. It remains to replace $\bar{l} \notin L$ when $l \in L$. We will have to use $\bar{l}' = \bar{l}^* \wedge \bar{l}$ to match with l'. This can be done by distributivity in Frege, if every clause $K \vee \bar{l}$ we have a copy of $K' \vee \bar{l}$ and $K' \vee \bar{l}^*$ where K' is a copy of K with every literal $l \in \mathcal{S}$ replaced by l'.

This turns out to be basically the case because K can only contain literals k such that $k \in L$ and $\bar{k} \notin L$. $k \in L$ because we can extend the path from $C \vee u$ to l through $\bar{l}$. If $\bar{k} \in L$ then $\bar{l}$ would be in L as well, against our assumption. If $K \vee \bar{l} \in \chi_0 \cap \chi_1$ we already have $K' \vee \bar{l}^*$ and $K' \vee \bar{l}$. It is possible that $K \vee \bar{l} \in \chi_0$ but not χ_1 in which case we create $K' \vee \bar{l}$ by weakening each $\mathcal{S}$ literal in $K \vee \bar{l}$ that is not $\bar{l}$.

Finally we need clause C' which is the subclause C where every $\mathcal{S}$-literal l is replaced by l'. Note that all $\mathcal{S}$-literals in C are by definition in L, so first we can derive C^* and then weaken to get C'. To do this we need the unit propagation proofs of $\phi \wedge \bar{C} \wedge \bar{O}_D \vdash_1 \bot$ for each $D \in \chi_0$ and $\bar{l} \in D$. We prove $\phi \wedge \bar{C} \wedge \bar{O}_D \to \bot$ in $O(|\phi|)$ many Frege steps. We can then derive $C \vee \bigwedge_{D \in \chi_0}^{\bar{u} \in D} O_D$ from ϕ next. Now we can replace all $\mathcal{S}$-literals l in C with $l^u \vee \bar{u}$ where $u \to (l^u \leftrightarrow l)$. This gets us $\bar{u} \vee \bigwedge_{D \in \chi_0}^{\bar{u} \in D} O_D \vee C^u$ i.e $\bar{u} \vee \bar{\mathcal{O}} \vee C^u$. We also use $\bar{u} \to (l^{\bar{u}} \leftrightarrow l)$ and $C \vee u$ to replace every $\mathcal{S}$-literal l with $l^{\bar{u}}$ and get $C^{\bar{u}} \vee u$. The u literal can be reduced, (since this rule generalises the original universal reduction this is crucial). $(u \wedge \mathcal{O}) \vee C^* \vee \bar{C}^{\bar{u}}$ is a tautology once the definition of l^* is considered which can be proven in a linear size Frege proof. Likewise $\bar{u} \vee \bar{\mathcal{O}} \vee C^* \vee \bar{C}^u$ is a tautology which can be proven in a linear size Frege proof. And by resolving disjuncts together we are left with C^*. We can get always get C' through weakening C^*.

$$\dfrac{\dfrac{C^{\bar{u}} \qquad (u \wedge \mathcal{O}) \vee C^* \vee \bar{C}^{\bar{u}}}{(u \wedge \mathcal{O}) \vee C^*} \qquad \dfrac{\bar{u} \vee \bar{\mathcal{O}} \vee C^u \qquad \bar{u} \vee \bar{\mathcal{O}} \vee C^* \vee \bar{C}^u}{\bar{u} \vee \bar{\mathcal{O}} \vee C^*}}{\dfrac{C^*}{C'}}$$

◀

▶ **Example 21.** Consider the QBF with the prefix $\exists x \forall u \exists y, z, a$ and the matrix

$$\psi = (\overbrace{u \vee z}^{C \vee u}) \wedge (\overbrace{x \vee \bar{u} \vee y \vee \bar{z}}^{C_1}) \wedge (\overbrace{\bar{y} \vee z \vee a}^{C_2}) \wedge (\overbrace{x \vee \bar{a}}^{C_3}) \wedge (\overbrace{x \vee y \vee z}^{C_4}) \wedge (\overbrace{\bar{y} \vee \bar{z}}^{C_5}) \wedge (\overbrace{\bar{x} \vee \bar{u} \vee a}^{C_6})$$

We have $\mathcal{S} = \{y, z, a\}$, $\chi_0 = \mathfrak{C}(\psi, C \vee u, \mathcal{S}) = \{C \vee u, C_1, \ldots, C_5\}$, and $L = \mathfrak{L}(\psi, C \vee u, \mathcal{S}) = \{y, \bar{y}, z, \bar{z}, a\}$. We are looking at outer clauses of those clauses in χ_0 that contain $\bar{u}$, which is only C_1, with $O_{C_1} = \{x\}$. Notice that there is also the clause C_6 with $\bar{u}$ in it, but $C_6 \notin \chi_0$ because the only possible predecessor to C_6 is C_3 and it cannot be both entered and exited via its only $\mathcal{S}$-literal $\bar{a}$. We need to verify whether $C_1 \wedge \cdots \wedge C_6 \wedge \bar{C} \wedge \bar{O}_{C_1} \vdash_1 \bot$. Setting $\bar{C} \wedge \bar{O}_{C_1} = \bar{x} \wedge \bar{z}$ propagates $\bar{a}$ from C_3, then $\bar{y}$ from C_2, and finally $\bot$ from C_4. The preconditions of Lemma 20 are therefore satisfied, and C can be soundly obtained by reduction from $C \vee u$.

Let us now perform the substitutions from the proof of Lemma 20. First, we introduce the extensions $y^{\bar{u}}, y^u, z^{\bar{u}}, z^u, a^{\bar{u}}, a^u$. Next, we define the *-variables using $\mathcal{O} = \bigvee_{D \in \chi_0}^{\bar{u} \in D} \bar{O}_D = \bar{x}$:

$$y^* = \left(\bar{u} \vee x \to y^{\bar{u}}\right) \wedge (u \wedge \bar{x} \to y^u) \qquad z^* = \left(\bar{u} \vee x \to z^{\bar{u}}\right) \wedge (u \wedge \bar{x} \to z^u)$$
$$a^* = \left(\bar{u} \vee x \to a^{\bar{u}}\right) \wedge (u \wedge \bar{x} \to a^u)$$

We have four types of clauses in χ_0 depending on which literals of u they contain. Let us start with the clauses that do not contain any u literal, namely C_2, C_3, C_4, C_5. By resolution with the extension definitions and subsequent reduction of u, we obtain

$$C_2^u = \bar{y}^u \vee z^u \vee a^u \qquad C_3^u = x \vee \bar{a}^u \qquad C_4^u = x \vee y^u \vee z^u \qquad C_5^u = \bar{y}^u \vee \bar{z}^u$$
$$C_2^{\bar{u}} = \bar{y}^{\bar{u}} \vee z^{\bar{u}} \vee a^{\bar{u}} \qquad C_3^{\bar{u}} = x \vee \bar{a}^{\bar{u}} \qquad C_4^{\bar{u}} = x \vee y^{\bar{u}} \vee z^{\bar{u}} \qquad C_5^{\bar{u}} = \bar{y}^{\bar{u}} \vee \bar{z}^{\bar{u}}$$

For the sake of brevity, we shall demonstrate the next step only on C_3. Now $\bar{u} \vee \bar{\mathcal{O}} \vee C_3^* \vee \bar{C}_3^{\bar{u}} = \bar{u} \vee x \vee a^* \vee \bar{a}^u$ can be simplified to $\bar{u} \vee x \vee a^u \vee \bar{a}^u$, which is obviously a tautology. It can be derived as follows: from the definition of a^*, we obtain $u \wedge \bar{x} \to a^* = a^u$, and thus $u \wedge \bar{x} \wedge \bar{a}^u \to \bar{a}^*$, or in other words, the clause above. Other *-replacements for the clauses C_1, C_2, C_4, C_5 are performed in a similar fashion.

For the y, z the variables y^*, z^* are the final replacements, but for the variable a we need to create the variable $a' = a^* \vee a$. We have $\chi_1 = \mathfrak{C}(\psi, \{C_6\}, \mathcal{S}) = \{C_3, C_6\}$. We take $C_3^* = x \vee \bar{a}^*$ and $C_3 = x \vee \bar{a}$ and apply distributivity to obtain $C_3' = x \vee (\bar{a}^* \wedge \bar{a})$. For other clauses that contain a positively, we simply weaken to obtain C_i'. Note that if C_3 contained, say a z literal, we would not be able to apply distributivity, however a literal like z would also cause a path from C to C_6.

Finally, we need to prove $C \vee \bar{\mathcal{O}} = x \vee z$, which we get by resolving the clauses on the unit propagation path: C_4, C_3, and C_2. This will give us $x \vee z$. By resolving with the extension definitions for z^u and $z^{\bar{u}}$, we obtain $u \vee x \vee z^{\bar{u}}$ and $\bar{u} \vee x \vee z^u$. Because z^u and $z^{\bar{u}}$ do not depend on u, we can apply universal reduction to get the clauses $C^u = z^u$ and $C^{\bar{u}} = z^{\bar{u}}$. Finally, from the definition of z^* we have $z^{\bar{u}} \to (z^* \leftrightarrow (u \wedge \bar{x} \to z^u))$, and thus $z^{\bar{u}} \wedge z^u \to z^*$, and we obtain the desired Frege proof of $C^* = \{z^*\}$.

We have demonstrated ability to replace a conjunctive normal form with another and preserve DQBF satisfiability, by replacing variables with new variables from the Independent Extension rule. Changing the formula, but preserving satisfiability is a key part of preprocessing, and this gives us a key connections to interference proof systems like DRAT, QRAT and DQRAT, whose introduction was to certify preprocessing steps.

▶ **Lemma 22.** *Both QRATU and EUR are special cases of the rule p-simulated in Lemma 20.*

Proof.

QRATU. Suppose $C \vee u$ reduces to C via QRATU. We know for every clause $D \in \phi$ with $\bar{u} \in D$, $\phi \wedge \bar{O}_D \wedge \bar{C} \vdash_1 \bot$. Therefore no matter how many of these D belong to χ_0 in Lemma 20, every one satisfies the condition.

EUR. Suppose $C \vee u$ reduces to C via EUR. Via the side condition of EUR, χ_0 does not contain any clauses $D \in \phi$ with $\bar{u} \in D$ therefore the side condition of the rule in Lemma 20. ◀

With the lemmas we have shown, we now have the p-simulation of QRAT.

▶ **Corollary 23.** IndExtFrege + ∀red *can p-simulate QRAT for false QBFs.*

Proof. When adding a clause through ATA. We simply use Lemma 18 to add the clause. Instead of deleting a clause, we simply ignore it. IndExtFrege + ∀red does not have any global conditions for any of its rules, except that a new extension rule require a new name for the variable, but this is immaterial to the proof complexity. For simulating QRATA, QRATU and EUR we create a new copy of the entire formula and then forget about clauses and variables outside of this. For QRATA we use Lemma 19 and for QRATU and EUR we use different special cases of Lemma 20 which is allowed by Lemma 22. ◀

4.3 P-simulation of the $\mathcal{D}^{\text{rrs}}$ Rule

The EUR rule uses resolution paths to calculate when it is safe to drop a single isolated literal. But in QBF and DQBF a more general approach can be taken, where resolution paths can be used to calculate that the dependency between a universal literal and a potential blocking existential literals is always spurious, in all clauses and throughout all stages of the proof.

The $\mathcal{D}^{\text{rrs}}$ relation [28] calculates (u, x) only if there is a resolution path from u to $\bar{u}$ through existential x. $\mathcal{D}^{\text{rrs}}$ is a sound way of modifying a DQBF prefix. We can add this to sound DQBF proof systems, in fact a number of QBF and DQBF proof system are already constructed in this way.

4.3.1 Definition of IR($\mathcal{D}^{\text{rrs}}$)-calc

Q($\mathcal{D}^{\text{rrs}}$)-Res [28] is a QBF proof system using the $\mathcal{D}^{\text{rrs}}$ rule, we can extend this to IR($\mathcal{D}^{\text{rrs}}$)-calc [3] which uses DQBF-IR-calc as its proof system after modifying the proof system via $\mathcal{D}^{\text{rrs}}$. The rules can be seen in Figure 7.

4.3.2 Definition of DQRAT

Another proof system that uses $\mathcal{D}^{\text{rrs}}$ is DQRAT. Here the $\mathcal{D}^{\text{rrs}}$ rule is a prefix modification rule used in conjunction with rules similar to QRAT. The rules of DQRAT are given in full in [10] and we provide a simplified overview here.
ATA: Add a clause C if $\phi \wedge \bar{C} \vdash_1 \bot$.

Axiom rule: $\mathsf{axiom}(\phi)$

$$\frac{}{\{a^{\tau|_{D_a^{\mathsf{rrs}}}} \mid a \in C,\ \mathrm{var}(a) \in \exists\}}$$

- C is a clause in the matrix ϕ.
- $\tau|_{D_a^{\mathsf{rrs}}} := \{\neg l \mid l \in C, \mathrm{var}(l) \in \forall, (\mathrm{var}(l), \mathrm{var}(a)) \in D^{\mathsf{rrs}}\}$.

Instantiation rule: $\mathsf{inst}(C, \beta)$

$$\frac{C}{\{a^{(\alpha\circ\beta)|_{D_a^{\mathsf{rrs}}}} \mid a^\alpha \in C\}}$$

- β is a partial assignment to universal variables.
- $\alpha \circ \beta$ is $\{l \mid (l \in \alpha) \vee (l \in \beta \wedge l \notin \alpha)\}$
- $(\alpha \circ \beta)(a) := \{l \in \alpha \circ \beta \mid (\mathrm{var}(l), \mathrm{var}(a)) \in D^{\mathsf{rrs}}\}$.

Resolution rule: $\mathsf{res}(C_1, C_2, x^\tau)$

$$\cdot\ \frac{C_1 \qquad C_2}{(C_1 \cup C_2) \setminus \{x^\tau, \neg x^\tau\}}$$

- $x^\tau \in C_1$ and $\neg x^\tau \in C_2$.

Figure 7 Proof rules of $\mathsf{IR}(\mathcal{D}^{\mathsf{rrs}})$-calc [3] .

DQRAT$_\exists$: Add a clause C if l is an existential literal in C and if for all $D \in \phi$ with $\bar{l} \in D$ that $\phi \wedge \bar{C} \wedge O_D \vdash_1 \bot$. Where the outer clause O_D with respect to l is the set of all universal literals u in D, that are in $D_{\mathrm{var}(l)}$, the dependency set of l and all existential literals x in D, whose dependency set is no larger than l's, i.e. $D_{\mathrm{var}(x)} \subset D_{\mathrm{var}(l)}$.
UR: Modify a clause $C \vee u$ to C if u is a universal literal and not in the dependency set of C.
DQRAT$_\forall$: Modify a clause $C \vee u$ to C, if u is a universal literal and if for all $D \in \phi$ with $\bar{u} \in D$ that $\phi \wedge \bar{C} \wedge O_D \vdash_1 \bot$. Where the outer clause O_D with respect to u is the set of all existential literals x in D, that have variables in the set S and the set of universal literals v in D that are in the dependency set of every variables in S with $\mathrm{var}(v) \neq \mathrm{var}(u)$. S is given as the set of all existential variables that depend on u.
BPM: A new variable is added to the prefix. When $\exists$, an arbitrary dependency set is used.
DRRS: The dependency set of every existential variable can be modified to remove any spurious dependencies calculated by the Reflexive Resolution Path dependency scheme ($\mathcal{D}^{\mathrm{rrs}}$).
DEL: Delete a clause.

4.3.3 P-simulations using $\mathcal{D}^{\mathsf{rrs}}$

▶ **Lemma 24.** *Suppose we have a DQBF $\forall U \exists E \psi$. We can calculate the relation $\mathcal{D}^{\mathrm{rrs}}$ on $\forall U \exists E \psi$ between universal variables and existential variables. Suppose there is a universal variable u and we let $\mathcal{S}$ define the set of these variables that depend on u, $\mathcal{S} = \{x \in X \mid u \in D_x\}$. Using* IndExtFrege + ∀red *we can derive the clauses of ψ' that replaces each variable $x \in \mathcal{S}$ such that $(u, x) \notin \mathcal{D}^{\mathrm{rrs}}$ with x' such that $D_{x'} \subset D_x \setminus u$.*

Proof. We focus on the variables $x \in \mathcal{S}$ that are declared independent to u via $\mathcal{D}^{\mathrm{rrs}}$ and we need to replace it and its negation in all clauses. If there is resolution path from u to x there is no resolution path from $\bar{u}$ to $\bar{x}$ and if there is a resolution path from $\bar{u}$ to x there is no resolution path from u to $\bar{x}$. We can write this formally, let χ_u be the subset of ψ which contains every clause with a u literal and let $\chi_{\bar{u}}$ be the subset of ψ which contains every clause with a $\bar{u}$ literal. Define $L_u = \mathfrak{L}(\psi, \chi_u, \mathcal{S})$ and $L_{\bar{u}} = \mathfrak{L}(\psi, \chi_{\bar{u}}, \mathcal{S})$, then for $(u, x) \notin \mathcal{D}^{\mathrm{rrs}}$, $(x \notin L_u \vee \bar{x} \notin L_{\bar{u}}) \wedge (x \notin L_{\bar{u}} \vee \bar{x} \notin L_u)$. This leaves the following possibilities (up to some symmetries):

1. $x \notin L_u$, $\bar{x} \notin L_u$, $x \notin L_{\bar{u}}$ and $\bar{x} \notin L_{\bar{u}}$
2. $x \in L_u$, $\bar{x} \notin L_u$, $x \notin L_{\bar{u}}$ and $\bar{x} \notin L_{\bar{u}}$
3. $x \in L_u$, $\bar{x} \notin L_u$, $x \in L_{\bar{u}}$ and $\bar{x} \notin L_{\bar{u}}$
4. $x \in L_u$, $\bar{x} \in L_u$, $x \notin L_{\bar{u}}$ and $\bar{x} \notin L_{\bar{u}}$

We also need the symmetric cases for $\bar{u}$ in case 3 using $u \to (x^u \leftrightarrow x)$. In cases 1, 2 and 3, we use both $\bar{u} \to (x^{\bar{u}} \leftrightarrow x)$ and $u \to (x^u \leftrightarrow x)$ to get clauses $u \vee \bar{x} \vee x^{\bar{u}}$ and $\bar{u} \vee \bar{x} \vee x^u$. Now resolve them to get $\bar{x} \vee x^{\bar{u}} \vee x^u$. Replace every x literal with $x^{\bar{u}} \vee x^u$ via resolution. We consider $x' = x^{\bar{u}} \vee x^u$. Now consider the presence of $\bar{x}$ in some clause $D \in \psi$. Every other $\mathcal{S}$-literal $l \in D$ cannot have $\bar{l} \in L_u$ nor $\bar{l} \in L_{\bar{u}}$, because it means $\bar{x} \in L_u$ or $\bar{x} \in L_{\bar{u}}$ which cannot happen outside of case 4. We should have a copy of D where every $\mathcal{S}$-literal l is replaced by $l^u \vee l^{\bar{u}}$ except $\bar{x}$. We duplicate this and in one copy replace $\bar{x}$ with $u \vee \bar{x}^{\bar{u}}$, the u literal can be reduced. In the other copy replace $\bar{x}$ with $\bar{u} \vee \bar{x}^u$ and here the $\bar{u}$ literal can be reduced. Now by taking a conjunction we effectively make the final S-literal replacement of this clause, which is to replace $\bar{x}$ with $\bar{x}^{\bar{u}} \wedge \bar{x}^u$.

In Case 4 use definition $\bar{u} \to (x^{\bar{u}} \leftrightarrow x)$ to replace every x literal with $u \vee x^{\bar{u}}$ and every $\bar{x}$ literal with $u \vee \bar{x}^{\bar{u}}$. We have one remaining issue. Clauses K with variables from case 4, may have an additional universal literal. Firstly we claim that both u and $\bar{u}$ literals are not present, this can only be introduced with another case 4 literal for $\bar{u}$ and this can only happen if there was a path from $\bar{u}$ to a literal k in K and a path from $\bar{u}$ to $\bar{k}$, but since $u \vee x^{\bar{u}}$, there would also be a path from u to k meaning k is not case 4. We also claim that you can always remove this universal literal u with Lemma 20, specifically the DQBF version of EUR. Suppose after the replacement there is a resolution path from K to some clause with $\bar{u}$ in it, this $\bar{u}$ comes from either a clause originally with $\bar{u}$ in it, or a clause with a case 4 variable in it of the opposite polarity. Either way, there would be a resolution path from $\bar{u}$ to literal x contradicting our assumptions in case 4. Therefore $x' = x^{\bar{u}}$ suffices. ◀

▶ **Corollary 25.** IndExtFrege + ∀red *can p-simulate IR(*$\mathcal{D}^{\text{rrs}}$*)-calc for false QBFs.*

▶ **Corollary 26.** IndExtFrege + ∀red *can p-simulate DQRAT for false DQBFs.*

5 Conclusion

We have introduced a powerful new proof rule that p-simulates most (D)QBF pre-processing and solving rules. Some proof complexity questions remain, in particular whether IndExtFrege + ∀red is reciprocally p-simulated by some of these proof systems. We have not provided any direct p-simulations that work for long-distance Q resolution and its variations in QBF proof systems. A major obstacle is that long-distance resolution in DQBF is unsound. It will still be possible to represent these rules with side conditions that force the prefix to be ordered and like a QBF. Technically a p-simulation is already known because QRAT p-simulates long-distance Q-resolution via blocked literal eliminations (using QRATU) [20]. It may be possible that other simulations work via the methods we have used. In the full preprint of this paper [15] we provide a proof that IndExtFrege + ∀red p-simulates G.

Other techniques that may have simulations include other dependency scheme rules such as the tautology free dependency scheme [5]. One difficulty in simulating this currently is that not much is yet known about the Skolem transformations which have helped us find the right definitions for replacement variables.

References

1. Valeriy Balabanov and Jie-Hong R. Jiang. Unified QBF certification and its applications. *Formal Methods in System Design*, 41(1):45–65, 2012. doi:10.1007/s10703-012-0152-6.
2. Valeriy Balabanov, Magdalena Widl, and Jie-Hong R. Jiang. QBF resolution systems and their proof complexities. In Carsten Sinz and Uwe Egly, editors, *SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2014. doi:10.1007/978-3-319-09284-3_12.
3. Olaf Beyersdorff, Joshua Blinkhorn, Leroy Chew, Renate A. Schmidt, and Martin Suda. Reinterpreting dependency schemes: Soundness meets incompleteness in DQBF. *Journal of Automated Reasoning*, 63(3):597–623, 2019. doi:10.1007/s10817-018-9482-4.
4. Olaf Beyersdorff, Joshua Blinkhorn, and Meena Mahajan. Building strategies into QBF proofs. In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019, March 13-16, 2019, Berlin, Germany*, volume 126 of *LIPIcs*, pages 14:1–14:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.STACS.2019.14.
5. Olaf Beyersdorff, Joshua Blinkhorn, and Tomáš Peitl. Strong (D)QBF dependency schemes via tautology-free resolution paths. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020*, volume 12178 of *Lecture Notes in Computer Science*, pages 394–411, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-51825-7_28.
6. Olaf Beyersdorff and Benjamin Böhm. Understanding the relative strength of QBF CDCL solvers and QBF resolution. In James R. Lee, editor, *12th Innovations in Theoretical Computer Science Conference, ITCS 2021, January 6-8, 2021, Virtual Conference*, volume 185 of *LIPIcs*, pages 12:1–12:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITCS.2021.12.
7. Olaf Beyersdorff, Ilario Bonacina, Leroy Chew, and Jan Pich. Frege systems for quantified Boolean logic. *J. ACM*, 67(2), April 2020. doi:10.1145/3381881.
8. Olaf Beyersdorff, Leroy Chew, and Mikoláš Janota. Extension variables in QBF resolution. In Adnan Darwiche, editor, *Beyond NP, Papers from the 2016 AAAI Workshop*, volume WS-16-05 of *AAAI Technical Report*. AAAI Press, 2016. URL: http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12612.
9. Olaf Beyersdorff, Leroy Chew, and Mikoláš 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.
10. Joshua Blinkhorn. Simulating DQBF preprocessing techniques with resolution asymmetric tautologies. *Electron. Colloquium Comput. Complex.*, TR20, 2020. URL: https://eccc.weizmann.ac.il/report/2020/112.
11. Sravanthi Chede and Anil Shukla. Does QRAT simulate IR-calc? QRAT simulation algorithm for ∀ exp+ res cannot be lifted to IR-calc. *arXiv preprint arXiv:2107.04547*, abs/2107.04547, 2021. doi:10.48550/arXiv.2107.04547.
12. Leroy Chew. Proof simulation via round-based strategy extraction for QBF. In Toby Walsh, Julie Shah, and Zico Kolter, editors, *AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA*, pages 11176–11184. AAAI Press, 2025. doi:10.1609/aaai.v39i11.33215.
13. 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.
14. Leroy Chew and Marijn J. H. Heule. Relating existing powerful proof systems for QBF. 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 10:1–10:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.10.

15 Leroy Chew and Tomáš Peitl. Better extension variables in dqbf via independence, 2025. doi:10.48550/arXiv.2505.20069.

16 Leroy Chew and Friedrich Slivovsky. Towards uniform certification in QBF. *Log. Methods Comput. Sci.*, 20(1), 2024. doi:10.46298/lmcs-20(1:14)2024.

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

18 Marijn Heule, Martina Seidl, and Armin Biere. A unified proof system for QBF preprocessing. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *7th International Joint Conference on Automated Reasoning (IJCAR)*, Lecture Notes in Computer Science, pages 91–106, 2014. doi:10.1007/978-3-319-08587-6_7.

19 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, *SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 201–214. Springer, 2007. doi:10.1007/978-3-540-72788-0_21.

20 Benjamin Kiesl, Marijn J. H. Heule, and Martina Seidl. A little blocked literal goes a long way. In Serge Gaspers and Toby Walsh, editors, *SAT 2017*, volume 10491 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2017. doi:10.1007/978-3-319-66263-3_18.

21 Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn JH Heule. Extended resolution simulates drat. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning: 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 516–531. Springer, Springer, 2018. doi:10.1007/978-3-319-94205-6_34.

22 Benjamin Kiesl and Martina Seidl. QRAT polynomially simulates ∀Exp+Res. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 193–202. Springer, 2019. doi:10.1007/978-3-030-24258-9_13.

23 Hans Kleine Büning, Marek Karpinski, and Andreas Flögel. Resolution for quantified Boolean formulas. *Inf. Comput.*, 117(1):12–18, 1995. doi:10.1006/inco.1995.1025.

24 Jan Krajíček and Pavel Pudlák. Quantified propositional calculi and fragments of bounded arithmetic. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 36:29–46, 1990. doi:10.1002/malq.19900360106.

25 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.

26 Markus N. Rabe. A resolution-style proof system for DQBF. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017*, pages 314–325, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-66263-3_20.

27 Franz-Xaver Reichl and Friedrich Slivovsky. Pedant: A certifying DQBF solver. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*, volume 236 of *LIPIcs*, pages 20:1–20:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.20.

28 Friedrich Slivovsky and Stefan Szeider. Variable dependencies and Q-resolution. 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 269–284. Springer, 2014. doi:10.1007/978-3-319-09284-3_21.

29 Grigorii Samuilovich Tseitin. On the complexity of proof in prepositional calculus. *Zapiski Nauchnykh Seminarov POMI*, 8:234–259, 1968.

30 Allen Van Gelder. Variable independence and resolution paths for quantified Boolean formulas. In *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876, pages 789–803. Springer, 2011. doi:10.1007/978-3-642-23786-7_59.

31 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.

SAT-Metropolis: Combining Markov Chain Monte Carlo with SAT/SMT Sampling

Maja Aaslyng Dall
IT University of Copenhagen, Denmark

Raúl Pardo
IT University of Copenhagen, Denmark

Thomas Lumley
University of Auckland, New Zealand

Andrzej Wąsowski
IT University of Copenhagen, Denmark

Abstract

Probabilistic inference via Markov Chain Monte Carlo (MCMC) is at the core of statistical analysis and has a myriad of applications. However, probabilistic inference in the presence of hard constraints, so constraints that must hold with probability one, remains a difficult task. The reason is that hard constraints make the state space of the target distribution sparse, and may even divide it into disjoint areas separated by probability-zero states. As a consequence, the random walk performed by MCMC algorithms fails to effectively sample from the complete set of states in the target distribution. In this paper, we propose the use of SAT/SMT sampling to adapt a classic and widely used MCMC algorithm, namely Metropolis sampling. We use SAT/SMT samplers as proposal distributions. In this way, the algorithm ignores probability-zero states. Our method, SAT-METROPOLIS, effectively works in problems with multivariate polynomial hard constraints where regular Metropolis fails. We evaluate the convergence and scalability of SAT-METROPOLIS using three different state-of-the-art SAT/SMT samplers: SPUR, CMSGen, and MegaSampler. The evaluation shows how different features of the SAT/SMT sampling tools affect the effectiveness of probabilistic inference. We conclude that SAT/SMT sampling is a viable and promising technology for probabilistic inference under hard constraints.

2012 ACM Subject Classification Mathematics of computing

Keywords and phrases SAT/SMT sampling, Probabilistic inference, Markov Chain Monte Carlo

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.12

Supplementary Material *Software*: `https://github.com/itu-square/sat-metropolis` [5]
archived at `swh:1:dir:bfc2169353301af051d1af4488050fcfba06a849`

Funding This work was partially funded by DIREC (Digital Research Centre Denmark), a collaboration between the eight Danish universities and the Alexandra Institute supported by the Innovation Fund Denmark.

1 Introduction

Probabilistic inference with Markov Chain Monte Carlo (MCMC) is at the core of statistical analysis with a myriad of applications in domains such as biology, medicine and physics. Probabilistic inference for problems with soft constraints has seen notable progress [14], but inference in the presence of hard constraints remains a difficult task [12]. Yet hard constraints appear in a wide range of real-world applications such as road routing, genetics or computer security; just to mention a few of them.

Probabilistic inference via MCMC is a technique to sample data or compute expected values from distributions whose explicit density functions are unknown, but we can evaluate them up to a multiplicative factor. For instance, in probabilistic Bayesian inference $P(x|y) =$

© Maja Aaslyng Dall, Raúl Pardo, Thomas Lumley, and Andrzej Wąsowski;
licensed under Creative Commons License CC-BY 4.0

$P(y|x)P(x)/P(y)$, the term $P(y|x)P(x)$ (the likelihood and the prior) can be evaluated at any point of the state space of the target distribution, but computing the normalizing factor $1/P(y)$ is infeasible in practice. MCMC algorithms tackle this problem by following a random walk on the state space guided by the term $P(y|x)P(x)$ [4].

Unfortunately, MCMC algorithms often fail in the presence of hard constraints that make the state space sparse or even partition it into disjoint parts of positive probability separated by areas of zero probability. Since MCMC algorithms reject zero probability samples, they struggle to explore such distributions completely, even though they are correct in the limit. Classic algorithms like Metropolis-Hastings [19, 11] and Gibbs sampling [9] cannot effectively reach all positive probability states under these conditions [12]. They fail to converge or must be run for a prohibitively long time.

Existing work addressing this problem focuses on the design of sampling algorithms for specific types of probabilistic inference problems [12, 13, 22]. For instance, Hazelton and coauthors propose a method to sample from a $\mathbb{Z}$-polytope defined by inverse linear counting problems – i.e., problems of the form $\mathbf{y} = \mathbf{A}\mathbf{x}$ where $\mathbf{y}$ is a vector of counts, $\mathbf{A}$ is a contingency matrix and $\mathbf{x}$ the vector of variables of interest [12]. Although efficient, their method is limited to problems of this particular form. Poon and Domingos pioneered using SAT samplers in probabilistic inference. They propose an inference method for problems with propositional constraints that encode relational properties [22]. They create a slice sampler using SAT sampling to perform probabilistic inference over Markov Logic Networks [23]. Unfortunately, this requires encoding probabilistic models as Markov Logic Networks, a model not used commonly in statistics and limited when continuous variables are needed.

In this work, we extend a classic MCMC method with SAT/SMT sampling to support probabilistic inference under mixed soft and hard constraints. In particular, we use SAT/SMT samplers as proposal distributions for the Metropolis algorithm [19]. Our method works on standard probabilistic models widely used in statistics. The theory of discrete integer linear arithmetic with multiplication allows us to handle multivariate polynomial hard constraints. We evaluate it using multiple modern SAT/SMT samplers that provide an excellent basis to "jump" between the set of states with positive probability and avoid probability-zero regions of the state space by design. Finally, we discuss what properties of SAT/SMT samplers help to effectively address probabilistic inference. In summary, our contributions are:

- A MCMC algorithm for probabilistic inference problems under mixed soft and hard constraints, supporting multivariate polynomial hard constraints,
- An implementation of the algorithm using three different SAT/SMT samplers,
- An evaluation of convergence of the inference for different encodings and samplers,
- An evaluation of the scalability of the algorithm in realistic case studies.

2 SAT/SMT sampling

Let T be a theory over a set of variables with some concrete domain, logical symbols, predicate symbols, function symbols and a fixed interpretation. Let φ denote a formula in T and $\mathbf{x}$ be a vector with the variables appearing in φ. A satisfying assignment is an assignment of domain values to the variables in $\mathbf{x}$ such that φ evaluates to true. We use $\{\mathbf{x} \mid \mathbf{x} \models \varphi\}$ to denote the set of the satisfying assignments of φ.

A Satisfiability Modulo Theory (SMT) sampler is an algorithm that generates samples from the set of solutions of a formula φ, denoted $\mathbf{x}_i \xleftarrow{\$} \mathcal{D}(\{\mathbf{x} \mid \mathbf{x} \models \varphi\})$, according to a probability distribution $\mathcal{D}$. When φ is a Boolean formula, we call it a SAT sampler. A SAT/SMT sampler is uniform if the probability of obtaining any satisfiable assignment is

$1/|\{\mathbf{x} \mid \mathbf{x} \models \varphi\}|$, denoted $\mathcal{U}(\{\mathbf{x} \mid \mathbf{x} \models \varphi\})$. A SAT/SMT sampler is *almost* uniform if the probability of sampling any satisfiable assignments is at least $1 - \epsilon/|\{\mathbf{x} \mid \mathbf{x} \models \varphi\}|$ and at most $1 + \epsilon/|\{\mathbf{x} \mid \mathbf{x} \models \varphi\}|$ for some $\epsilon > 0$.

In this paper, we use the theory T_{MIA} of integer linear arithmetic extended with multiplication to handle multivariate polynomial constraints. A T_{MIA} formula includes logical connectives $\{\neg, \wedge\}$ (and derived), predicates $\{>, =\}$ (and derived) and arithmetic operations $\{+, -, *\}$, but no division. The interpretation of the symbols is standard. Variables have finite integer domains. We use bit-blasting to encode formulae in Boolean logics for SAT-samplers.

3 Metropolis Algorithm for Bayesian Probabilistic Inference

The Metropolis algorithm samples from a *target* distribution $P(\mathbf{y}) = P^*(\mathbf{y})/Z$ by using evaluations of $P^*(\mathbf{y})$, a function different by an unknown constant factor $1/Z$. The vector $\mathbf{y}$ denotes the list of parameters for the target distribution. When applied to Bayesian inference, the target distribution is the posterior defined by the Bayes rule $P(\mathbf{x} \mid D) = P(D \mid \mathbf{x})P(\mathbf{x})/Z$. The term $P^*(\mathbf{y})$ is instantiated as $P(D \mid \mathbf{x})P(\mathbf{x})$, i.e., the *likelihood* times the *prior*. When no ambiguity arises, we write $P(\cdot)$ to denote a probability measure on a measurable space that is compatible with its parameters.

The likelihood function $P(D \mid \mathbf{x})$ is used to model soft and hard constraints in the underlying probabilistic model. Let $D = \{c_1, \ldots, c_n\}$ be a set of events that model constraint parameters, e.g., observations from real-world data or logical formulae. The set D is commonly known as a *set of observations*. The likelihood is defined as $P(D \mid \mathbf{x}) = \prod_i P(c_i \mid \mathbf{x})$. A hard constraint is a constraint with 0–1 likelihood. For example, for a model with parameter x, we can impose $x > 0$, i.e., $c_i = \{x \mid x > 0\}$, by defining $P(x > 0 \mid \mathbf{x}) = \mathbf{1}_{x>0}(x)$ where $\mathbf{1}_{x>0}(x)$ is the indicator function; its value is 1 if $x > 0$ and 0 otherwise. Note that the posterior probability of values of x that do not satisfy the constraint is zero, $P(x < 0 \mid \mathbf{x}) = 0$. Soft constraints, on the other hand, allow for specifying noisy properties in the model. For example, consider that value $c_i = v$ has been collected using a recording device with a known measurement error of $\sigma_v \in \mathbb{R}_{>0}$. It is possible to model the measurement error using a soft constraint $P(v \mid x) = \mathcal{N}(v; \mu = x, \sigma = \sigma_v)$. In this example, we use a Gaussian distribution ($\mathcal{N}$) to model that values of x close to v have high likelihood, and other values, though less likely, are also possible; due to the measurement error of the recording device.

To sample values from the posterior, the Metropolis algorithm takes a random walk over the parameter space starting in a random point $\mathbf{x}$ with a positive posterior probability, $P(\mathbf{x} \mid D) > 0$. Given a state $\mathbf{x}_i$, a *proposal distribution*, $q(\mathbf{x}^* | \mathbf{x})$, is used to sample a candidate next state in the random walk. The Metropolis algorithm requires $q(\cdot)$ to be symmetric, i.e., $q(\mathbf{x}^* | \mathbf{x}) = q(\mathbf{x} | \mathbf{x}^*)$. An extension of the algorithm, Metropolis-Hastings, relaxes this constraint [11, 16]. The probabilistic inference presented in Section 5 does not require asymmetric proposal distributions. Given a state $\mathbf{x}_i$ and proposal $\mathbf{x}^*$, the next state $\mathbf{x}_{i+1}$ is selected according to the following *acceptance ratio*:

$$r(\mathbf{x}, \mathbf{x}^*) = \frac{P(\mathbf{x}^* \mid D)}{P(\mathbf{x} \mid D)} = \frac{P(D \mid \mathbf{x}^*)P(\mathbf{x}^*)/Z}{P(D \mid \mathbf{x})P(\mathbf{x})/Z} = \frac{P(D \mid \mathbf{x}^*)P(\mathbf{x}^*)}{P(D \mid \mathbf{x})P(\mathbf{x})} \tag{1}$$

In particular, $\mathbf{x}_{i+1}$ is selected as

$$\mathbf{x}_{i+1} = \begin{cases} \mathbf{x}^* & \text{with probability } \min(1, r(\mathbf{x}, \mathbf{x}^*)) \\ \mathbf{x} & \text{otherwise.} \end{cases} \tag{2}$$

That is, the proposed state is accepted with probability $\min(1, r(\mathbf{x}, \mathbf{x}^*))$. Otherwise, the proposal is rejected and the random walk stays in the current state.

If the proposal distribution is positive, i.e., $q(\mathbf{x}^* \mid \mathbf{x}) > 0$ for all $\mathbf{x}, \mathbf{x}^*$, the probability distribution induced by a collection of n samples $\{\mathbf{x}_i\}_{i=1..n}$ is known to converge to $P(\mathbf{x}|D)$ as $n \to \infty$ [11, 16]. The theoretical guarantees aside, the choice of proposal distribution drastically affects the speed of convergence in practice. Much of the work on MCMC addresses designing efficient (in terms of convergence speed) proposal distributions such as Gibbs sampling or Hamiltonian Monte Carlo (HMC) [4], predominantly for continuous variables. Despite these advances, it remains a challenge to sample from posterior distributions with hard discrete constraints.

4 The Problem with Hard Constraints in Markov Chain Monte Carlo

The popular MCMC algorithms, such as Metropolis, Metropolis-Hastings, Gibbs sampling and HMC, are not suitable for sampling from distributions with hard constraints. The problem is that the proposal distribution is unaware of the hard constraints and, consequently, proposes states violating them. Such proposals are promptly rejected – for any $\mathbf{x}^*$ violating hard constraints, we have $P(\mathbf{x}^* \mid D) = 0$, the acceptance ratio $r(\mathbf{x}, \mathbf{x}^*) = 0/P(D|\mathbf{x})P(\mathbf{x}) = 0$, and the probability of accepting $\mathbf{x}^*$ is $\min(1, 0) = 0$. Thus, the random walk gets stuck in its current state ($\mathbf{x}$).

▶ **Example 1.** Consider a simple reconstruction problem (cf. Section 6). Two integers (x_1, x_2) in the closed interval $[0, 10]$ are known to sum to eleven. The task is to determine the most likely values of x_1 and x_2 given the sum. We pose this as a Bayesian inference problem.

$$
\begin{aligned}
x_1 &\sim \text{DiscUni}(0, 10) \\
x_2 &\sim \text{DiscUni}(0, 10) \\
y &:= x_1 + x_2 \\
y &= 11
\end{aligned}
$$

The first two lines model the prior belief regarding valuations of x_1 and x_2. As we use a discrete uniform distribution in the interval $[0, 10]$, all the values from 0 to 10 are equally likely *a priori*. The third line models the sum of the variables. The last line states the hard constraint on the sum. The posterior $P(x_1, x_2 \mid y = 11)$ gives the answer to the reconstruction problem.

To use Metropolis to estimate this posterior, we need a proposal distribution. A common choice is a standard normal distribution $\mathcal{N}(0, 1)$. This is, e.g., the default proposal distribution in the PyMC library [1]. For discrete random variables, the proposal values from the normal distribution are discretized to the closest integer. Thus, given a state for x_i, the next state is computed as $[x_i + \nu]$ where $\nu_i \sim \mathcal{N}(0, 1)$ and $[\cdot]$ denotes rounding to the closest integer. For a state (x_1, x_2) we write $q(x_1^*, x_2^*|x_1, x_2)$ for the probability of proposing state (x_1^*, x_2^*).

To appreciate the detrimental effect of the hard constraints on Metropolis, consider the probability of transitioning to a state satisfying $y = 11$ from $x_1 = 1, x_2 = 10$. The most likely proposal is to remain in the current state, as the mode of the normal distribution equals its mean. Considering the rounding to integers, the probability of sampling the current state is $q((x_1, x_2) \mid (x_1, x_2)) \approx 0.14$. The next most likely states, those within L1 distance of one from the current state $||(x_1', x_2') - (x_1, x_2)||_1 = 1$, are proposed with $q((x_1', x_2')|(x_1, x_2)) \approx 0.09$. New states within distance of two are proposed with probability < 0.06. In general, new states satisfying $y = 11$ distinct from the current state $(1, 10)$ *at any distance* are proposed with probability just above 0.06. Consequently, the probability of staying in the current state is Q is $\approx 1 - 0.06 = 0.94$. It is approximately 16 times more likely to remain in the

current state than to move to a new state satisfying the hard constraint. In practice, this leads to very low sampling speed and slow convergence; as the algorithm spends most time being stuck in the same state. ⌟

This problem gets much worse for sparse constraints when the state space increases. It is intractable to randomly sample a satisfying solution to a hard constraint. As we show in Section 6, problems do not have to get very complex for the sampler to get stuck completely.

5 SAT/SMT-Powered Metropolis

The key idea to address the above problems is to use SAT/SMT sampling as the proposal distribution. We present, SAT-METROPOLIS, our modification of the Metropolis sampler as Algorithm 1 below. The algorithm takes four arguments: (i) a possibly not normalized probability density P to approximate, (ii) a set of hard constraints Φ, (iii) an initial state for the random walk, and (iv) the number of samples n to generate. As common for the sampling algorithms, the initial state must have positive probability $P(\mathbf{x}_{\text{init}}) > 0$, which in this case means that it must also satisfy the hard constraints. The output is a set of samples $\{x_i\}_{i=1}^{n}$. When used for Bayesian inference, P is instantiated as likelihood times prior, i.e., the unnormalized posterior distribution (cf. Section 3).

We begin in the provided initial state (line 2). The main novelty is found in line 4. The proposal distribution, $\mathcal{U}(\{\mathbf{x} \mid \mathbf{x} \models \Phi\})$ is a uniform distribution over all states that satisfy the hard constraints. A uniform proposal distribution over non-zero probability states reduces autocorrelation among samples, which helps reducing bias and improving speed of convergence [17]. Furthermore, it is compatible with the inner workings of existing SAT/SMT samplers, which work by producing batches of uniform samples. However, the proposal distribution in Metropolis can be non-uniform and conditional on a state. This type of proposal distributions can also achieve low autocorrelation, if they are properly configured. Since SAT/SMT sampling from conditional and non-uniform is not supported by existing tools, we do not consider it in this version of SAT-METROPOLIS. The remaining steps in Algorithm 1 proceed as usual in Metropolis (cf. Section 3). Lines 5 and 6 compute the acceptance probability. Lines 7 to 11 select the new proposal $\mathbf{x}^*$ with probability α. Otherwise we remain in the current state. The procedure is repeated n times.

The proposal distribution in SAT-METROPOLIS is symmetric and positive; as it is independent of the current state and it assigns non-zero probability to any pair of states. Therefore, it trivially follows that SAT-METROPOLIS converges in the limit (cf. Section 3). However, the practical use of this algorithm will depend on how efficiently and uniformly we can sample from the proposal distribution.

We implemented SAT-METROPOLIS in Python, using three engines for sampling solutions to hard constraints: SPUR [2], CMSGen [10], and MegaSampler [21]. We selected these engines as they vary in: the uniformity of generated samples, speed of sampling and amount of pre-processing. These features are the main factors that affect the speed of convergence and scalability of our algorithm (cf. section 6). Furthermore, these engines are at the forefront of the research in SAT/SMT sampling. The implementation is publicly available at `https://github.com/itu-square/sat-metropolis`.

Figure 1 gives a high level overview of the implementation. The implementation takes the target distribution as a Python function from the domain of the parameter space ($\mathbf{x} \in \mathcal{X}^m$) to reals $\mathbb{R}$. This distribution specifies the soft constraints. To specify hard constraints, the user should provide an SMT problem in abstract syntax constructed using the Z3 API [6].

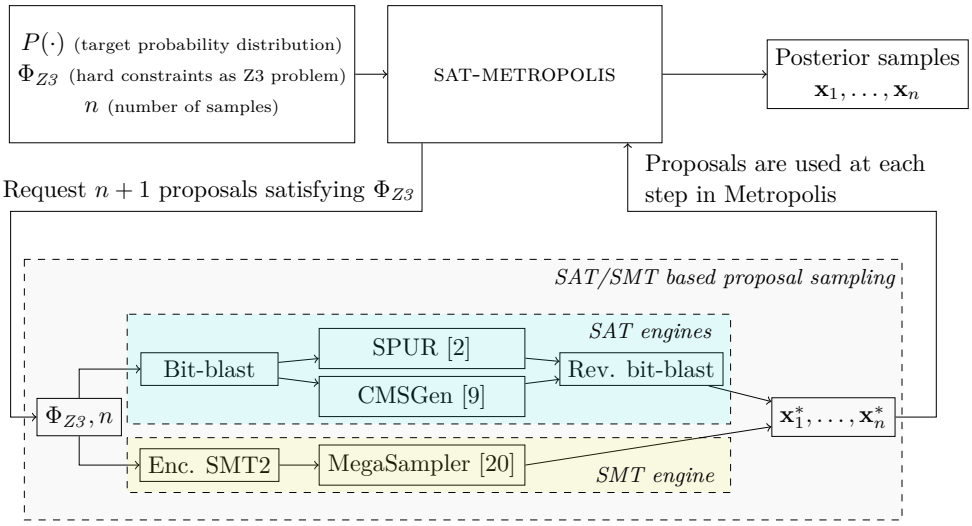


Figure 1 An overview of the SAT-METROPOLIS implementation.

The implementation starts by requesting $n+1$ proposal samples from an SMT/SAT sampling engine. For simplicity, we use the first sample as the initial state $\mathbf{x}_{\text{init}}$. This assumes that the target distribution has positive density for all states satisfying the hard constraints, which is the case for the problems we consider. Then, it proceeds as shown in Algorithm 1 with the only difference that the sampling step on line 4 retrieves a sample from the received proposals. This is equivalent to the behavior specified by Algorithm 1, as proposals are sampled independently of the current state. Depending on the type of engine used for generating samples, there are requirements on the input Z3 problem and pre-processing steps. We discuss them below. However all the engines used allow us to handle problems with polynomial hard constraints that was not possible for previous methods for probabilistic inference.

Algorithm 1 Metropolis algorithm with SAT/SMT uniform sampling as proposal distribution.

Input: Target probability density P (not normalized)
Set of hard constraints Φ
Initial state $\mathbf{x}_{\text{init}}$ such that $P(\mathbf{x}_{\text{init}}) > 0$
Number of samples n
Output: Set of samples $\mathbf{x}_1, \ldots, \mathbf{x}_n$

1: **procedure** SAT-METROPOLIS($P, \Phi, \mathbf{x}_{\text{init}}, n$)
2: $\quad\mathbf{x}_1 \leftarrow \mathbf{x}_{\text{init}}$
3: $\quad$**for** $i \leftarrow 1$ to n **do**
4: $\quad\quad\mathbf{x}^* \xleftarrow{\$} \mathcal{U}(\{\mathbf{x} \mid \mathbf{x} \models \Phi\})$
5: $\quad\quad r \leftarrow P(\mathbf{x}^*)/P(\mathbf{x}_i)$
6: $\quad\quad\alpha \leftarrow \min(1, r)$
7: $\quad\quad u \xleftarrow{\$} \mathcal{U}(0, 1)$
8: $\quad\quad$**if** $u \leq \alpha$ **then**
9: $\quad\quad\quad\mathbf{x}_{i+1} \leftarrow \mathbf{x}^*$
10: $\quad\quad$**else**
11: $\quad\quad\quad\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i$
12: $\quad\quad$**end if**
13: $\quad$**end for**
14: **end procedure**

SMT-sampling engine. For MegaSampler, the Z3 problem is specified in T_{MIA} over integers (`Int` in Z3). This allows for modeling discrete random variables with infinite support. We convert the problem into the SMT2 format using the Z3 built-in facility and feed it into MegaSampler, requesting to generate n samples. As before, we parse the output file with the samples and convert it into a format that can be used by SAT-METROPOLIS.

SAT-sampling engines. To use the SAT engines SPUR or CMSGen, we require that the constraint is specified in T_{MIA} (Section 2) using bit-vector variables. The user provides the bit-lengths for the variables and we use Z3 to translate the constraints into the Boolean space to obtain a CNF problem [15] and eventually DIMACS. We combine three Z3 tactics to carry out this process: `'simplify'`, `'bit-blast'`, and `'tseitin-cnf'`. The first tactic, `'simplify'`, produces a equisatisfiable formula with the goal of making it easier to tackle by the solver. The second tactic, `'bit-blast'`, performs standard bit-blasting. The last tactic, `'tseitin-cnf'`, performs a Tseiting's transformation that results in a CNF formula. We use our own script to turn the output of Z3 into DIMACS format. Then, we instruct SPUR or CMSGen to generate n samples. The output files with the samples are parsed, reverse bit-blasted, and converted into a format that our SAT-METROPOLIS algorithm can use.

Due to the bit-vector encoding, SAT engines can only handle discrete distributions with finite support. However, this is does not affect the results and conclusions of our evaluation (cf. Section 6), as we consider problems with finite support and bit-vectors of sufficient length.

6 Evaluation

We evaluate SAT-METROPOLIS by answering the following research questions.

RQ1 How does the SAT/SMT sampling method affect the convergence of SAT-METROPOLIS?

RQ2 How big problems can be handled by SAT-METROPOLIS with the contemporary SAT/SMT samplers?

Table 1 lists the probabilistic inference problems used in the evaluation. To answer **RQ1**, we use two database reconstruction problems, DB CACM and NZ Stats. DB CACM was previously introduced as an example of database reconstruction using SMT solvers [8]. NZ Stats uses data published by Statistics New Zealand. We use the data and the information on the anonymization algorithms used in these cases, to reason about beliefs for the possible reconstructions. These problems are simple enough to solve analytically, and we use the analytical solution as the ground truth for evaluation of convergence. Note that evaluating the convergence of the estimation on large problems without ground truth is not possible.

To answer **RQ2**, we use the last two problems in the table (Books and Roads). Books and Roads are examples of problems with realistic sizes regarding the number of variables and constraints and domain sizes. They had been used to evaluate specialized methods to solve probabilistic inference under linear constraints [12] (Section 7). Determining how SAT-METROPOLIS performs on them is a good indicator on whether modern SAT/SMT-sampling can handle realistic tasks. We consider multiple versions of these two problems by systematically adjusting their size; with the objective of finding the limits of scalability.

In general, we consider problems that range from 10 to 72 variables and 2 to 141 constraints. The formulae include polynomial and linear constraints (cf. Table 1). The structure of the formulae is problem dependent and we describe it the sections below. The SAT version of these problems ranges from 148 to 10252 variables and from 499 to 47299 clauses. Clauses contain from 2 to 4 literals and the clause-to-var ratio ranges from 3.06 to 5.08. Table 4 in Appendix A contains the complete list of problems that we consider in this paper.

Table 1 Inference problems used for evaluation.

Problem	#vars	#constraints	Hard constraints type
Database reconstruction (DB CACM)	10	20	Polynomial
Database reconstruction (NZ Stats)	38	141	Linear
Highway traffic flow (Roads)	28	7	Linear
Book ratings (Books)	72	42	Linear

The experiments were run on Ubuntu 24.04 with an Intel Core i7-1355U CPU and 16GB RAM. The experiment package is available at `https://github.com/itu-square/sat-metropolis`.

6.1 Convergence

We evaluate the convergence of SAT-METROPOLIS using the SAT/SMT sampling engines: SPUR, CMSGen and MegaSampler. The goal of the evaluation is to determine how the different features of these engines affect convergence. SPUR ensures uniformity by design. CMSGen does not guarantee uniformity, but it has been shown empirically to produce near uniform samples. Technically, MegaSampler is not a sampler in the statistical sense, but a solution enumerator. It never produces the same state twice. We use it to evaluate the strategy of reweighing the set of unique samples using the unormalized target distribution. To evaluate convergence, we use two database reconstruction problems, DB CACM and NZ Stats. The former includes polynomial constraints and the latter only linear constraints. This allows us to observe the effect of different types of constraints on the algorithm.

We compute the mean absolute difference between the ground truth and the SAT-METROPOLIS estimation of probability for each outcome for each variable. Let X be the set of variables in a problem and $x \in X$ be a problem variable with domain $\mathrm{dom}(x)$. Let $P(x)$ the target distribution and $\hat{P}(x)$ its approximation by SAT-METROPOLIS. The absolute mean difference is defined as:

$$\mu_{\mathrm{diff}}(X) = |X|^{-1} \sum_{x \in X} |\mathrm{dom}(x)|^{-1} \sum_{v \in \mathrm{dom}(x)} |P(x = v) - \hat{P}(x = v)| \tag{3}$$

where $|\cdot|$ is overloaded to mean cardinality for sets and absolute value for reals. When μ_{diff} tends to 0 as the number of samples increases, the SAT-METROPOLIS converges to the right histogram. When μ_{diff} does not approach zero, we conclude no convergence or a biased convergence (convergence to a wrong value). We generate 20000 samples for each experiment. We executed every experiment 3 times and report the mean.

DB CACM. A database contains records of five individuals with ages $(x_0, \ldots, x_4)$ and whether they identify as male $(m_0, \ldots, m_4)$. Ages x_i are integers in the interval $[0, 125]$, and variables m_i are Booleans. A data analyst releases the following information about the database: the median age is 30, the mean age is 38, the number of males is 3, and the mean age of males is 44. The database reconstruction problem is to determine the values of variables x_i and m_i that are consistent with the released data. This problem can be addressed using probabilistic inference [20]. We assign prior distributions to variables x_i and m_i modeling attacker knowledge before observing the published data. Then, using probabilistic inference, we compute the posterior distribution given the released data encoded as hard constraints, which models the attacker knowledge after observing the released data. We model an

Table 2 A simplified database over South Head, NZ demographics.

Age range	0-4	5-9	10-14	15-19	20-24	25-29	30-34	35-39	40-44	45-49	50-54	55-59	60-64	65-69	70-74	75-79	80-84	85-89	90+
Male	3	6	6	12	9	3	9	9	6	6	15	6	3	6	3	3	0	0	0
Female	6	0	3	6	3	6	3	9	3	6	9	6	9	3	3	3	0	0	0
Total	15	3	12	15	15	12	9	18	9	12	21	12	12	12	6	0	0	0	0

attacker who considers the all ages equally possible for all individuals, and also assumes that individuals are male with probability 0.5. Thus, we use uniform distributions over all possible values as prior distributions, i.e., $x_i \sim \text{DiscUni}(0, 125)$ and $m_i \sim \text{DiscUni}(0, 1)$, where $0, 1$ denote false and true respectively.

$$0 \leq x_0 \leq x_1 \leq x_2 = 30 \leq x_3 \leq x_4 \leq 125 \qquad \textit{Age ranges and median} \qquad (4)$$

$$\textstyle\sum_i x_i = 38 \cdot 5 \qquad \textit{Mean age of all individuals} \qquad (5)$$

$$\textstyle\sum_i m_i = 3 \qquad \textit{Num. individuals who identify as male} \qquad (6)$$

$$\textstyle\sum_i m_i \cdot x_i = 44 \cdot 3 \qquad \textit{Mean age of 3 males} \qquad (7)$$

We use SAT-METROPOLIS to generate samples from $P(x_0, \ldots, x_4, m_0, \ldots, m_4 | (4), (5), (6), (7))$.

NZ Stats. This is a case study similar to the one above, but based on data from the statistics agency in New Zealand, NZ Stats. Table 2 shows a set of anonymized counts of the number of males and females in the South Head area in New Zealand. All values in the table are multiples of three, as NZ Stats uses a method known as "round to base 3" to anonymize counts. Given a count value v, if v is a multiple of three, release v. Otherwise consider the multiples of three below and above v and return the closer one with probability p and the further one with probability $1 - p$.

Let x_i be the real counts and y_i be the corresponding anonymized values shown in Table 2. We model an attacker that assumes all values from 0 to 31 to be equally likely for all x_i, so $x_i \sim \text{DiscUni}(0, 31)$. This range is chosen to ensure a sufficient number of bits to represent the largest value in Table 2 (i.e., 23). To model the anonymization method, we impose the constraint that $y_i - 2 \leq x_i \leq y_i + 2$, i.e., each y_i value must have been computed from a value x_i that was at most two units away. If $y_i = 0$, we impose $x_i \leq y_i + 2$, as negative counts are not possible. For the total counts (last row in Table 2), the real count is the sum $x_i = x_i^{\text{m}} + x_i^{\text{f}}$ of real counts of males and females. The total counts are also anonymized.

Results. Figures 2 and 3 summarize the convergence results. SAT-METROPOLIS with SPUR converges to the correct values in both case studies: μ_{diff} is below 0.011 for DB CACM and below 0.008 for NZ Stats after 5000 samples. After 15000+ samples, it falls below 0.004 for both problems. The convergence for the linear problem (NZ Stats) is only minimally faster than for the polynomial problem (DB CACM). These are excellent results, showing that SAT-METROPOLIS with SPUR solves these problems effectively.

SAT-METROPOLIS with CMSGen converges, but with an observable bias. The value of $\mu_{\text{diff}}(\{x_i\})$ for marginals $x_0 \,..\, x_4$ and x_9 is below 0.014 for DB CACM after 5000 samples. However, for the binary variables $x_5 \,..\, x_8$ the error increases; notably $\mu_{\text{diff}}(\{x_5\}) \approx 0.19$. The value of μ_{diff} is below 0.076 for all variables in the NZ Stats case study after 5000 samples. The plots show that μ_{diff} does not notably reduce as the number of samples increases, which indicates that results are biased. The bias is rather small for integer variables, but

problematic for binary variables. The results are acceptable, but, an order of magnitude away from the results with the SPUR backend, suggesting that handling variables with binary domains might be a limitation of the CMSGen backend in this application.

SAT-METROPOLIS with MegaSampler converges as well, but, like with CMSGen, it exhibits bias for select variables. The value of $\mu_{\text{diff}}(\{x_i\})$ for variables $x_0 .. x_4$ and x_9 is below 0.022 in DB CACM after 5000+ samples. However, for binary variables $x_5 .. x_8$ the value increases up to 0.23. For NZ Stats and 5000+ samples, the value of μ_{diff} is between 0.06 and 0.15 for most variables. We do observe a larger absolute mean difference for integer variables in the NZ Stats problem. At the same time the MegaSampler backend shows the slowest speed of convergence, with values of μ_{diff} above 0.3 for some variables in all problems. Increasing the number of samples above ≈ 5000 does not reduce the error significantly, which suggests bias. These results also indicate that the reweighing method that we used for MegaSampler is not effective in removing bias.

Discussion. The results of the convergence experiments indicate that uniformity helps un-biased convergence, and in this regard SPUR performs very well. This might be because a uniform proposal distribution, is not only positive, but also symmetric. It would be interesting to try more robust versions of the Metropolis algorithms, to see whether they would be able to overcome the non-uniformity of CMSGen and MegaSampler.

The type of constraints did not significantly impact the speed of convergence or bias. It is also noteworthy that all backends converged to their lowest absolute mean difference value within 5000 samples. This indicates that the samples are indeed obtained in a (practically) independent fashion. Generating un-correlated samples is one of the key factors in the efficiency of MCMC inference algorithms. If proposals are correlated, convergence is slow.

We executed an additional experiment that combines hard and soft constraints. We evaluated the convergence of SAT-METROPOLIS for the DB CACM and NZ Stats problems using a Poisson distribution for count variables and bias Bernoulli for binary variables. We assigned to each variable x_i a distribution Poisson(λ) and each variable m_i a distribution Bernoulli(p). The Poisson distribution assigns the highest posterior probability to the value λ and probability decreases as values move away from λ. Poisson is a common distribution for count variables, as it has a positive support – i.e., it assigns probability 0 to negative values. The Bernoulli distribution assigns a probability of p to m_i equals true and $1-p$ to the opposite. The hard constraints are the same as in the case studies, but this distributions add soft constraints showing higher preference for values close to λ for count variables and bias p for binary variables. Our results indicated no significant changes in convergence behavior. SPUR converges to the right value, but with slower speed of convergence. CMSGen and Metropolis are bias, but μ_{diff} decreases with respect to the results without soft constraints. These changes are due to the regularizing effect of the soft constraints. This shows that SAT-METROPOLIS can handle a mix of soft and hard constraints well, even though the Metropolis part does not know the hard constraints, and the SAT/SMT sampler does not know about the soft constraints.

6.2 Scalability

We evaluate the runtime performance of SAT-METROPOLIS for increasing number of constraints and domain size for variables. The goal is to determine whether the performance of SAT/SMT samplers is sufficient to generate samples from realistic problems. We have selected two probabilistic inference problems that were used as benchmarks by Hazelton et al. [12]. Their work is one of most recent ones on MCMC algorithms for probabilistic inference with (linear) hard constraints, and, as such, these benchmarks demonstrate whether SAT-METROPOLIS using

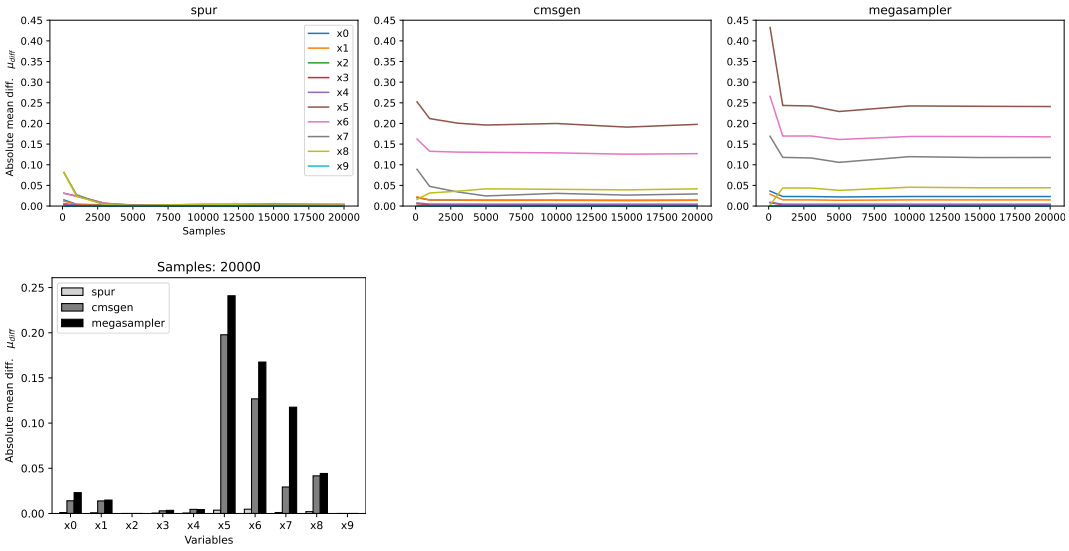


Figure 2 Convergence with different SAT/SMT samplers in the DB CACM case study. Variables $x_0 .. x_4$ represent age and $x_5 .. x_9$ are the binary variables $m_0 .. m_4$ representing gender.

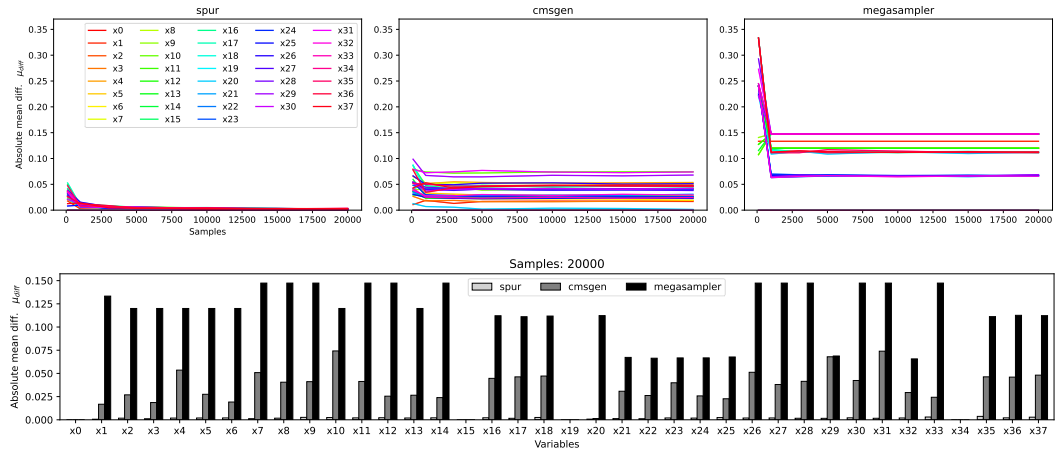
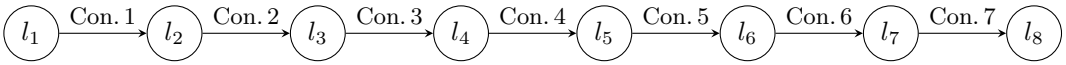


Figure 3 Convergence with different SAT/SMT samplers in the NZ stats case study. Variables $x_0 .. x_{18}$ represents the male counts from (0-4) to (90+), and variables $x_{19} .. x_{37}$ represents the female counts from (0-4) to (90+).

state-of-the-art SAT/SMT samplers is capable of tackling realistic problems. Furthermore, one can easily control the number of constraints, number of variables, and domain size for each variable for their problems, which allows us to find the limits of each sampling backend when they cannot sample from the problem as originally defined.

We perform experiments for an increasing number of samples $\{100, 500, 1000, 3000, 7000\}$, with the range matching the convergence speed in the experiments of Section 6.1. We set a timeout of ten minutes for each sampling run. This timeout is set to be twice the time the method of Hazelton and colleagues takes for the hardest cases. Note, that we expect SAT-METROPOLIS to be slower, as it is solves a more general class of problems than their method, and admits a general constrained-based formulation of these problems. We executed every experiment 3 times and report the mean.

Roads. We are estimating the traffic flow over a highway in UK. Nodes l_i represent locations in the highway. The directed edges are road segments connecting locations (connections).



To travel from l_i to l_j one must pass through all intermediate connections $i..j-1$. The traffic flows in the direction of the edges, so it is only possible to travel from l_i to l_j if $i < j$.

The task is to estimate the flow of traffic for each connection defined as a pair of nodes based on measurement of flow at each connection between adjacent nodes. Hazelton et al. use a probabilistic model for this [12]. For instance, trips between (l_1, l_2), (l_1, l_3), (l_1, l_4), ..., (l_7, l_8). For each possible trip (l_i, l_j) the probabilistic model contains a variable x_{ij} that models the number of trips from i to j. The hard constraints are given by the observed flow v_c on each segment between adjacent nodes; flow is measured in terms of vehicle counts. Given a connection c, let $T_c = \{x_{ij} \mid i \leq c < j\}$ be the set of trip variables that pass through c. Then, hard constraints are defined as $v_c = \sum_{x \in T_c} x$. The set of observed vehicle counts is $\{1087, 1008, 1068, 1204, 1158, 1151, 1143\}$, corresponding to $\{v_1, \ldots, v_7\}$. We assign a prior DiscUni$(0, m)$ for each variable x_{ij}, where m is a positive integer modeling the maximum number of trips. We set $m = 2^n - 1$ where n is the lowest integer such that hard constraints are satisfiable. For SAT backends, we set the bit-vector size to $s = \max(\lceil \log_2(v_c) \rceil) + 1$ to avoid overflow errors. Note that $s > n$ must hold, as the result of $\sum_{x \in T_c} x$ must not overflow (this constraint ensures that we do not get accidental solutions due to overflow).

The full model contains 28 variables (x_{ij}) and 7 connections. To adjust the complexity of the problem we will apply two modifications. First, we consider problems from two to seven connections. Second, we reduce the vehicle counts for each connection by a *reduction factor* r, i.e., for a reduction factor r the vehicle counts for a connection is updated as v_c/r and rounded to the closest integer. Note that this reduction also decreases domain size and bit-vector size of all variables x_{ij}. We consider reduction factors of 1, 2 and 4.

Books. We are estimating the number ratings from dataset of book rating counts with three dimensions: six different books (labeled A–F), four countries (Australia, Canada, UK and USA) and three ranges of age (0-25, 26-50 and +51). The data is shown in Table 3. The task is to estimate book ratings for all book/country/age combinations by only observing two-way interactions (marginals), i.e., number of book ratings: for all books given country and age, for all countries given age and book, and for all ages given country and book. More precisely, the set of two-way interaction counts is

$$
\begin{aligned}
B = \{&c_{\text{Aus},0-25}, c_{\text{Aus},26-50}, c_{\text{Aus},51+}, c_{\text{Can},0-25} \ldots, c_{\text{USA},51+}\} \cup && \textit{(Country,Age)} \\
&\{c_{0-25,\text{A}}, c_{26-50,\text{A}}, \ldots, c_{51+,\text{F}}\} \cup && \textit{(Age,Book)} \\
&\{c_{\text{Aus},\text{A}}, c_{\text{Can},\text{A}}, \ldots, c_{\text{USA},\text{F}}\} && \textit{(Country,Book)}
\end{aligned}
$$

where, for instance $c_{\text{Aus},0-25} = 3$ is computed by summing the counts of the first column in Table 3. The probabilistic model for this problem consists of variables of the form x_{bca} where b, c, a are indexes for book labels, countries and ages, respectively. Thus, hard constraints are defined as: $c_{ca} = \sum_{i \in \{A\,..\,F\}} x_{ica}$ (Country, Age), $c_{ab} = \sum_{i \in \{\text{Aus}\,..\,\text{USA}\}} x_{bia}$ (Age, Book), and $c_{cb} = \sum_{i \in \{0-25\,..\,51+\}} x_{bci}$ (Country, Book). Similar to the previous problem, each variable x_{bca} has a prior DiscUni$(0, m)$ where $m = 2^n - 1$, n is the lowest integer such that hard constraints are satisfiable, and, for SAT backends, bit-vectors have size $\max(\lceil \log_2(c_{ij}) \rceil) + 1$.

The model has 72 variables. We control its complexity by increasing the number of two-way interactions. We start with (Country, Age), add (Age, Book), and finally include all two-way interactions. We also reduce the book rating counts to c_{ij}/r by a reduction factor r, to adapt the domain size of the variables and thus the bit-vectors. We use factors 1, 2 and 4.

Table 3 The dataset for book rating counts per title (A–F), Country (Australia, Canada, U.K. and U.S.A) and ages (0-25, 26-50 and +51). Source: Hazelton et al. [12]

	Ages 25 and under				Ages 26 to 50				Ages 51 and over			
Book	Aus.	Can.	UK	USA	Aus.	Can.	UK	USA	Aus.	Can.	UK	USA
A	0	0	1	39	1	3	4	94	0	0	1	12
B	1	7	1	49	7	25	4	217	0	4	0	34
C	0	2	0	16	1	10	1	92	0	2	1	37
D	0	2	0	17	1	8	12	126	0	0	1	24
E	1	8	0	43	2	29	3	186	0	6	1	28
F	1	2	0	20	0	11	0	107	1	4	1	28

Results. Figures 4 and 5 summarize the average runtime of SAT-METROPOLIS for the different SAT/SMT samplers. SAT-METROPOLIS with SPUR performs successfully in eight out of 35 benchmarks. It is able to successfully complete, so generate up to 7000 samples without timing out, the Roads experiments for up to three connections – with the exception of three connections and no reduction ($r = 1$). Increasing the reduction for three connections reduces the running time notably. For Books, the SPUR variant can only solve the case with all constraints and the reduction by four. SPUR scales only to problems with low number of variables or variables with small domains, but it is noteworthy that when it does not timeout, the runtime is essentially independent of the number of samples.

The CMSGen variant is by far the most successful one, handling 18 out of 35 benchmarks. SAT-METROPOLIS with CMSGen scales up to five connections for Roads. For Books it can handle all cases with more than 27 constraints. There are two exceptions: i) the roads problem with 5 connections, reductions by 1 and 2 with 1000+ samples; and ii) the books problem with 27 constraints and no reduction. Moreover, when it does not timeout, it always outperforms the SPUR and MegaSampler backends. As expected, for all successful experiments, increasing the reduction always results in a decrease in running time. The running time seems to increase linearly with the number of samples to generate; although the increase is steeper for lower reduction factors. The CMSGen backend exhibits a similar scalability tendency to that of SPUR, but CMSGen can handle larger number of variables, constraints and domain sizes. Notably, SAT-METROPOLIS with CMSGen can solve Books as originally posed by Hazelton et al. [12].

SAT-METROPOLIS with MegaSampler performs successfully in seven out of 35 benchmarks. It solves the roads problem for five to seven connections and the books problem for all amounts of constraints – with the exception of generating 7000 samples for 27 and 42 constraints. Unfortunately, it only works for reductions by four, which for these benchmarks means a domain size of 64. MegaSampler-based SAT-METROPOLIS is adequate only for problems with small domain size and large number of constraints.

Discussion. We did consider using a vanilla Metropolis algorithm, as a baseline for the experiments above, using the implementation of the widely used probabilistic programming library PyMC [1]. However, this algorithm is unable to generate more than one sample in any of the benchmark above. It fails them all. Furthermore, to work with it, we had to use Z3 to provide a satisfying initial state, as otherwise the PyMC implementation of Metropolis cannot start the execution. In this respect, SAT-METROPOLIS is a major improvement compared to using MCMC with hard constraints, as anticipated in Section 4.

It is remarkable that SAT-METROPOLIS with CMSGen is able to solve Books as originally defined by Hazelton et al. [12], bias issues notwithstanding. Their method, known as Dynamic Lattice Basis (DLB), takes about ten seconds to produce 7000 effective samples. The number

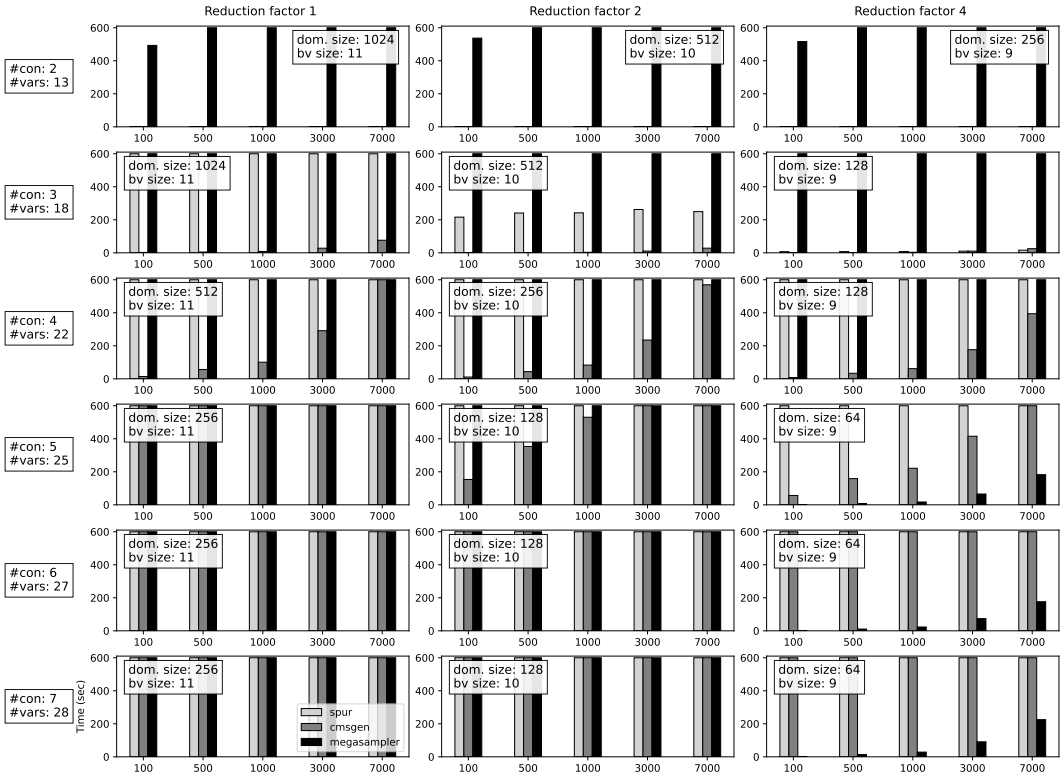


Figure 4 Running times for Roads with increasing number of connections (top to bottom), and decreasing variable domains (left to right). The original problem [12] is in the bottom-left corner.

of effective samples measures the number of samples with low autocorrelation, and it is a common measure of to compare MCMC methods of different nature [17]. This means DLB is 28x faster than SAT-METROPOLIS. However, DLB was specifically designed to solve problems with linear hard constraints in counting problems, like the case studies above, whereas SAT-METROPOLIS uses generic SAT sampling technology and can handle higher-order polynomial constraints. Also DLB requires presenting the problem as a linear program of a specific kind, while SAT-METROPOLIS does not have this limitation.

Our measurements split the running time of the SAT/SMT engine, and the rest of the algorithm, i.e., computing the acceptance ratio and deciding whether to move to the proposed state. We observed an average runtime for the latter of 0.01 and 0.05 seconds for Roads and Books, respectively. This indicates that the runtime is heavily dominated by the SAT/SMT sampler, and the effect soft constraints does significantly impact the overall runtime.

7 Related work

Recent work on MCMC based probabilistic inference with hard constraints targets discrete inverse linear problems [12, 18, 7, 3]. These are problems of the form $\mathbf{y} = \mathbf{A}\mathbf{x}$ where $\mathbf{y}$ is an observed counts vector and $\mathbf{A}$ is a contingency matrix that defines the constraints on the variables in the vector $\mathbf{x}$ w.r.t. the observed counts. A major advance in tackling this type of problems was the use of Markov (sub-)basis (see [3] for an overview). Let $\mathcal{F}_{\mathbf{y}} = \{\mathbf{x}|\mathbf{y} = \mathbf{A}\mathbf{x}\}$ be the set of count vectors satisfying the hard constraints – this set is known as the $\mathbf{y}$-*fiber*.

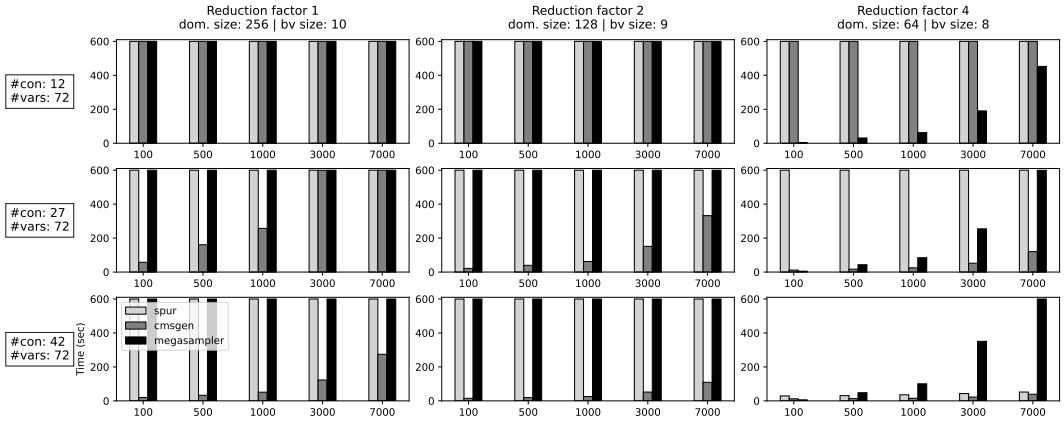


Figure 5 Running times for Books with increasing number of constraints (top to bottom) and decreasing variable domains (left to right). The original problem [12] is in the bottom-left corner.

Markov basis methods produce a Markov chain such that all points within the $\mathbf{y}$-*fiber* are reachable from any state – recall that this is a requirement for convergence of MCMC. However, computing the Markov basis for a given problem can be computationally expensive. This has motivated works suppressing the need for computing the full Markov basis. Dobra [7] designed a method to generate sampling directions dynamically while ensuring that the union of all directions is within the $\mathbf{y}$-*fiber*. More recently, the work by Hazelton et al. [12] introduces a method based on dynamic lattice basis which allows for sampling from the $\mathbf{y}$-*fiber* without computing the full Markov basis, which notably improves sampling performance. Our method does not achieve the same level of performance as these specialized methods. However, SAT-METROPOLIS can tackle a larger class of problems. Our method can handle problems with polynomial hard constraints whereas discrete inverse linear problems only feature linear constraints. Nevertheless, despite being able to tackle a more general class of problems, our evaluation showed that SAT-METROPOLIS with CMSGen can solve the book ratings problem of Hazelton.

We are not the first to propose the use of SAT samplers for probabilistic inference. Poon and Domingos [22] define a slice sampler to tackle probabilistic inference problems specified as Markov Logic Networks [23]. Markov Logic Networks are useful to express statistical problems with relational constraints, but they require uncommon encodings for other problems such as linear or polynomial constraints. Instead of designing a sampler for a specific type of probabilistic models, namely Markov Logic Networks, we adapt Metropolis, which a standard and widely used algorithm in statistics. SAT-METROPOLIS takes probabilistic inference problems expressed as an unnormalized density function (typically the numerator in Bayes rule: prior $\times$ likelihood), which is common place in the statistics literature. The slice sampler uses SampleSAT [24] to sample solutions for the relational constraints (expressed as propositional formulae). SampleSAT does not have uniformity guarantees and works on SAT problems. As a consequence, it cannot be used to analyze the effect of uniformity or bit-blasting, as we did in our work; with the backends SPUR (perfect uniformity) or MegaSampler (which works directly on SMT problems and can be compared to SPUR/CMSGen that require bit-blasting). Furthermore, the most scalable backend of SAT-METROPOLIS (cf. Section 6.2), CMSGen, has been recently shown to be one of the most scalable SAT sampling tools [10]. Thus, our work provides a more detailed and updated perspective on the use of SAT/SMT sampling methods for probabilistic inference than Poon and Domingos [22].

8 Conclusion

We have demonstrated that SAT/SMT sampling technology can effectively be used to tackle probabilistic inference problems with multivariate polynomial hard constraints. SAT/SMT sampling presents itself as a promising technology for tackling the problem of probabilistic inference via MCMC, where hard and soft constraints are mixed. To this end, we have adapted the Metropolis algorithm to use SAT/SMT samplers as proposal distributions. This prevents proposing zero-probability states, which lead the standard Metropolis algorithm to fail to explore the target distribution well. We have implemented SAT-METROPOLIS with three SAT/SMT samplers; SPUR, CMSGen and MegaSampler, and have evaluated convergence and scalability for these three variants. The results indicate that uniformity impacts convergence of SAT-METROPOLIS positively; as SPUR is the backend that exhibits the best convergence. At the same time, CMSGen is the most scalable sampler in this application. Even though SAT-METROPOLIS is generic with respect to the constraint class supported, it can handle problems used to benchmark specialized methods.

References

1 Oriol Abril-Pla, Virgile Andreani, Colin Carroll, Larry Dong, Christopher J. Fonnesbeck, Maxim Kochurov, Ravin Kumar, Junpeng Lao, Christian C. Luhmann, Osvaldo A. Martin, Michael Osthege, Ricardo Vieira, Thomas Wiecki, and Robert Zinkov. PyMC: a modern, and comprehensive probabilistic programming framework in Python. *PeerJ Comput. Sci.*, 9, 2023. doi:10.7717/peerj-cs.1516.

2 Dimitris Achlioptas, Zayd Hammoudeh, and Panos Theodoropoulos. Fast sampling of perfectly uniform satisfying assignments. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT'18)*, volume 10929 of *LNCS*, pages 135–147. Springer, 2018. doi:10.1007/978-3-319-94144-8_9.

3 Satoshi Aoki, Hisayuki Hara, and Akimichi Takemura. *Markov bases in algebraic statistics*, volume 199. Springer Science & Business Media, 2012.

4 Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of markov chain monte carlo*. CRC press, 2011.

5 Maja Aaslyng Dall, Raúl Pardo, Thomas Lumley, and Andrzej Wąsowski. SAT-Metropolis. Software, swhId: swh:1:dir:bfc2169353301af051d1af4488050fcfba06a849 (visited on 2025-07-28). URL: https://github.com/itu-square/sat-metropolis, doi:10.4230/artifacts.24207.

6 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

7 Adrian Dobra. Dynamic markov bases. *Journal of Computational and Graphical Statistics*, 21(2):496–517, 2012.

8 Simson Garfinkel, John Abowd, and Christian Martindale. Understanding database reconstruction attacks on public data: These attacks on statistical databases are no longer a theoretical danger. *Commun. ACM*, 62:46–53, 2019. doi:10.1145/3287287.

9 Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–741, 1984. doi:10.1109/TPAMI.1984.4767596.

10 Priyanka Golia, Mate Soos, Sourav Chakraborty, and Kuldeep S. Meel. Designing samplers is easy: The boon of testers. In *Proceedings of Formal Methods in Computer Aided Design, (FMCAD'21)*, pages 222–230. IEEE, 2021. doi:10.34727/2021/isbn.978-3-85448-046-4_31.

11 W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

12 Martin Hazelton, M Mcveagh, and Bruce van Brunt. Geometrically Aware Dynamic Markov Bases for Statistical Linear Inverse Problems. *Biometrika*, 108:609–626, October 2020.

13 Martin Hazelton, Michael McVeagh, Christopher Tuffley, and Bruce van Brunt. Some rapidly mixing hit-and-run samplers for latent counts in linear inverse problems. *Bernoulli*, 30(4):2676–2699, 2024.

14 Matthew D Hoffman, Andrew Gelman, et al. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623, 2014. doi:10.5555/2627435.2638586.

15 Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008. doi:10.1007/978-3-540-74105-3.

16 David J.C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

17 R. McElreath. *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. A Chapman & Hall book. CRC Press, 2020.

18 Michael McVeagh. *Efficient Markov bases for Z-polytope sampling: a thesis presented in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Mathematics at Massey University, Manawatū, New Zealand*. PhD thesis, Massey University, 2021.

19 Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

20 Raúl Pardo, Willard Rafnsson, Christian W. Probst, and Andrzej Wąsowski. Privug: Using probabilistic programming for quantifying leakage in privacy risk analysis. In *Proceedings of European Symposium on Research in Computer Security (ESORICS'21)*, volume 12973 of *LNCS*, pages 417–438. Springer, 2021. doi:10.1007/978-3-030-88428-4_21.

21 Matan I. Peled, Bat-Chen Rothenberg, and Shachar Itzhaky. SMT sampling via model-guided approximation. In *Proceedings of International Symposium on Formal Methods (FM'23)*, volume 14000 of *LNCS*, pages 74–91. Springer, 2023. doi:10.1007/978-3-031-27481-7_6.

22 Hoifung Poon and Pedro Domingos. Sound and efficient inference with probabilistic and deterministic dependencies. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI'06)*, volume 1, pages 458–463. AAAI Press, 2006. URL: http://www.aaai.org/Library/AAAI/2006/aaai06-073.php.

23 Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine learning*, 62:107–136, 2006. doi:10.1007/s10994-006-5833-1.

24 Wei Wei, Jordan Erenrich, and Bart Selman. Towards efficient sampling: Exploiting random walk strategies. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI'04)*, pages 670–676. AAAI Press / The MIT Press, 2004. URL: http://www.aaai.org/Library/AAAI/2004/aaai04-106.php.

A Experiments Details Summary

Table 4 show the details of each of the experiments that we considered in the paper. In total there are 37 benchmarks. We include the number of variables and clauses (both in the original SMT problem and after bit-blasting), the bit-vector size and domain size for each experiment. Additionally, we include the clause-to-variable ratio of the bit-blasted version of each experiment. The last three columns indicate whether the corresponding backend was able to generate at least 7000 within 10 minutes (✔) or not (✘).

Table 4 Qualitative summary of experiments results. The symbol (✔) denotes that running SAT-METROPOLIS with the backend indicated in the column was able to generate 7000 samples or more within the 10min timeout; which we consider a success. If not, we mark the experiment with the symbol (✘).

Problem	#vars	#constraints	#vars (bit-blasted)	#clauses (bit-blasted)	reduction	bv size	dom. size	clause-to-var ratio	spur	cmsgen	megasampler
DB CACM	10	20	1605	7507	NA	8	126	4.67	✔	✔	✔
NZ Stats	38	141	661	2025	NA	5	24	3.06	✔	✔	✔
Roads	13	2	203	713	1	11	1023	3.51	✔	✔	✘
Roads	13	2	183	641	2	10	511	3.50	✔	✔	✘
Roads	13	2	148	499	3	9	255	3.37	✔	✔	✘
Roads	13	2	162	565	4	9	255	3.48	✔	✔	✘
Roads	18	3	646	2722	1	11	1023	4.21	✘	✔	✘
Roads	18	3	586	2464	2	10	511	4.20	✔	✔	✘
Roads	18	3	512	2154	3	9	255	4.20	✔	✔	✘
Roads	18	3	516	2168	4	9	127	4.20	✔	✔	✘
Roads	22	4	1409	6491	1	11	511	4.60	✘	✘	✘
Roads	22	4	1267	5827	2	10	255	4.59	✘	✔	✘
Roads	22	4	1112	5118	3	9	127	4.60	✘	✔	✘
Roads	22	4	1128	5180	4	9	127	4.59	✘	✔	✘
Roads	25	5	2537	12279	1	11	255	4.83	✘	✘	✘
Roads	25	5	2288	11055	2	10	127	4.83	✘	✘	✘
Roads	25	5	2029	9776	3	9	127	4.81	✘	✘	✘
Roads	25	5	1998	9655	4	9	63	4.83	✘	✘	✔
Roads	27	6	4146	20654	1	11	255	4.98	✘	✘	✘
Roads	27	6	3742	18605	2	10	127	4.97	✘	✘	✘
Roads	27	6	3239	16122	3	9	127	4.97	✘	✘	✘
Roads	27	6	3298	16383	4	9	63	4.96	✘	✘	✔
Roads	28	7	6266	31854	1	11	255	5.08	✘	✘	✘
Roads	28	7	5683	28798	2	10	127	5.06	✘	✘	✘
Roads	28	7	4966	25201	3	9	63	5.07	✘	✘	✔
Roads	28	7	4962	25174	4	9	63	5.07	✘	✘	✔
Books	72	12	5895	24629	1	10	255	4.17	✘	✘	✘
Books	72	12	5229	21830	2	9	127	4.17	✘	✘	✘
Books	72	12	5237	21955	3	9	63	4.19	✘	✔	✔
Books	72	12	4565	19045	4	8	63	4.17	✘	✘	✔
Books	72	27	8622	38995	1	10	255	4.52	✘	✘	✘
Books	72	27	7632	34482	2	9	127	4.51	✘	✔	✘
Books	72	27	7657	34671	3	9	63	4.52	✘	✔	✘
Books	72	27	6627	29941	4	8	63	4.51	✘	✔	✔
Books	72	42	10252	47299	1	10	255	4.61	✘	✔	✘
Books	72	42	8987	41502	2	9	127	4.61	✘	✔	✘
Books	72	42	7811	36101	4	8	63	4.62	✔	✔	✘
								Success/timeout experiment counts:	10/37	20/37	9/37

Random Local Access for Sampling k-SAT Solutions

Dingding Dong
Harvard University, Cambridge, MA, USA

Nitya Mani
Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

We present a *sublinear time* algorithm that gives *random local access* to the uniform distribution over satisfying assignments to an arbitrary k-SAT formula Φ, at exponential clause density. Our algorithm provides *memory-less* query access to variable assignments, such that the output variable assignments consistently emulate a single global satisfying assignment whose law is close to the uniform distribution over satisfying assignments to Φ. Random local access and related models have been studied for a wide variety of natural Gibbs distributions and random graphical processes. Here, we establish feasibility of random local access models for one of the most canonical such sample spaces, the set of satisfying assignments to a k-SAT formula.

Our algorithm proceeds by leveraging the local uniformity of the uniform distribution over satisfying assignments to Φ. We randomly partition the variables into two subsets, so that each clause has sufficiently many variables from each set to preserve local uniformity. We then sample some variables by simulating a systematic scan Glauber dynamics backward in time, greedily constructing the necessary intermediate steps. We sample the other variables by first conducting a search for a polylogarithmic-sized local component, which we iteratively grow to identify a small subformula from which we can efficiently sample using the appropriate marginal distribution. This two-pronged approach enables us to sample individual variable assignments without constructing a full solution.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms

Keywords and phrases sublinear time algorithms, random generation, k-SAT, local computation

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.13

Related Version *Full Version*: `https://arxiv.org/abs/2409.03951`

Funding NM was supported by the Hertz Graduate Fellowship and the NSF Graduate Research Fellowship Program #2141064. This material includes work supported by the NSF Grant DMS-1928930, while the authors were in residence at SLMath during the Spring 2025 semester.

Acknowledgements We would like to thank Kuikui Liu and Ronitt Rubinfeld for helpful discussions, suggestions, and feedback on a draft. We would like to thank the anonymous referees for identifying an issue with a previous version of this article.

1 Introduction

Efficiently sampling from an exponentially sized solution space is a fundamental problem in computation. The quintessential such problem is sampling a uniformly random satisfying assignment to a given k-SAT formula, the setting of this work. Throughout, we let Φ denote an n-variable (k, d)-*formula*, which is a k-SAT formula in *conjunctive normal form* (which we also call a k-CNF) such that each variable appears in at most d clauses. We let $\Omega = \Omega_\Phi$ denote the set of satisfying assignments to Φ, and let $\mu = \mu_\Phi$ denote the uniform distribution over Ω. We consider the regime where k is a (large) constant and take $n \to \infty$.


© Dingding Dong and Nitya Mani;
licensed under Creative Commons License CC-BY 4.0



For $d \leq (2ek)^{-1}2^k$, it has long been known (e.g., by the Lovász local lemma introduced in [8]) that any such k-SAT formula Φ has at least one satisfying assignment. Moreover, seminal work of Moser and Tardos [27] gave a simple algorithm that computes such a satisfying assignment in expected linear time. However, a similarly basic question in the sampling regime is far more poorly understood: when is it possible to efficiently sample an (approximately uniformly) random satisfying assignment to Φ?

The first breakthrough in this direction came from Moitra [25] who gave a deterministic, fixed-parameter tractable algorithm for approximately sampling from μ provided that $d \lesssim 2^{ck}$ for $c \approx 1/60$. This was followed up by work of Feng, Guo, Yin and Zhang [10], who used a Markov chain approach to give a polynomial time algorithm for sampling from μ provided that $d \lesssim 2^{k/20}$. Recent works [18, 31] have given state of the art results for sampling from k-CNFs, when $d \lessapprox 2^{k/4.82}$ (in fact for large domain sizes, [31] gives essentially tight bounds on sampling from atomic constraint satisfaction problems). There has also been a tremendous amount of work sampling from random k-CNFs, recently essentially resolved by work of [6], building on previous progress of [13, 7, 12, 19].

A natural question about such algorithms to sample from k-CNFs is whether one can adapt them to more efficiently answer *local* questions about a random satisfying assignment in *sublinear time*. For variable v, let μ_v denote the marginal distribution on v induced by μ. When n is large, one might wish to sample a small handful of variables S from their marginal distributions μ_S in $o(n)$ time, without computing an entire $\Omega(n)$-sized satisfying assignment $\sigma \sim \mu$. Further, many traditional algorithms for counting the number of satisfying assignments to a given k-SAT formula proceed by computing marginal probabilities of variable assignments, a task that can be completed given local access to a random assignment. Therefore, sublinear time algorithms for answering local questions can also yield speedups in more general counting algorithms.

Ideally, a *random local access* model should provide query access to variable assignments such that the output enjoys the following properties:

- **(a)** the model is *consistent*: queries for different variable assignments consistently emulate those of a single random satisfying assignment $\sigma \sim \mu$;
- **(b)** the model is *sublinear*: sampling variable assignments takes $o(n)$ time in expectation;
- **(c)** the model is *memory-less*: given the same initialization of randomness, answers to multiple, possibly independent queries for variable assignments are consistent with each other.

We give a more precise definition of random local access models in Section 2. Random local access models were formally introduced in work of Biswas, Rubinfeld, and Yodpinyanee [5], motivated by a long line of work over the past decades studying sublinear time algorithms for problems in theoretical computer science. The authors of [5] studied a different natural Gibbs distribution, the uniform distribution over proper q-colorings of a given n-vertex graph with maximum degree Δ. By adapting previously studied classical and parallel sampling algorithms for q-colorings, they were able to construct a local access algorithm to a random proper q-coloring when $q \geq 9\Delta$. The related problem of sampling partial information about huge random objects was pioneered much earlier in [15, 16]; further work in [28] considers the implementation of different random graph models. Random local access and related (e.g., parallel computation) models have also been recently studied for several other random graphical processes and Gibbs distributions (cf. [5, 4, 26, 23]).

The above model for local access to random samples extends a rich line of work studying *local computation algorithms* (LCAs), originally introduced in works of Rubinfeld, Tamir, Vardi, and Xie [29] and Alon, Rubinfeld, Vardi, and Xie [2]. Local computation algorithms

are widely used in parallel and distributed computing, with notable practical success in areas such as graph sparsification [1], load balancing [24], and sublinear-time coloring [2]. A primary initial application of LCAs in [29] was locally constructing a satisfying assignment to a given (k, d)-formula Φ when $d \lesssim 2^{k/10}$. Similar to the desired properties of random local access, LCA implements query access to a large solution space in sublinear time using local probes. However, rather than sampling from a given distribution, LCA only provides local access to *some* consistent valid solution in the desired solution space that may otherwise be completely arbitrary in nature.

In this work, we present a polylogarithmic time algorithm that gives random local access to the uniform distribution over satisfying assignments of an arbitrary k-SAT formula Φ at exponential clause density (i.e., the number of occurrences of each variable is bounded by an exponential function of k). The following is an informal version of our main result; a more formal version can be found at Theorem 24.

▶ **Theorem 1** (Main theorem: informal)**.** *There exists a random local access algorithm $\mathcal{A}$ that satisfies the following conditions. For any (k, d)-formula $\Phi = (V, \mathcal{C})$ with maximum degree $d \lesssim 2^{k/400}$ and any variable $v \in V$, we can use $\mathcal{A}$ to sample a single assignment for v, $\mathcal{A}(v) \in \{0, 1\}$ in expected polylogarithmic time (in $|V|$), such that the distribution of $\mathcal{A}(v)$ is the same as the marginal distribution of $\sigma(v)$, where σ is a uniformly random satisfying assignment to Φ.*

Similar to [5], the proof of Theorem 1 adapts some of the algorithmic tools used to study parallel and distributed sampling. The proof also builds upon the work of Moitra [25] and Feng–Guo–Yin–Zhang [10] on sampling from bounded degree k-CNFs in polynomial time. The authors of [25, 10] both critically took advantage of a global variable *marking* (see Definition 10) to better condition the marginal distributions of variables. Such an approach allows for a subset of the variable assignments to be sampled with ease; the resulting *shattering* of the solution space conditioned on such a partial assignment then allows one to efficiently complete the random satisfying assignment. These initial approaches have been extended and strengthened to nearly linear time algorithms that succeed for larger maximum variable degree in a flurry of recent work (c.f. [25, 10, 20, 17, 13, 7, 12, 31, 6]).

Recently, Feng, Guo, Wang, Wang, and Yin [9] used a recursive sampling scheme to simulate the systematic scan projected Glauber dynamics via a strategy termed *coupling towards the past*, which they used to derandomise several Markov chain Monte Carlo (MCMC) algorithms for sampling from CSPs. Additionally, recent work of He, Wang and Yin [18] used a *recursive sampling* scheme to sample k-SAT solutions. Their work immediately gives a sublinear (in fact, expected constant time) algorithm for sampling the assignment of a single variable one time; however, this work does not immediately extend out of the box to a random local access model that enjoys the desired *consistency* and *memory-less* properties if multiple variables are sampled. Recursive sampling schemes have also been recently used to make progress on analyzing and designing fast sampling algorithms for a variety of Gibbs distributions (cf. [18, 7, 3]). As noted earlier, such schemes have been particularly useful for *derandomising* and constructing *parallel* and *distributed* versions of many popular MCMC algorithms for sampling solutions to CSPs [9, 11, 17, 18].

An immediate roadblock to adapting such global or parallel sampling strategies to random local access to k-SAT solutions is that the vast majority of the aforementioned existing algorithms critically either use some *global* information – like a variable marking, particular ordering of variables, or other global state compression – as an input to make algorithmic decisions or postpone sampling of certain problematic variable assignments until a *linear* number of other choices are made. Both these issues necessitate a substantive

departure from these approaches for any hope of local access adaptation. In this work, we overcome these obstacles by adapting the *coupling towards the past* strategy used in [9] to derandomise MCMC in conjunction with a local implementation of the variable marking strategy introduced in [25].

We use these algorithms to carefully select and sample a small number of auxiliary variable assignments on an as-needed basis, showing that bad cases can be reduced to small calculations after a sublinear number of steps. Our proof of Theorem 1 requires a novel adaptation of sampling techniques to avoid requiring global information or complete passes of the variable set; we show that we can perform analogous operations locally or leverage local uniformity properties to circumvent them altogether in the course of locally sampling a variable assignment. Importantly, we demonstrate how to execute local sampling in an *oblivious, consistent* fashion, so that the algorithm need not retain any memory and that variables need not be sampled in any particular order.

2 Preliminaries

Notation

Throughout, let $\Phi = (V, \mathcal{C})$ be a k-CNF on variable set $V = \{v_1, \dots, v_n\}$ with associated collection of clauses $\mathcal{C}$. In this work we do not let the same variable appear multiple times in any clause of Φ, although our algorithm could be adapted to the more general scenario. We further assume that Φ is a (k, d)-*formula*; that is, each variable v_i appears in at most d clauses. For every clause $C \in \mathcal{C}$, let $\mathrm{vbl}(C)$ denote the collection of variables in the clause C. We further define $\Delta := \max_{C \in \mathcal{C}} |\{C' \in \mathcal{C} : \mathrm{vbl}(C) \cap \mathrm{vbl}(C') \neq \emptyset\}|$, so in particular $\Delta \leq kd$.

In the regime we work in, we assume k is a large fixed constant and view $n \to \infty$. We use $f \lesssim g$ to denote that there is some constant C (not depending on k) such that $f \leq Cg$. We also use the standard asymptotic notation (O, o, Ω, ω, Θ), where when not specified, we assume these are in the $n \to \infty$ limit. We use $\mathsf{Law}(X)$ to denote the underlying distribution of a random variable X.

We let $\Omega = \Omega_\Phi \subseteq \{0,1\}^n$ denote the set of satisfying assignments to Φ and let $\mu = \mu_\Phi$ denote the uniform distribution over Ω. We suppress the Φ subscripts when the formula is clear from context. We introduce a few more concepts associated to a k-SAT instance that will be used later.

▶ **Definition 2.** *Given probability distributions ν_1, ν_2 over Ω, the total variation distance is*

$$d_{\mathsf{TV}}(\nu_1, \nu_2) := \frac{1}{2} \sum_{\omega \in \Omega} |\nu_1(\omega) - \nu_2(\omega)|.$$

If we have a random variable X with $\mathsf{Law}(X) = \nu$, we may write $d_{\mathsf{TV}}(\mu, X)$ instead of $d_{\mathsf{TV}}(\mu, \nu)$ in a slight abuse of notation.

▶ **Definition 3** (Dependency hypergraph)**.** *Given a (k, d)-formula Φ, let $H_\Phi = (V, \mathcal{E})$ be the dependency hypergraph (with multiple edges allowed), where V is the set of variables and $\mathcal{E} = \{\mathrm{vbl}(C) : C \in \mathcal{C}\}$ is the collection of variable sets of clauses of Φ.*

▶ **Definition 4** (Partial assignment)**.** *Given (k, d)-formula $\Phi = (V, \mathcal{C})$, let*

$$\mathcal{Q}^* := \bigcup_{\Lambda \subseteq V} \{0, 1, \perp\}^\Lambda$$

denote the space of partial assignments with the following symbology. Given a partial assignment $\sigma \in \{0, 1, \perp\}^\Lambda$ on some $\Lambda \subseteq V$, each variable $v \in \Lambda$ is classified as follows:

- $\sigma(v) \in \{0,1\}$ *means that* v *is accessed by the algorithm and assigned with* $\sigma(v) \in \{0,1\}$*;*
- $\sigma(v) = \perp$ *means that* v *is accessed by the algorithm but unassigned yet with any value.*

We sometimes use $\emptyset$ *to denote the empty assignment (i.e.,* Λ *is the empty set). We say that* σ *is feasible if it can be extended to a satisfying assignment to* Φ*.*

▶ **Definition 5** (Reduced formula on partial assignment)**.** *Let* Φ *be a* (k,d)*-formula. Given a partial assignment* σ *on* $\Lambda \subseteq V$*, let* Φ^σ *be the result of simplifying* Φ *under* σ*. That is, define* $\Phi^\sigma := (V^\sigma, \mathcal{C}^\sigma)$ *where*

- $V^\sigma = V \setminus \{v \in \Lambda : \sigma(v) \in \{0,1\}\}$,
- $\mathcal{C}^\sigma$ *is obtained from* $\mathcal{C}$ *by removing all clauses that have been satisfied under* σ*, and removing any appearance of variables that are assigned 0 or 1 by* σ*.*

Let H^σ_Φ *be the associated (not necessarily* k*-uniform) hypergraph to* Φ^σ*. For variable* $v \in V \setminus \Lambda$*, let* Φ^σ_v *denote the maximal connected component of* Φ^σ *to which* v *belongs.*

▶ **Definition 6** (Marginal distribution)**.** *For an arbitrary set of variables* $S \subseteq V$*, let* μ_S *be the marginal distribution on* S *induced by* μ*, so that*

$$\mu_S(\tau) = \sum_{\tau' \in \{0,1\}^n : \tau'|_S = \tau} \mu(\tau') \qquad \forall \tau \in \{0,1\}^S.$$

When $S = \{v\}$ *is a single vertex, we let* $\mu_v = \mu_{\{v\}}$*. Furthermore, given some partial assignment* $\sigma \in \{0,1,\perp\}^\Lambda$ *for* $\Lambda \subseteq V$*, if* $S \cap \Lambda = \emptyset$*, we let* $\mu^\sigma_S(\cdot) := \mu_S(\cdot \mid \sigma)$ *be the marginal distribution on* S *conditioned on the partial assignment* σ*. When* $S = V \setminus \Lambda$*, we simply use* μ^σ *to denote* $\mu^\sigma_{V \setminus \Lambda}$*.*

2.1 The random local access model and local computation algorithms

One of the most widely studied models of local computation are local computation algorithms (LCAs) introduced in [2, 29]. Given a computation problem F, an LCA (in an *online fashion*) provides the i-th bit to *some* consistent solution F in sublinear time. As originally defined, local computation algorithms need not be *query-oblivious*; in other words, the output solution can depend on the order of queried bits. However, several follow-up works have given query-oblivious analogues of LCAs for a variety of natural problems. Such models are the non-random-sampling version of the random local access models we study here.

In this work, we construct a query-oblivious LCA for an intermediate step in our construction of the random local access model (described in more detail in Section 2.2). We thus precisely define both LCAs and random local access models below.

▶ **Definition 7** (Local computation algorithm)**.** *Given an object family* (Π, Ω) *with input* Π *and sample space* $\Omega \subseteq \Sigma^n$ *(for some alphabet* Σ*), a* $(t(n), \delta(n))$*-local computation algorithm (LCA) provides an oracle* $\mathcal{A}$ *that implements query access to some arbitrary* $\sigma \in \Omega$ *that satisfies the following conditions:*

- $\mathcal{A}$ *has query access to the input description* Π *and to a tape of public random bits* $\mathbf{R}$*.*
- $\mathcal{A}$ *gets a sequence of queries* $i_1, \ldots, i_q$ *for any* $q > 0$*, and after each query* i_j*, it produces an output* σ_{i_j} *such that the outputs* $\sigma_{i_1}, \ldots, \sigma_{i_q}$ *are consistent with some* $\sigma \in \Omega$*.*
- *The probability of success over all* q *queries is at least* $1 - \delta(n)$ *(where* $\delta(n) < 1/3$*).*
- *The expected running time of* $\mathcal{A}$ *on any query is at most* $t(n)$*, which is sublinear in* n*.*

We further say that $\mathcal{A}$ *is query-oblivious if the outputs of* $\mathcal{A}$ *do not depend on the order of the queries but depend only on* Π *and* $\mathbf{R}$*.*

Motivated by the above, we give a formal definition of the random local access model introduced in [5].

▶ **Definition 8** (Random local access). *Given a random object family (Π, Ω, μ) with input Π, sample space $\Omega \subseteq \Sigma^n$ (with alphabet Σ) and distribution μ supported on Ω, a $(t(n), \delta(n))$-random local access implementation of a family of query functions $\{F_1, F_2, \ldots\}$ provides an oracle $\mathcal{A}$ with the following properties:*

- *$\mathcal{A}$ has query access to the input description Π and to a tape of public random bits $\mathbf{R}$.*
- *For a given input Π, upon being queried with F_i, the oracle with probability $1 - o(1)$ uses at most $t(n)$ resources (where $t(n)$ is sublinear in n) to return the value $\mathcal{A}(\Pi, \mathbf{R}, F_i(Y))$ for some specific $Y \in \Omega$.*
- *The choice of $Y \in \Omega$ only depends on Π and $\mathbf{R}$.*
- *The distribution of Y over the randomness $\mathbf{R}$ satisfies*

$$d_{\mathsf{TV}}(Y, \mu) = \frac{1}{2} \sum_{\omega \in \Omega} |\mathbb{P}(Y = \omega) - \mu(\omega)| < \delta(n),$$

where $\delta(n) \lesssim \frac{1}{n^c}$ for constant $c > 1$.

In other words, if $\mathcal{A}$ is a random local access oracle for a set of queries, then when provided the same input Π and the same random bits $\mathbf{R}$, it must provide outputs that are consistent with a single choice of Y regardless of the order and content of these queries.

▶ **Remark 9.** *In this work, we do not study or seek to optimize the memory usage of our algorithms. However, there is also a rich literature studying space-efficient and parallelizable local models (e.g., [2, 14, 21]).*

2.2 Marking and a query-oblivious LCA

As noted in the introduction, the Lovász local lemma [30] guarantees the existence of a satisfying assignment to any (k, d)-formula Φ if $d \leq (2ek)^{-1}2^k$. Furthermore, the Moser-Tardos algorithm [27] gives a simple linear-time algorithm to construct such a satisfying assignment in the above (k, d)-regime:

- Sample $v_1, \ldots, v_n \xleftarrow{\mathbf{R}} \{0, 1\}$ uniformly at random;
- While there exists a clause $C \in \mathcal{C}$ that is violated by the current assignment, resample variables in C uniformly at random.

The Lovász local lemma can be made *quantitative*, showing that not only is there some satisfying assignment to Φ if $d \leq (2ek)^{-1}2^k$, but both that there are exponentially many such satisfying assignments and that the marginal distributions μ_v are approximately uniform with $\mu_v(1) \leq \frac{1}{2}\exp(1/k)$ (see [25, 10]). Such *local uniformity* is critical to the success of algorithms that approximately sample from μ. In his breakthrough work, Moitra [25] noted that this local uniformity continues to hold for conditional distributions μ^σ provided that each clause has a sufficiently large number of free variables under partial assignment σ. This motivates the idea of a *marking*, as introduced in [25], which is a careful selection (via the Lovász local lemma) of a linear sized subset of variables $\mathcal{M} \subseteq V$ that has the following properties:

- For every clause C, $|\operatorname{vbl}(C) \cap \mathcal{M}| \gtrsim k$. Having a large number of marked variables in each clause would hopefully result in a desired *shattering condition*; namely, we can sample a partial assignment $\sigma \in \{0, 1\}^{\mathcal{M}}$ on this marking that partitions the original formula into sufficiently small connected components.
- For every clause C, $|\operatorname{vbl}(C) \backslash \mathcal{M}| \gtrsim k$. This large non-intersection preserves the *local uniformity* of the marginal distributions of the as yet-unsampled variables in $\mathcal{M}$.

The general strategy of arguments following the marking approach is to show that it is "easy" to sample a partial assignment $\sigma \in \{0,1\}^{\mathcal{M}}$, and moreover, conditioned on any such assignment, the reduced formula Φ^σ is very likely to shatter into sufficiently small connected components such that the remaining variable assignments can be efficiently sampled from the conditional distribution by solving a smaller instance of the original problem. We now make this notion precise.

▶ **Definition 10** (α-marking). *Given (k,d)-formula Φ and constant $\alpha \in (0,1/2)$, we say that $\mathcal{M} \subseteq V$ is an α-marking if for every $C \in \mathcal{C}$, $|\operatorname{vbl}(C) \cap \mathcal{M}| \geq \alpha k$ and $|\operatorname{vbl}(C) \backslash \mathcal{M}| \geq \alpha k$.*

In this work, we locally, greedily construct an α-marking $\mathcal{M}$ using a local implementation of Moser-Tardos. We will further argue that because of the shattering property, we can locally compute the connected component of Φ_v^σ for some $\sigma \sim \mu_{\mathcal{M}}$ and a given vertex v, without having to actually assign linearly many vertices.

Precisely, we construct a query-oblivious LCA, $\mathsf{IsMarked}(\cdot)$, where for a (k,d)-formula Φ and a given range of $\alpha \in (0,1)$, $\mathsf{IsMarked}(\cdot)$ can take in any variable $v \in V$ and output either 0 or 1 indicating whether v is in some α-marking of V. Moreover, $\mathsf{IsMarked}(\cdot)$ takes sublinear time and when queried on all of V, gives a consistent α-marking of V.

▶ **Theorem 11.** *Let $\Phi = (V, \mathcal{C})$ be a (k,d)-formula. Suppose there exist constants $1/2 < \beta_1 < \beta_2 < 1 - \alpha$ that satisfy the following conditions:*

$$
\begin{aligned}
&4\alpha < 2(1-\beta_2) < 1-\beta_1,\\
&16k^4d^5 \leq 2^{(\beta_1 - h(1-\beta_1))k},\\
&16k^4d^5 \leq 2^{(\beta_2-\beta_1)k - h\left(\frac{\beta_2-\beta_1}{1-\beta_1}\right)(1-\beta_1)k},\\
&\delta \mapsto (\beta_2 - \delta) - h\left(\frac{\beta_2-\delta}{1-\delta}\right)(1-\delta) \text{ is decreasing on } 0 \leq \delta \leq \beta_1,\\
&2e(kd+1) < 2^{\left(1-h\left(\frac{\alpha}{1-\beta_2}\right)\right)(1-\beta_2)k}.
\end{aligned}
\tag{1}
$$

(Here $h(x) := -x\log_2(x) - (1-x)\log_2(1-x)$ is the binary entropy.)

Fix constant $c > 0$. Then there exists a $\operatorname{polylog}(n)$ *time local computation algorithm $\mathsf{IsMarked}(\cdot)$ which, given any variable $v \in V$, returns an assignment $s_v \in \{0,1\}$ denoting whether v is contained in an α-marking of Φ. Moreover, with probability at least $1 - n^{-c}$, the responses for all $v \in V$ yield a consistent α-marking of Φ.*

The construction of $\mathsf{IsMarked}(\cdot)$ and the verification that it is a query-oblivious LCA draws inspiration from the approach in [2] to obtain an oblivious LCA for hypergraph 2-coloring. From a high level, $\mathsf{IsMarked}(\cdot)$ determines s_v by randomly and greedily including variables in the marking and subsequently determining those that must/must not be in the marking, and finally (if needed) performing a rejection sampling on a smaller connected component that contains v. The formulae in Equation (1) are some technical conditions that guarantee this process to go through. We refer the readers to Appendix E of the full version (arxiv:2409.03951) for a proof of Theorem 11.

3 A random local access algorithm for k-SAT solutions

In this section, we introduce the main algorithm (Algorithm 1) that locally accesses the assignment of a variable $v \in V$ in a uniformly random satisfying assignment σ to Φ.

Recall from Theorem 1 that given a (k,d)-formula $\Phi = (V, \mathcal{C})$, variable $v \in V$, and any constant $c > 0$, we wish to come up with a random local access algorithm $\mathcal{A}$ such that (1) the expected running time of $\mathcal{A}$ is $\operatorname{poly}\log(n)$, and (2) the output $\widehat{\mu}_v$ of $\mathcal{A}$ for every $v \in V$ satisfies $d_{\mathsf{TV}}(\mathsf{Law}(\widehat{\mu}_v), \mu_v) \leq \frac{1}{n^c}$. As a high level description, given the above inputs, Algorithm 1 performs the following:

1. Locally decides whether v lies in an α-marking of Φ using $\mathsf{IsMarked}(v)$ (Theorem 11), such that the responses for all $v \in V$ yield a consistent α-marking $\mathcal{M} \subseteq V$.
2. Suppose $\sigma \sim \mu$ is a uniformly random satisfying assignment to Φ. If v is marked, then we sample $\sigma(v)$ by computing $\mathsf{MarginSample}(v)$ (adapted from [9, Algorithm 8]), which may make further recursive calls to sample other *marked* variables.
3. If v is not marked, then we perform a depth-first search starting from v to compute σ restricted to $\mathcal{M}$. We start from $\sigma = \emptyset$; for every $w \in \mathcal{M}$ we encounter that we have not yet sampled, we compute $\mathsf{MarginSample}(w)$ and update $\sigma(w)$, to eventually obtain a (w.h.p. polylogarithmic in size) connected component $\Phi_v := \Phi_v^\sigma$ that contains v. This part is carried out by the algorithm $\mathsf{Conn}(v)$.
   After obtaining the connected component Φ_v^σ, we call $\mathsf{UniformSample}$ (Theorem 13) to sample a uniformly random satisfying assignment to Φ_v^σ and extend σ. We then output $\sigma(v)$.

Algorithm 1 The sampling algorithm.

Input: k-CNF formula $\Phi = (V, \mathcal{C})$ and variable $v \in V$
Output: random assignment $\sigma(v) \in \{0,1\}$

```
if IsMarked(v) then
    return MarginSample(v)
else
    Φ_v ← Conn(v)
    σ ← UniformSample(Φ_v)
    return σ(v)
```

To illustrate the workflow of Algorithm 1, we present a toy example on a small k-CNF formula, omitting some subroutine details for brevity.

▶ **Example 12.** Suppose $k = 3$, $d = 2$ and $\alpha = 1/3$. Consider the following (k,d)-formula $\Phi = (V, \mathcal{C})$ with variables $V = \{x_1, x_2, x_3, x_4, x_5\}$ and clauses $\mathcal{C} = \{C_1, C_2, C_3\}$, where

$$C_1 = (x_1 \vee x_2 \vee \neg x_3), \quad C_2 = (\neg x_2 \vee x_3 \vee x_4), \quad C_3 = (\neg x_4 \vee x_5 \vee \neg x_1).$$

Suppose we wish to approximately sample $\sigma(x_1)$ using the local access algorithm $\mathcal{A}$. We run $\mathsf{IsMarked}$ on each variable; suppose the resulting marking is $\mathcal{M} = \{x_2, x_5\}$, so that x_1 is not marked.

Since $x_1 \notin \mathcal{M}$, we call $\mathsf{Conn}(x_1)$ to explore the connected component of $\Phi^{\mathcal{M}}$ containing x_1. We initialize $\mathcal{S} := \{C_1, C_3\}$ and partial assignment $\sigma = \emptyset$.

- Process clause C_1. Since $x_2 \in \mathcal{M}$, we call $\mathsf{MarginSample}(x_2)$, which returns (say) $\sigma(x_2) = 0$. The clause becomes $(x_1 \vee 0 \vee \neg x_3) = (x_1 \vee \neg x_3)$, which is unsatisfied with the current σ. We add adjacent clause C_2 (via x_2 and x_3) to $\mathcal{S}$.
- Process clause C_3. Since $x_5 \in \mathcal{M}$, we call $\mathsf{MarginSample}(x_5)$, which returns (say) $\sigma(x_5) = 1$. The clause becomes $(\neg x_4 \vee 1 \vee \neg x_1)$, which is satisfied.
- Process clause C_2. We already have $\sigma(x_2) = 0$, so $\neg x_2 = 1$, and the clause is satisfied. Remove C_2 from $\mathcal{S}$.

Now the discovered component $\Phi^{\sigma}_{x_1}$ includes (1) clauses C_1, (2) marked variable assignments $\sigma(x_2) = 0$, $\sigma(x_5) = 1$, (3) free variables: x_1, x_3, x_4. We now run UniformSample on the reduced subformula $C_1' = (x_1 \vee \neg x_3)$. The satisfying assignments (x_1, x_3) are $\{(1,0),(1,1),(0,0)\}$. We pick one uniformly at random, say $(x_1, x_3) = (1,0)$. Then we return $\widehat{\mu}_{x_1} = 1$.

Algorithm 1 is our main routine; it has several subroutines IsMarked, MarginSample, Conn, and UniformSample that we now describe. Recall that IsMarked has been discussed in Theorem 11. We now introduce the algorithm UniformSample given by the work of He–Wang–Yin [18].

▶ **Theorem 13** ([18, Theorems 5.1 and 6.1]). *Suppose Φ is a (k,d)-CNF formula on n variables with $k \cdot 2^{-k} \cdot (dk)^5 \cdot 4 \le \frac{1}{150e^3}$. Then there exists an algorithm* UniformSample *that terminates in $O(k^3(dk)^9 n)$ time in expectation, and outputs a uniformly random satisfying assignment to Φ.*

As seen in Theorem 13, for an n-variable (k,d)-CNF formula, the algorithm has running time $O(n)$. However, as we will only apply UniformSample to connected components of size $\operatorname{poly}\log(n)$ (that are shattered by the partial assignment on the marked variables), the execution of UniformSample in Line 6 of Algorithm 1 will only take polylogarithmic time.

We will define subroutines MarginSample and Conn below and show that they satisfy the desired correctness and efficiency properties, beginning by verifying MarginSample has the desired properties in Section 4.

▶ **Theorem 14.** *Let $\Phi = (V, \mathcal{C})$ be a (k,d)-formula, $\alpha > 0$, and $\mathcal{M} \subseteq V$ be an α-marking as in Definition 10. Fix positive constant $c > 0$. Suppose $\theta := 1 - \frac{1}{2}\exp\left(\frac{2edk}{2^{\alpha k}}\right) \ge 0.4$, $36ed^3k^4 \cdot 0.6^{\alpha k} \le 1/2$, and $2^{-\frac{1}{48dk^4}} \cdot e^{\frac{2d^2/\alpha}{2^{\alpha k}}} \le 0.9$. Then there exists a random local access algorithm* MarginSample$(\cdot)$ *such that for every $u \in \mathcal{M}$,* MarginSample(u) *returns a random value in $\{0,1\}$ that satisfies the following:*

1. *Let $\nu := \mu_{\mathcal{M}}$ and $\widehat{\nu}$ be the joint distribution of the outputs* $(\text{MarginSample}(u))_{u\in\mathcal{M}}$. *Then we have $d_{\mathsf{TV}}(\widehat{\nu}, \nu) < n^{-c}$.*
2. *For every $u \in \mathcal{M}$, the expected cost of* MarginSample(u) *is* $\operatorname{poly}\log(n)$.

We also require Conn to be correct and have low expected cost.

▶ **Theorem 15.** *Let $\Phi = (V, \mathcal{C})$ be a (k,d)-formula, $\alpha > 0$, and $\mathcal{M} \subseteq V$ be an α-marking as in Definition 10. Suppose $\theta := 1 - \frac{1}{2}\exp\left(\frac{2edk}{2^{\alpha k}}\right) \ge 0.4$ and $d \le 2^{\alpha k/4}$. Then there exists a random local access algorithm* Conn$(\cdot)$ *such that for every $v \in V \setminus \mathcal{M}$,* Conn$(v)$ *returns the connected component in Φ^{σ} that contains v, where σ is the partial assignment given by* $(\text{MarginSample}(u))_{u\in\mathcal{M}}$. *Moreover, for every $v \in V \setminus \mathcal{M}$, the expected cost of* Conn(v) *is* $\operatorname{poly}\log(n)$.

From a high level, the algorithm Conn(v) explores the clauses and marked variables in the CNF formula that are reachable from v, greedily sampling the marked variables and expanding through unsatisfied clauses. It iteratively builds a partial assignment σ that "shatters" the formula into disconnected components, isolating the one containing v. We will verify Theorem 15 in Appendix D of the full version (arxiv:2409.03951).

4 Proof of Theorem 14

In this section we show Theorem 14. Throughout, let $\Phi = (V, \mathcal{C})$ be a (k,d)-formula, $\alpha > 0$, and $\mathcal{M} \subseteq V$ be an α-marking with $|\mathcal{M}| = m$. We introduce a local access algorithm MarginSample that satisfies the key property that the joint distribution of outputs

$(\mathsf{MarginSample}(u))_{u\in\mathcal{M}}$ consistently follows the marginal distribution $\mu_{\mathcal{M}}$. In particular, $(\mathsf{MarginSample}(u))_{u\in\mathcal{M}}$ will simulate the output of a *systematic scan Glauber dynamics* on the marked variables.

▶ **Definition 16.** *Let $\mathcal{M} = \{u_1, \ldots, u_m\}$ denote the marked variables (so $|\mathcal{M}| = m$).*
- *For every time $t \in \mathbb{Z}$, define $i(t) := (t \mod m) + 1$.*
- *For every time $t \in \mathbb{Z}$ and $u_i \in \mathcal{M}$, define $\mathsf{pred}_{u_i}(t) := \max\{s \le t : i(s) = i\}$.*

In the systematic scan Glauber dynamics, we always resample vertex $u_{i(t)}$ at time t (as opposed to randomly choosing a variable to resample at each step). For every $u \in \mathcal{M}$ and time t, $\mathsf{pred}_u(t)$ denotes the most recent time up to time t at which u got resampled. Observe that for all $t \in \mathbb{Z}$ and $u \in \mathcal{M}$, we have $t - m < \mathsf{pred}_u(t) \le t$. Moreover, for all $w \neq u_{i(t)}$, we have $\mathsf{pred}_w(t) < t$.

Algorithm 2 Systematic scan projected Glauber dynamics.

Input: a k-CNF formula $\Phi = (V, \mathcal{C})$, a set of marked variables $\mathcal{M} \subseteq V$, time T, and an ordering $\mathcal{M} = \{u_1, \ldots, u_m\}$

Output: a random assignment $X_* \in \{0,1\}^{\mathcal{M}}$

1: Sample $X_0(u) \overset{\mathbf{R}}{\leftarrow} \{0,1\}$ uniformly and independently for each $u \in \mathcal{M}$
2: **for** $t = 1, \ldots, T$ **do**
3: Let $u := u_{i(t)}$
4: For all $u' \in \mathcal{M} \setminus \{u\}$, $X_t(u') \leftarrow X_{t-1}(u')$
5: Sample $X_t(u) \overset{\mathbf{R}}{\leftarrow} \mu_u(\cdot \mid X_{t-1}(\mathcal{M}\setminus\{u\})$
6: Return $X_* = X_T$

We refer the readers to Appendix A of the full version (arxiv:2409.03951) for more introduction on the systematic scan Glauber dynamics and its comparison with the (original) projected Glauber dynamics Markov chain for sampling k-SAT solutions.

We have the following non-quantitative convergence for systematic scan Glauber dynamics.

▶ **Theorem 17** ([22]). *Let $(X_t)_{t=0}^{\infty}$ denote the Glauber dynamics or the systematic scan Glauber dynamics with stationary distribution π. If $(X_t)_{t=0}^{\infty}$ is irreducible over $\Omega \subseteq \{0,1\}^{\mathcal{M}}$, then for every $X_0 \in \Omega$, we have $\lim_{t\to\infty} d_{\mathsf{TV}}(X_t, \pi) = 0$.*

Let $\mathcal{M} \subseteq V$ be a fixed set of marked variables for a given k-CNF Φ. Let $\nu := \mu_{\mathcal{M}}$ be the marginal distribution of μ on $\mathcal{M}$. We will crucially use the following *local uniformity* of Algorithm 2 (the proof in the systematic scan setting follows essentially identically to [10, Lemma 5.3]):

▶ **Lemma 18.** *Suppose Φ is a k-CNF with $1 < s \le \frac{2^{\alpha k}}{2edk}$. Let $X \subseteq \{0,1\}^{\Lambda}$ be either X_0 or $X_t(\mathcal{M}\setminus\{u_{i(t)}\})$ for some $t > 0$ (so correspondingly, Λ is either $\mathcal{M}$ or $\mathcal{M}\setminus\{u_{i(t)}\}$ for some $t > 0$). Then for any $S \subseteq \Lambda$ and $\sigma : S \to \{0,1\}$, we have*

$$\mathbb{P}(X(S) = \sigma) \le \left(\frac{1}{2}\right)^{|S|} \exp\left(\frac{|S|}{s}\right).$$

Specifically, for any $v \in \mathcal{M}$ and $c \in \{0,1\}$, we have $\mathbb{P}(X(v) = c) \ge 1 - \frac{1}{2}\exp(\frac{1}{s}) \ge \frac{1}{2} - \frac{1}{s}$.

▶ **Definition 19.** *Let π be a distribution on $\{0,1\}^{\mathcal{M}}$. We say that π is b-marginally lower bounded if for all $v \in \mathcal{M}$, $\Lambda \subseteq \mathcal{M} \setminus \{v\}$ and feasible partial assignment $\sigma_\Lambda \in \{0,1,\perp\}^{\Lambda}$, we have*

$$\pi_v^{\sigma_\Lambda}(0), \pi_v^{\sigma_\Lambda}(1) \ge b.$$

Let π be a b-marginally lower bounded distribution over $\{0,1\}^{\mathcal{M}}$. For every $v \in \mathcal{M}$, we define the following distributions:

1. *Lower bound distribution π_v^{LB} over $\{0,1,\perp\}$: we define $\pi_v^{\mathrm{LB}} := \pi^{\mathrm{LB}}$, with*

$$\pi^{\mathrm{LB}}(0) = \pi^{\mathrm{LB}}(1) = b, \qquad \pi^{\mathrm{LB}}(\perp) = 1 - 2b.$$

2. *Padding distribution $\pi_v^{\mathrm{pad},\sigma_\Lambda}$ over $\{0,1\}$: for $\Lambda \subseteq \mathcal{M} \setminus \{v\}$ and feasible partial assignment $\sigma_\Lambda \in \{0,1,\perp\}^\Lambda$, we define*

$$\pi_v^{\mathrm{pad},\sigma_\Lambda}(\cdot) := \frac{\pi_v^{\sigma_\Lambda}(\cdot) - b}{1-2b}.$$

Per Lemma 18, we have that $\nu = \mu_{\mathcal{M}}$ is θ-lower bounded for

$$\theta := 1 - \frac{1}{2}\exp\left(\frac{2edk}{2^{\alpha k}}\right) \geq \frac{1}{2} - \frac{1}{2^{(\alpha - 2\log_2 d)k}}. \tag{2}$$

4.1 Systematic scan Glauber dynamics on marked variables

We adapt the approach of [9] to a local sampling algorithm by simulating the systematic scan projected Glauber dynamics on $\mathcal{M}$ from time $-T$ to 0, which is an aperiodic and irreducible Markov chain by results in [10, 9].

Let $(X_t)_{-T \leq t \leq 0}$ be the output of Algorithm 2, where we relabel $X_0, \ldots, X_T$ by $X_{-T}, \ldots, X_0$. We know from Theorem 17 that, as $T \to \infty$, we have $d_{\mathsf{TV}}((X_0(v))_{v\in\mathcal{M}}, \nu) \to 0$ where $\nu = \mu_{\mathcal{M}}$ is the marginal distribution of μ on the marked variables. In particular, for every fixed n and $\gamma > 0$, there exists $T_\gamma \in \mathbb{N}$ such that for all $T > T_\gamma$, the Markov chain $(X_t)_{-T\leq t\leq 0}$ satisfies $d_{\mathsf{TV}}((X_0(v))_{v\in\mathcal{M}}, \nu) < \gamma$.

We know from Lemma 18 that ν is θ-lower bounded, with the lower bound distribution ν^{LB} defined by $\nu^{\mathrm{LB}}(0) = \nu^{\mathrm{LB}}(1) = \theta$ and $\nu^{\mathrm{LB}}(\perp) = 1 - 2\theta$. Thus, for every $-T < t \leq 0$ and $u = u_{i(t)}$, sampling $X_t(u) \sim \nu_u^{X_{t-1}(\mathcal{M}\setminus\{u\})}$ can be decomposed into the following process:

1. With probability 2θ, set $X_t(u)$ to 0 or 1 uniformly at random;
2. With probability $1 - 2\theta$, sample $X_t(u)$ from the padding distribution $\nu_u^{\mathrm{pad},X_{t-1}(\mathcal{M}\setminus\{u\})}$.

Our goal is to obtain $X_0(u)$ for $u \in \mathcal{M}$, which by Theorem 17 will closely follow the marginal distribution ν_u for T sufficiently large. It suffices to simulate the last update for u. The key observation here is that updates of Glauber dynamics may depend only on a very small amount of extra information. When θ is close to $1/2$, it is reasonably likely that we can determine $X_0(u) = X_{\mathsf{pred}_u(0)}(u)$ without any other information. Thus, we can deduce $X_0(u)$ by recursively revealing only the necessary randomness backwards in time. This method was termed *coupling towards the past* and studied for a variety of constraint satisfaction problems in [9].

We now give a general algorithm $\mathsf{Glauber}_{T,Y}(t, M, R)$ in Algorithm 5, whose output *simulates* the effects of Algorithm 2 at any particular time t, by looking *backwards* in time at what happened over the course of running $(X_t)_{-T\leq t\leq 0}$. The eventual algorithm $\mathsf{MarginSample}(u)$ we give in Theorem 14 will retrieve the most recent update of each variable u, i.e., retrieve the coordinate $X_0(u)$.

The algorithm $\mathsf{Glauber}_{T,Y}(t, M, R)$ contains another subroutine $\mathsf{LB\text{-}Sample}_{T,Y}(t, R)$ that is defined in Algorithm 4. For every time t, the output of $\mathsf{LB\text{-}Sample}_{T,Y}(t, R)$ follows the distribution ν^{LB} (see Definition 19). In other words, $\mathsf{LB\text{-}Sample}_{T,Y}(t, R)$ preliminarily decides which of the above two regimes we fell into while resampling $X_{u_{i(t)}}$ at time t.

Throughout Algorithm 5, we maintain two global data structures.

Algorithm 3 MarginSample(u).

Input: a k-CNF formula $\Phi = (V, \mathcal{C})$, a set of marked variables $\mathcal{M} = \{u_1, \ldots, u_m\} \subseteq V$, and a marked variable $u \in \mathcal{M}$

Output: a random value in $\{0, 1\}$

1: $T \leftarrow T_{n^{-(2+c)}}$
2: $Y \overset{\mathbf{R}}{\leftarrow} \{0,1\}^{\mathcal{M}}$
3: return $\mathsf{Glauber}_{T,Y}(\mathsf{pred}_u(0), M = \perp^{\mathbb{Z}}, R = \emptyset)$

- We let $M : \mathbb{Z} \to \{0, 1, \perp\}$ record the previously revealed outcomes of Algorithm 5. That is, for every $t' \leq 0$ such that $\mathsf{Glauber}_{T,Y}(t', M, R)$ has already been executed, we set $M(t')$ to be the outcome of $\mathsf{Glauber}_{T,Y}(t', M, R)$.
- We let $R = \{(s, r_s)\} \subseteq \mathbb{Z} \times \{0, 1, \perp\}$ record the previously revealed outcomes of Algorithm 4. That is, for every $t' \leq 0$ such that $\mathsf{LB\text{-}Sample}_{T,Y}(t', M, R)$ has already been executed and returned $r_{t'} \in \{0, 1, \perp\}$, we add $\{(t', r_{t'})\}$ to R.

Since T, Y remain constant throughout Algorithms 4 and 5, and all recursive calls access and update the same M and R, we sometimes write $\mathsf{Glauber}(t) = \mathsf{Glauber}_{T,Y}(t, M, R)$ and $\mathsf{LB\text{-}Sample}(t) = \mathsf{LB\text{-}Sample}_{T,Y}(t, R)$ for short.

At the beginning of $\mathsf{Glauber}(t)$, we first check a few early stopping conditions:

- (Lines 1–2) If variable $u_{i(t)}$ remains its initial assignment $Y(u_{i(t)})$ at the end of time t (i.e., is never resampled), we terminate and return $Y(u_{i(t)})$.
- (Lines 3–4) If $|R|$, the number of stored outcomes of $\mathsf{LB\text{-}Sample}$, already reaches $80dk^4 \log n$, we terminate and return 1.
- (Lines 5–6) If previous iterations have already computed $\mathsf{Glauber}(t)$ and stored $M(t) \in \{0, 1\}$, we terminate and return $M(t)$.

If none of the above conditions occurs, we then resample, first by applying $\mathsf{LB\text{-}Sample}(t)$ (Algorithm 4) in Lines 7–8. If $\mathsf{LB\text{-}Sample}(t) \in \{0, 1\}$ (which occurs with probability 2θ), we can update $u = u_{i(t)}$ by choosing an assignment from $\{0, 1\}$ uniformly at random without investigating the assignments of any other variables at earlier time steps (i.e., we fall into the *zone of local uniformity*).

If $\mathsf{LB\text{-}Sample}(t) = \perp$ (which occurs with probability $1 - 2\theta$), then we fall into a *zone of indecision* and must resample $u = u_{i(t)}$ from the padding distribution $\nu_u^{\mathrm{pad}, X_{t-1}(\mathcal{M} \setminus \{u\})}$. To resample its spin, we slowly proceed backwards in time, lazily computing previously resampled assignments, until we have determined enough information to compute the assignment of u at the desired time step. Verifying accuracy is somewhat involved, given our lazy computation strategy and partitioning of ν into a locally uniform piece and an associated padding distribution. Thus, in Section A we show that Lines 9–19 correctly complete the desired task, proving the following bound on $d_{\mathsf{TV}}((\mathsf{MarginSample}(u))_{u \in \mathcal{M}}, \nu)$.

▶ **Proposition 20.** *For any $c > 0$ and n sufficiently large, we have*

$$d_{\mathsf{TV}}((\mathsf{MarginSample}(u))_{u \in \mathcal{M}}, \nu) < n^{-c}.$$

We next require that Algorithm 5 has expected polylogarithmic cost. This is largely a consequence of the local uniformity of $\mu_{\mathcal{M}}$ and our lazy recursive computation of variable assignments in Algorithm 5.

▶ **Lemma 21.** *Suppose $2^{-\frac{1}{48dk^4}} \cdot e^{\frac{2d^2/\alpha}{2^{\alpha k}}} \leq 0.9$. For every $t \leq 0$, the expected cost of $\mathsf{Glauber}(t)$ is $O(k^{17} d^{10} \log^2 n/\alpha)$.*

Algorithm 4 $\mathsf{LB\text{-}Sample}_{T,Y}(t, R)$.

Input: An integer $T \geq 0$ and assignment $Y \in \{0,1\}^{\mathcal{M}}$; an integer $t \leq 0$
Global variables: a set $R \subseteq \mathbb{Z} \times \{0, 1, \perp\}$ and α-marking $\mathcal{M} = \{u_1, \ldots, u_m\}$
Output: a random value in $\{0, 1, \perp\}$ distributed as ν^{LB}

if $\mathsf{pred}_{u_{i(t)}}(t) \leq -T$ **then**
  return $Y(u_{i(t)})$
if $(t, r) \in R$ **then**
  return r
Draw $x \in [0, 1]$ uniformly at random
if $x < 2\theta$ **then**
  $R \leftarrow R \cup \{(t, \lfloor x/\theta \rfloor)\}$ and return $\lfloor x/\theta \rfloor$
$R \leftarrow R \cup \{(t, \perp)\}$ and return $\perp$

Algorithm 5 $\mathsf{Glauber}_{T,Y}(t, M, R)$.

Input: An integer $T \geq 0$ and assignment $Y \in \{0,1\}^{\mathcal{M}}$; an integer $t \leq 0$
Global variables: (k, d)-CNF $\Phi = (V, \mathcal{C})$, α-marking $\mathcal{M} = \{u_1, \ldots, u_m\}$, $M : \mathbb{Z} \to \{0, 1, \perp\}$, and $R \subseteq \mathbb{Z} \times \{0, 1, \perp\}$
Output: a random value in $\{0, 1\}$

if $\mathsf{pred}_{u_{i(t)}}(t) \leq -T$ **then**
  return $Y(u_{i(t)})$
if $|R| \geq 80dk^4 \log n$ **then**
  return 1
if $M(t) \neq \perp$ **then**
  return $M(t)$
if $\mathsf{LB\text{-}Sample}_{T,Y}(t, R) \neq \perp$ **then**
  $M(t) \leftarrow \mathsf{LB\text{-}Sample}_{T,Y}(t, R)$ and return $M(t)$
$u \leftarrow u_{i(t)}$, $\Lambda \leftarrow \emptyset$, $\sigma_\Lambda \leftarrow \emptyset$, $V' \leftarrow \{u\}$
while $\exists C \in \mathcal{C}$ such that $\mathrm{vbl}(C) \cap V' \neq \emptyset$, $\mathrm{vbl}(C) \cap (V \setminus V') \neq \emptyset$ and C is not satisfied by σ_Λ **do**
  choose C with the lowest index
  if for all marked $w \in \mathrm{vbl}(C) \setminus \{u\}$, $\mathsf{LB\text{-}Sample}_{T,Y}(\mathsf{pred}_w(t), R)$ does not satisfy C **then**
    for all marked $w \in \mathrm{vbl}(C) \setminus \{u\}$ **do**
      $\Lambda \leftarrow \Lambda \cup \{w\}$ and $\sigma_\Lambda(w) \leftarrow \mathsf{Glauber}_{T,Y}(\mathsf{pred}_w(t), M, R)$
    $V' \leftarrow V' \cup \mathrm{vbl}(C)$
  else
    for all marked $w \in \mathrm{vbl}(C) \setminus \{u\}$ **do**
      $\Lambda \leftarrow \Lambda \cup \{w\}$ and $\sigma_\Lambda(w) \leftarrow \mathsf{LB\text{-}Sample}_{T,Y}(\mathsf{pred}_w(t), R)$
Let Ψ be the connected component in Φ^{σ_Λ} with u, and sample $c \sim (\nu_\Psi)_u^{\mathrm{pad},\sigma_\Lambda}$
$M(t) \leftarrow c$ and return $M(t)$

We prove Lemma 21 in Section B. These two results together allow us to prove Theorem 14.

Proof of Theorem 14. The theorem directly follows from combining Proposition 20 and Lemma 21. By Proposition 20, we know that the joint distribution $\widehat{\nu} := (\mathsf{MarginSample}(u))_{u\in\mathcal{M}}$ satisfies $d_{\mathsf{TV}}(\widehat{\nu},\nu) < n^{-c}$. By Lemma 21, we know that for every $u \in \mathcal{M}$, the expected cost of $\mathsf{MarginSample}(u) = \mathsf{Glauber}_{T,Y}(\mathsf{pred}_u(0), M = \perp^{\mathbb{Z}}, R = \emptyset)$ is of order $\operatorname{poly}\log(n)$. ◀

5 Proof of the main theorem

We are finally able to state and prove the formal version of Theorem 1. Before picking the relevant parameters, we first collect the list of conditions required to apply Theorems 13–15.

▶ **Condition 22.**

$$\begin{aligned}
&k \cdot 2^{-\alpha k} \cdot (dk)^5 \cdot 4 \le \frac{1}{150e^3},\\
&\theta := 1 - \frac{1}{2}\exp\left(\frac{2edk}{2^{\alpha k}}\right) \ge 0.4,\\
&36ed^3k^4 \cdot 0.6^{\alpha k} \le 1/2,\\
&2^{-\frac{1}{48dk^4}} \cdot e^{\frac{2d^{2/\alpha}}{2^{\alpha k}}} \le 0.9,\\
&d \le 2^{\alpha k/4}.
\end{aligned} \tag{3}$$

Recall that we also need conditions Equation (1) to apply Theorem 11. One can verify that for $d \le 2^{k/400}$ and k sufficiently large, we can choose all the parameters appropriately so that Equations (1) and (3) are satisfied.

▶ **Lemma 23.** *Let*

$$\alpha = 1/75, \qquad \beta_1 = 0.778, \qquad \beta_2 = 0.96.$$

If k is sufficiently large, and $d = d(k) \le 2^{k/400}$, then Conditions Equations (1) *and* (3) *are satisfied.*

We defer the proof of Lemma 23 to Appendix F of the full version (arxiv:2409.03951). We can now state and prove Theorem 24, the formal version of our main result.

▶ **Theorem 24.** *Suppose $\Phi = (V, \mathcal{C})$ is a (k,d)-formula with $d \le 2^{k/400}$ and k sufficiently large. Let μ be the uniform distribution over satisfying assignments to Φ, with marginal distributions μ_v for $v \in V$. Then for all $c > 0$, there is a $(\operatorname{poly}\log(n), n^{-c})$-random local access algorithm $\mathcal{A}$ for sampling the variable assignment of $v \in V$ as $\widehat{\mu}_v$, such that*

$$d_{\mathsf{TV}}((\widehat{\mu_v})_{v\in V}, \mu) \le \frac{1}{n^c}.$$

Here we remark that $c > 0$ is any fixed constant, and the runtime of $\mathcal{A}$ depends on it. As written, both the algorithmic runtime and correctness are random, since we give expected running time and bounds on the marginal distribution in total variation distance. However, our algorithm allows derandomising either correctness or running time at the expense of worse bounds on the other.

Proof. Suppose $\Phi = (V, \mathcal{C})$ is a (k, d)-formula with $d \leq 2^{k/400}$ and k sufficiently large, and $c > 0$ is any constant. Choose parameters

$$\alpha = 1/75, \qquad \beta_1 = 0.778, \qquad \beta_2 = 0.96.$$

By Lemma 23, we know that with these parameters, conditions Equations (1) and (3) are satisfied. Thus, by Theorem 11, there exists a $\operatorname{poly}\log(n)$ time oblivious local computation algorithm $\mathsf{IsMarked}(\cdot)$ that with probability at least $1 - n^{-2c}$ gives a consistent α-marking $\mathcal{M} \subseteq V$.

Suppose $\mathsf{IsMarked}(\cdot)$ gives a consistent α-marking $\mathcal{M} \subseteq V$. By Theorem 14, we know that there is a random local access algorithm $\mathsf{MarginSample}(\cdot)$ with expected cost $\operatorname{poly}\log(n)$ such that the distribution of $(\mathsf{MarginSample}(u))_{u \in \mathcal{M}} \sim \widehat{\nu}$ satisfies $d_{\mathsf{TV}}(\widehat{\nu}, \mu_{\mathcal{M}}) < n^{-2c}$.

Let $\tau = (\mathsf{MarginSample}(u))_{u \in \mathcal{M}}$. By the proof of Theorem 15, we know that for every unmarked variable $v \in V \setminus \mathcal{M}$, with probability $1 - n^{-0.1 \log n}$, the number of clauses in Φ_v^τ is at most $kd \log^2 n$. We already proved in Theorem 15 that for every $v \in V \setminus \mathcal{M}$, the expected cost of $\mathsf{Conn}(v)$ is at most $\operatorname{poly}\log(n)$.

Furthermore, since the reduced formula Φ_v^τ has at least αk variables and at most k variables in each clause, and every variable lies in at most d clauses with $d \leq 2^{\alpha k / 5.4}$, by Theorem 13, the expected cost of $\mathsf{UniformSample}(\Phi_v^\tau)$ is asymptotically at most

$$k^3 (dk^9)(kd \log^2 n + n^{-0.1 \log n} \cdot n) = \operatorname{poly}\log(n).$$

Since both Conn and $\mathsf{UniformSample}$ succeed with probability 1, we get that $\widehat{\mu}$, the joint distribution of outputs of Algorithm 1 for all $v \in V$, satisfies $d_{\mathsf{TV}}(\widehat{\mu}, \mu) < n^{-c}$ for all $c > 0$.

By construction, Algorithm 1 is memory-less as it samples all necessary variable assignments in order to compute the assignment of a queried variable v. Furthermore, Algorithm 1 queried on different variables $v \in V$ collectively returns an assignment $\sigma \sim \widehat{\mu}$ that has $d_{\mathsf{TV}}(\widehat{\mu}, \mu) < n^{-c}$. Since this holds for any $c > 0$ constant, we obtain the desired result. ◀

6 Concluding remarks

With more involved refinements and optimizations of the arguments in this work, the density constraint $d \lesssim 2^{k/400}$ of Theorem 1 can be substantially improved (to something closer to $d \lesssim 2^{k/50}$). We omit these additional calculations in favor of expositional clarity to highlight our main result: random local access models exist for arbitrary bounded degree k-CNFs at exponential clause density. Furthermore, these arguments can also be adapted (in a similar fashion to [13, 19, 7]) to obtain a random local access model for *random* k-CNFs in a comparable density regime.

Nonetheless, the limit of the approaches in this work would still fall well short of obtaining random local access for, e.g., approximately $d \lesssim 2^{k/4.82}$, the maximum density at which we currently know how to efficiently sample solutions to an arbitrary bounded degree k-CNF in nearly-linear time [18, 31]. This is because of our reliance on a query-oblivious LCA to construct a local marking and our application of weaker sampling results to a correspondingly reduced CNF.

The approach we take in this work is only one of many possible schema to reduce from existing classical, parallel, and/or distributed algorithms to more local algorithms. Our approach involved using ideas and techniques from a variety of previous works (notably [25, 9, 10]), many of which were in the non-local setting, and adapting them in a novel way to obtain a sublinear sampler. Our approach bears some resemblance to work of [5] where authors adapted a parallel Glauber dynamics algorithm to obtain random local access to

proper q-colorings, and to work of [3] that used a recursive strategy to give perfect sampling algorithms from certain spin systems in amenable graphs. We expect that many other existing algorithms (including [18, 17, 19, 31]) can be adapted with some work to give random local access algorithms.

References

1 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '12, pages 5–14, New York, NY, USA, 2012. doi:10.1145/2213556.2213560.

2 Noga Alon, Ronitt Rubinfeld, Shai Vardi, and Ning Xie. Space-efficient local computation algorithms. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 1132–1139. SIAM, 2012. doi:10.1137/1.9781611973099.89.

3 Konrad Anand and Mark Jerrum. Perfect sampling in infinite spin systems via strong spatial mixing. *SIAM Journal on Computing*, 51(4):1280–1295, 2022. doi:10.1137/21M1437433.

4 Amartya Shankha Biswas, Edward Pyne, and Ronitt Rubinfeld. Local access to random walks. In *13th Innovations in Theoretical Computer Science Conference (ITCS 2022)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ITCS.2022.24.

5 Amartya Shankha Biswas, Ronitt Rubinfeld, and Anak Yodpinyanee. Local access to huge random objects through partial sampling. *11th Innovations in Theoretical Computer Science (ITCS 2020)*, 2020. doi:10.4230/LIPIcs.ITCS.2020.27.

6 Zongchen Chen, Aditya Lonkar, Chunyang Wang, Kuan Yang, and Yitong Yin. Counting random k-sat near the satisfiability threshold. *arXiv preprint*, 2024. doi:10.48550/arXiv.2411.02980.

7 Zongchen Chen, Nitya Mani, and Ankur Moitra. From algorithms to connectivity and back: finding a giant component in random k-SAT. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3437–3470. SIAM, 2023.

8 Paul Erdos and László Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. *Infinite and finite sets*, 10(2):609–627, 1975.

9 Weiming Feng, Heng Guo, Chunyang Wang, Jiaheng Wang, and Yitong Yin. Towards derandomising markov chain monte carlo. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1963–1990, 2023. doi:10.1109/FOCS57990.2023.00120.

10 Weiming Feng, Heng Guo, Yitong Yin, and Chihao Zhang. Fast sampling and counting k-SAT solutions in the local lemma regime. *J. ACM*, 68(6):Art. 40, 42, 2021. doi:10.1145/3469832.

11 Weiming Feng and Yitong Yin. On local distributed sampling and counting. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 189–198, 2018. doi:10.1145/3212734.3212757.

12 Andreas Galanis, Leslie Ann Goldberg, Heng Guo, and Andrés Herrera-Poyatos. Fast sampling of satisfying assignments from random k-SAT. *arXiv*, 2022. arXiv:2206.15308.

13 Andreas Galanis, Leslie Ann Goldberg, Heng Guo, and Kuan Yang. Counting solutions to random CNF formulas. *SIAM Journal on Computing*, 50(6):1701–1738, 2021. doi:10.1137/20M1351527.

14 Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1636–1653. SIAM, 2019. doi:10.1137/1.9781611975482.99.

15 Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, August 1986. doi:10.1145/6490.6503.

16 Oded Goldreich, Shafi Goldwasser, and Asaf Nussboim. On the implementation of huge random objects. *SIAM Journal on Computing*, 39(7):2761–2822, 2010. doi:10.1137/080722771.

17 Kun He, Xiaoming Sun, and Kewen Wu. Perfect sampling for (atomic) Lovász local lemma. *arXiv*, 2021. arXiv:2107.03932.

18 Kun He, Chunyang Wang, and Yitong Yin. Sampling Lovász local lemma for general constraint satisfaction solutions in near-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 147–158. IEEE, 2022. doi:10.1109/FOCS54457.2022.00021.

19 Kun He, Kewen Wu, and Kuan Yang. Improved bounds for sampling solutions of random CNF formulas. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3330–3361. SIAM, 2023. doi:10.1137/1.9781611977554.CH128.

20 Vishesh Jain, Huy Tuan Pham, and Thuy-Duong Vuong. On the sampling Lovász local lemma for atomic constraint satisfaction problems. *arXiv*, 2021. arXiv:2102.08342.

21 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *Journal of the ACM (JACM)*, 63(2):1–44, 2016. doi:10.1145/2742012.

22 David A. Levin and Yuval Peres. *Markov chains and mixing times*. American Mathematical Society, Providence, RI, second edition, 2017. With contributions by Elizabeth L. Wilmer, With a chapter on "Coupling from the past" by James G. Propp and David B. Wilson. doi:10.1090/mbk/107.

23 Hongyang Liu and Yitong Yin. Simple parallel algorithms for single-site dynamics. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1431–1444, 2022. doi:10.1145/3519935.3519999.

24 Yishay Mansour, Aviad Rubinstein, Shai Vardi, and Ning Xie. Converting online algorithms to local computation algorithms. In *Automata, Languages, and Programming*, pages 653–664, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-31594-7_55.

25 Ankur Moitra. Approximate counting, the Lovász local lemma, and inference in graphical models. *J. ACM*, 66(2):Art. 10, 25, 2019. doi:10.1145/3268930.

26 Peter Mörters, Christian Sohler, and Stefan Walzer. A sublinear local access implementation for the chinese restaurant process. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2022)*, 2022. doi:10.4230/LIPIcs.APPROX/RANDOM.2022.28.

27 Robin A. Moser and Gábor Tardos. A constructive proof of the general Lovász local lemma. *J. ACM*, 57(2):Art. 11, 15, 2010. doi:10.1145/1667053.1667060.

28 Moni Naor and Asaf Nussboim. Implementing huge sparse random graphs. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 596–608, 2007. doi:10.1007/978-3-540-74208-1_43.

29 Ronitt Rubinfeld, Gil Tamir, Shai Vardi, and Ning Xie. Fast local computation algorithms. *arXiv*, 2011. arXiv:1104.1377.

30 Joel Spencer. Asymptotic lower bounds for ramsey functions. *Discrete Mathematics*, 20:69–76, 1977. doi:10.1016/0012-365X(77)90044-9.

31 Chunyang Wang and Yitong Yin. A sampling lovász local lemma for large domain sizes. In *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 129–150. IEEE, 2024. doi:10.1109/FOCS61266.2024.00019.

A Correctness of Algorithm 5

In this section, we show Proposition 20 that with high probability, $(\mathsf{Glauber}(\mathsf{pred}_u(0)))_{u\in\mathcal{M}}$ faithfully returns the final outcome $(X_0(u))_{u\in\mathcal{M}}$ of the systematic scan Glauber dynamics $(X_t)_{-T\le t\le 0}$ initialized at $X_{-T} = Y$. We will later use Theorem 17 to show that therefore, when T is set to be sufficiently large, the distribution of $(\mathsf{MarginSample}(u))_{u\in\mathcal{M}} = (\mathsf{Glauber}(\mathsf{pred}_u(0)))_{u\in\mathcal{M}}$ is close to the marginal distribution $\nu = \mu_{\mathcal{M}}$.

▶ **Proposition 25.** *Fix $t \le 0$ and let $u = u_{i(t)}$. Suppose $|R| < 80dk^4 \log n$ after the execution of* $\mathsf{Glauber}(t)$ *(i.e., Line 4 of Algorithm 5 is never triggered). Then* $\mathsf{Glauber}(t)$ *faithfully returns $X_t(u)$, where $(X_t)_{-T \le t \le 0}$ is the systematic scan Glauber dynamics started at $X_{-T} = Y$.*

Proof. The statement clearly holds for $t = -T$ and $u = u_{i(-T)}$. Since $\mathsf{pred}_u(-T) = -T$, by Lines 1–2 of Algorithm 5, we have $\mathsf{Glauber}(-T) = Y(u) = X_{-T}(u)$.

Inductively, for $-T < t \le 0$, assume the proposition for all $-T \le t' < t$. Let $u = u_{i(t)}$. Suppose $|R| < 80dk^4 \log n$ after the execution of $\mathsf{Glauber}(t)$. Observe that, since $\mathsf{pred}_w(t) < t$ for all $w \neq u$, $\mathsf{Glauber}(t)$ only makes further calls to $\mathsf{Glauber}(t')$ with $t' < t$. Thus, by the inductive hypothesis, all further calls of $\mathsf{Glauber}(t')$ have correctly returned the outcomes $X_{t'}(u_{i(t')})$.

We wish to show that the resampled outcome $\mathsf{Glauber}(t)$ follows the marginal distribution $\nu_u^{X_{t-1}(\mathcal{M}\setminus\{u\})}$. Per Lines 5–6, we may assume that $\mathsf{Glauber}(t)$ has never been called before, in which case we directly go to Line 7 of Algorithm 5. Lines 7–8 guarantee that with probability 2θ, we assign $X_t(u)$ to be 0 or 1 uniformly at random. It remains to show that in Lines 9–19, we are able to resample $X_t(u)$ from the padding distribution $\nu_u^{\mathrm{pad},X_{t-1}(\mathcal{M}\setminus\{u\})}$.

To show this, we first verify that the sets Λ, V' and the partial assignment $\sigma_\Lambda \in \{0,1,\perp\}^\Lambda$ obtained in Line 19 satisfy the following four conditions:

(1) $u \in V'$;
(2) $(V' \cap \mathcal{M}) \subseteq \Lambda \cup \{u\}$;
(3) for all $C \in \mathcal{C}$ such that $\mathrm{vbl}(C) \cap V' \neq \emptyset$ and $\mathrm{vbl}(C) \cap (V \setminus V') \neq \emptyset$, C is satisfied by σ_Λ;
(4) for all marked variables $w \in V' \setminus \{u\}$, we have $\sigma_\Lambda(w) = X_{t-1}(w) \in \{0,1\}$; for all variables $w \in \Lambda \setminus V'$, we either have $\sigma_\Lambda(w) = X_{t-1}(w) \in \{0,1\}$, or have $\sigma_\Lambda(w) = \perp$.

Here, property (1) holds because u is added to V' in the initialization, and V' never removes variables. Property (2) holds because if variables in some clause C are added to V' in Line 15, then all marked variables in C are added to Λ in Line 14. As the while loop terminates, the opposite condition of Line 10 holds, which is exactly property (3).

We now show property (4). For every $w \in V' \setminus \{u\}$, we know that w is added to V' in Line 15 due to some clause C; if w is marked, then by Line 14 and the inductive hypothesis, we know that we have assigned $\sigma_\Lambda(w) \leftarrow \mathsf{Glauber}(\mathsf{pred}_w(t)) = X_{t-1}(w) \in \{0,1\}$. For every $w \in \Lambda \setminus V'$, by Line 18, we have assigned $\sigma_\Lambda(w) \leftarrow \mathsf{LB\text{-}Sample}(\mathsf{pred}_w(t))$. If $\mathsf{LB\text{-}Sample}(\mathsf{pred}_w(t)) \neq \perp$, then we have $\mathsf{LB\text{-}Sample}(\mathsf{pred}_w(t)) = X_{\mathsf{pred}_w(t)}(w) = X_{t-1}(w) \in \{0,1\}$.

Let Ψ denote the connected component in Φ^{σ_Λ} that contains u. Let μ_Ψ denote the distribution of a uniformly random satisfying assignment to Ψ. By property (3), we know that the connected component in Φ^{σ_Λ} that contains u is supported on V'. By property (4), we know that $X_{t-1}(\mathcal{M} \setminus \{u\})$ is an extension of σ_Λ, which means that the connected component in $\Phi^{X_{t-1}(\mathcal{M}\setminus\{u\})}$ that contains u is also supported on V'. Moreover, property (4) implies that $\sigma_\Lambda(V' \setminus \{u\}) = X_{t-1}(V' \setminus \{u\})$, which means that the two marginal distributions $(\mu_\Psi)_u^{\sigma_\Lambda}$ and $(\mu_\Psi)_u^{X_{t-1}(V'\setminus\{u\})}$ are the same. Altogether, we get that

$$\mu_u^{\sigma_\Lambda} = (\mu_\Psi)_u^{\sigma_\Lambda} = (\mu_\Psi)_u^{X_{t-1}(\mathcal{M}\setminus\{u\})} = \mu_u^{X_{t-1}(\mathcal{M}\setminus\{u\})}.$$

Recall that ν is the marginal distribution of μ on $\mathcal{M}$. Let ν_Ψ denote the marginal distribution of μ_Ψ on $\mathcal{M}$. Since $X_{t-1}(\mathcal{M} \setminus \{u\})$ and σ_Λ are both supported on subsets of $\mathcal{M}$, the above gives

$$\nu_u^{\sigma_\Lambda} = (\nu_\Psi)_u^{\sigma_\Lambda} = (\nu_\Psi)_u^{X_{t-1}(\mathcal{M}\setminus\{u\})} = \nu_u^{X_{t-1}(\mathcal{M}\setminus\{u\})}.$$

Recall that we wish to sample from $\nu_u^{\mathrm{pad},X_{t-1}(\mathcal{M}\setminus\{u\})}$. Observe that for any partial assignment σ, $\nu_u^{\mathrm{pad},\sigma}$ is a deterministic function of ν_u^σ (see Definition 19). Since $\nu_u^{X_{t-1}(\mathcal{M}\setminus\{u\})} = (\nu_\Psi)_u^{\sigma_\Lambda}$, we have $\nu_v^{\mathrm{pad},X_{t-1}(\mathcal{M}\setminus\{u\})} = (\nu_\Psi)_v^{\mathrm{pad},\sigma_\Lambda}$ as well. Thus, it suffices to

sample $c \sim (\nu_\Psi)_u^{\mathrm{pad},\sigma_\Lambda} = \nu_u^{\mathrm{pad},X_{t-1}(\mathcal{M}\setminus\{u\})}$ which was performed in Line 19. This shows that Lines 9–19 draws $X_t(u)$ from the padding distribution $\nu_u^{\mathrm{pad},X_{t-1}(\mathcal{M}\setminus\{u\})}$, and finishes the proof. ◀

We now show that the condition $|R| < 80dk^4 \log n$ happens with high probability. To do this, we show that in a single instance of $\mathsf{Glauber}(t)$, the "chain" of further recursions $\mathsf{Glauber}(t')$ is unlikely to be large. We build the following graph G to track these recursions.

▶ **Definition 26.** *For every $C \in \mathcal{C}$ and $t \le 0$, let*

$$\phi(C,t) := (\mathrm{vbl}(C), \{\mathsf{pred}_w(t) : w \in \mathrm{vbl}(C) \cap \mathcal{M}\}),$$

i.e., $\phi(C,t)$ is the pair comprising the variables of C and the most recent times that any marked variable in C was resampled up until time t. Consider an associated graph G_t defined by

$$V(G_t) = \{\phi(C,t') : C \in \mathcal{C},\ -T \le t' \le t\},$$

such that $\phi(C_1,t_1) \sim \phi(C_2,t_2)$ in G_t if and only if the following holds:

1. $\mathrm{vbl}(C_1) \cap \mathrm{vbl}(C_2) \neq \emptyset$,
2. *For $\mathcal{T} := \{\mathsf{pred}_w(t_1) : w \in \mathrm{vbl}(C_1) \cap \mathcal{M}\} \cup \{\mathsf{pred}_w(t_2) : w \in \mathrm{vbl}(C_2) \cap \mathcal{M}\}$, we have $\max \mathcal{T} - \min \mathcal{T} < 2m$ (recall that $m = |\mathcal{M}|$).*

We also recall the notion of a 2-tree in a graph or hypergraph.

▶ **Definition 27.** *Let $G = (V(G), E(G))$ be a graph or hypergraph. We say that $Z \subseteq V(G)$ is a 2-tree if Z satisfies the following conditions:*

1. *for all $u, v \in Z$, $\mathrm{dist}_G(u,v) \ge 2$;*
2. *the associated graph with vertex set Z and edge set $\{\{u,v\} \subseteq Z : \mathrm{dist}_G(u,v) = 2\}$ is connected.*

There are not many 2-trees containing a fixed vertex in a graph of bounded maximum degree. We recall one example upper bound below that will be sufficient for our purposes.

▶ **Observation 28** (see [10, Corollary 5.7])**.** *Let $G = (V(G), E(G))$ be a graph or hypergraph with maximum degree D. Then for every $v \in V$, the number of 2-trees $Z \subseteq V(G)$ containing v with $|Z| = \ell$ is at most $\frac{(eD^2)^{\ell-1}}{2}$.*

The following proposition shows that the size of R is unlikely to be large when we terminate $\mathsf{Glauber}(t)$ for any $t \le 0$.

▶ **Proposition 29.** *Fix $t \le 0$. Suppose $\theta := 1 - \frac{1}{2}\exp\left(\frac{2edk}{2^{\alpha k}}\right) \ge 0.4$ and $36ed^3k^4 \cdot 0.6^{\alpha k} \le 1/2$. Upon the termination of $\mathsf{Glauber}(t)$, for every $\eta \ge 1$, the size of R satisfies*

$$\mathbb{P}[|R| \ge 24dk^4(\eta+1)] \le 2^{-\eta}.$$

Proof. Fix $t \le 0$ and consider some instance of $\mathsf{Glauber}(t)$. For $t_1 < t_0 \le t$, we say that $\mathsf{Glauber}(t_1)$ is *triggered* by $x := \phi(C,t_0)$ if $\mathsf{Glauber}(t_1)$ is called in Line 14 of $\mathsf{Glauber}(t_0)$ with clause C. Let $W = \{x \in V(G_t) : x \text{ triggers recursive calls}\}$. We begin by verifying a few basic properties that R and $G := G_t$ enjoy.

▷ Claim 30. Upon the termination of $\mathsf{Glauber}(t)$, we have $|R| \le 2k^2|W| + 2k$.

Proof. Observe that for every $t_0 \le t$, $\mathsf{Glauber}(t_0)$ calls $\mathsf{LB\text{-}Sample}$ at most $k+1$ times (in Line 7 and Line 12 of Algorithm 5) before possibly going into another subroutine $\mathsf{Glauber}(t_1)$. Let A denote the set of timestamps t_0 such that $\mathsf{Glauber}(t_0)$ was called at least once. Then we have $|R| \le (k+1)|A|$.

Observe that every $\mathsf{Glauber}(t_0)$ with $t_0 < t$ is triggered by some $x \in W$. Moreover, every $x \in W$ triggers at most k subroutines $\mathsf{Glauber}(t_0)$ in Line 14. Thus, we get that $|A| \le k|W| + 1$, which gives $|R| \le (k+1)|A| \le (k^2+k)|W| + k + 1 \le 2k^2|W| + 2k$. ◁

▷ Claim 31. The maximum degree of G is at most $6k^2d - 1$.

Proof. Fix $\phi(C_1, t_1) \in W$. There are at most kd clauses $C_2 \in \mathcal{C}$ such that $\mathrm{vbl}(C_1) \cap \mathrm{vbl}(C_2) \neq \emptyset$. For any such C_2, we count the number of possible $\phi(C_2, t_2)$ so that $\mathcal{T} = \{\mathsf{pred}_w(t_1) : w \in \mathrm{vbl}(C_1) \cap \mathcal{M}\} \cup \{\mathsf{pred}_w(t_2) : w \in \mathrm{vbl}(C_2) \cap \mathcal{M}\}$ satisfies $\max \mathcal{T} - \min \mathcal{T} < 2m$.

Suppose

$$\{\mathsf{pred}_w(t_1) : w \in \mathrm{vbl}(C_1) \cap \mathcal{M}\} = \{s_1, \dots, s_{k_1}\} \text{ with } s_1 < \cdots < s_{k_1},$$
$$\{\mathsf{pred}_w(t_2) : w \in \mathrm{vbl}(C_2) \cap \mathcal{M}\} = \{s'_1, \dots, s'_{k_2}\} \text{ with } s'_1 < \cdots < s'_{k_2}.$$

Observe that $s_{k_1} \le t_1 < s_1 + m$ and $s'_{k_2} \le t_2 < s'_1 + m$. If $\max \mathcal{T} - \min \mathcal{T} < 2m$, then we have

$$s_{k_1} - 2m < s'_1 < s'_{k_2} < s_1 + 2m,$$

which gives $t_2 < s'_1 + m < s_1 + 3m$ and $t_2 \ge s'_{k_2} > s_{k_1} - 2m$. Thus, we have

$$s_{k_1} - 2m < t_2 < s_1 + 3m.$$

Let $S := \{s_{k_1} - 2m + 1, s_{k_1} - 2m + 2, \dots, s_1 + 3m - 1\}$. In particular, S is an interval of size $\le 5m$ given by $\phi(C_1, t_1)$.

Observe that as t_2 increments from $s_{k_1} - 2m + 1$ to $s_1 + 3m - 1$, we have $\{\mathsf{pred}_w(t_2) : w \in \mathrm{vbl}(C_2) \cap \mathcal{M}\} \neq \{\mathsf{pred}_w(t_2 - 1) : w \in \mathrm{vbl}(C_2) \cap \mathcal{M}\}$ only if $\mathsf{pred}_w(t_2) > \mathsf{pred}_w(t_2 - 1)$ for some $w \in \mathrm{vbl}(C_2) \cap \mathcal{M}$. Moreover, since $|\mathrm{vbl}(C_2) \cap \mathcal{M}| \le k$ and $|S| \le 5m$, we know that there are at most $5k$ such numbers t_2 in S, and these numbers have been completely determined by $\mathrm{vbl}(C_2)$ and S (i.e., by $\phi(C_1, t_1)$ and C_2). These numbers partition S into at most $5k + 1$ intervals such that $\{\mathsf{pred}_w(t_2) : w \in \mathrm{vbl}(C_2) \cap \mathcal{M}\}$ is the same for all t_2 on each interval. Thus, for every fixed $\phi(C_1, t_1)$ and C_2, the set $\{\{\mathsf{pred}_w(t_2) : w \in \mathrm{vbl}(C_2) \cap \mathcal{M}\} : t_2 \in S\}$ has size at most $5k + 1$.

Therefore, given any $\phi(C_1, t_1)$, we can pick a neighbor $\phi(C_2, t_2) \sim \phi(C_1, t_1)$ by first picking C_2 (which has $\le kd$ choices) and then picking an element in $\{\{\mathsf{pred}_w(t_2) : w \in \mathrm{vbl}(C_2) \cap \mathcal{M}\} : t_2 \in S\}$ (which has $\le 5k + 1$ choices). So the number of possible $\phi(C_2, t_2) \sim \phi(C_1, t_1)$ in G is at most $kd(5k + 1) \le 6k^2d - 1$. ◁

▷ Claim 32. Let $u = u_{i(t)}$ and $W' = \{\phi(C, t) \in W : u \in \mathrm{vbl}(C)\}$. Then $G[W']$ is a clique.

Proof. Consider any two clauses C, C' such that $u \in \mathrm{vbl}(C) \cap \mathrm{vbl}(C')$. Suppose $\phi(C, t), \phi(C', t) \in W'$. Clearly $\mathrm{vbl}(C) \cap \mathrm{vbl}(C') \neq \emptyset$. Moreover, since the timestamps $\mathsf{pred}_w(t)$ over all marked variables $w \in \mathcal{M}$ lie in the range $\{t - m + 1, \dots, t\}$, we get that the maximum and minimum of $\{\mathsf{pred}_w(t) : w \in \mathrm{vbl}(C) \cap \mathcal{M}\} \cup \{\mathsf{pred}_w(t) : w \in \mathrm{vbl}(C') \cap \mathcal{M}\}$ differ by at most $m - 1 < 2m$. Thus we have $\phi(C, t) \sim \phi(C', t)$ in G. ◁

▷ Claim 33. Let W' be as in Claim 32. For every $x \in W$, there exists a path $p_0 \dots p_\ell$ in $G[W]$ such that $p_0 \in W'$ and $p_\ell = x$.

Proof. Let $x = \phi(C, t_1)$ be any element in W. We perform a double induction, the outside on t_1 and the inside on C.

- **Base case: $t_1 = t$.**
  Suppose first that $t_1 = t$, so $x = \phi(C, t)$ for some C. Let $u = u_{i(t)}$. If $u \in \mathrm{vbl}(C)$, then $x \in W'$ and we are done. Inductively, suppose Claim 33 holds for all $x' = \phi(C', t)$ that triggers a recursive call earlier than x in Algorithm 5. If $u \notin \mathrm{vbl}(C)$, then by the while loop condition Line 10, there must exist some $x' = \phi(C', t) \in W$ such that x' triggers a recursive call earlier than x, and $\mathrm{vbl}(C) \cap \mathrm{vbl}(C') \neq \emptyset$. By the inductive hypothesis, there exists a path $p_0 \dots p_\ell$ in $G[W]$ such that $p_0 \in W'$ and $p_\ell = x'$. Since the maximum and minimum of $\{\mathsf{pred}_w(t) : w \in \mathrm{vbl}(C) \cap \mathcal{M}\} \cup \{\mathsf{pred}_w(t) : w \in \mathrm{vbl}(C') \cap \mathcal{M}\}$ differ by at most $m - 1 < 2m$, we get that $x' \sim x$ in G. Therefore we can extend the path $p_0 \dots p_\ell$ with $p_{\ell+1} = x$. This finishes the inductive step.
- **Inductive step: $t_1 < t$.**
  Now suppose $x = \phi(C, t_1)$ with $t_1 < t$. By the inductive hypothesis, we can assume Claim 33 for all $t_0 \in \{t_1 + 1, \dots, t\}$. Let $u_1 = u_{i(t_1)}$.
  Suppose first that $u_1 \in \mathrm{vbl}(C)$. Since $t_1 < t$, there must exist $t_0 \in \{t_1 + 1, \dots, t\}$ and $C' \in \mathcal{C}$ such that $\phi(C', t_0)$ triggers $\mathsf{Glauber}(t_1)$, with $u_1 \in \mathrm{vbl}(C')$ and $t_1 = \mathsf{pred}_{u_1}(t_0)$. Let $y = \phi(C', t_0)$. Clearly $\mathrm{vbl}(C) \cap \mathrm{vbl}(C') \neq \emptyset$. Since $t_0 - m < t_1 < t_0$, we also know that the maximum and minimum of $\{\mathsf{pred}_w(t_1) : w \in \mathrm{vbl}(C) \cap \mathcal{M}\} \cup \{\mathsf{pred}_w(t_0) : w \in \mathrm{vbl}(C') \cap \mathcal{M}\}$ differ by at most $2m - 1 < 2m$. Thus we have $x \sim y$ in G. By the inductive hypothesis for t_0, there exists a path $p_0 \dots p_\ell$ in $G[W]$ such that $p_0 \in W'$ and $p_\ell = y$. Since $x \sim y$ in G, we can extend the path by $p_{\ell+1} = x$.
  Inductively, suppose $u_1 \notin \mathrm{vbl}(C_1)$, and Claim 33 holds for all $t_0 \in \{t_1 + 1, \dots, t\}$ and for all $x' = (C', t_1)$ that triggers a recursive call earlier than x. Then there must exist some $x' = \phi(C', t_1) \in W$ such that x' triggers a recursive call earlier than x, and $\mathrm{vbl}(C) \cap \mathrm{vbl}(C') \neq \emptyset$. By the inductive hypothesis, there exists a path $p_0 \dots p_\ell \in G[W]$ such that $p_0 \in W'$ and $p_\ell = x'$. Since the maximum and minimum of $\{\mathsf{pred}_w(t_1) : w \in \mathrm{vbl}(C) \cap \mathcal{M}\} \cup \{\mathsf{pred}_w(t_1) : w \in \mathrm{vbl}(C') \cap \mathcal{M}\}$ differ by at most $m - 1 < 2m$, we get that $x' \sim x$ in G. Thus we can extend the path by $p_{\ell+1} = x$. This finishes the inductive step.

◁

▷ Claim 34. For all $x = \phi(C, t') \in W$ and $w \in \mathrm{vbl}(C) \cap \mathcal{M}$, the function $\mathsf{LB\text{-}Sample}(\mathsf{pred}_w(t'))$ does not satisfy C.

Proof. This directly follows from Lines 12–14 of Algorithm 5. Since $\phi(C, t')$ triggers a recursion in Line 14, by the condition in Line 12, for all marked variables $w \in \mathrm{vbl}(C) \cap \mathcal{M}$, the function $\mathsf{LB\text{-}Sample}(\mathsf{pred}_w(t'))$ does not satisfy C. ◁

With these claims, we can now prove the proposition. Fix $t \leq 0$ and $\eta \geq 1$. Assume $|R| \geq 24dk^4(\eta + 1)$, which by Claim 30 implies that $|W| \geq 6dk^2(\eta + 1)$. By Claim 33, we know that $W \cap W' \neq \emptyset$; by Claim 31, G has maximum degree $\leq 6dk^2 - 1$. Thus, by a greedy selection, we can find a 2-tree $Z \subseteq W$ containing some element in W' such that $|Z| = \eta + 1$.

Fix any 2-tree $Z \subseteq V(G)$ such that $|Z| = \eta + 1$. For every $x = \phi(C, t') \in Z$, if $x \in W$, then we know from Claim 34 that $\mathsf{LB\text{-}Sample}(\mathsf{pred}_w(t'))$ does not satisfy C for all $w \in \mathrm{vbl}(C) \cap \mathcal{M}$. Since the latter happens with probability at most $(1 - \theta)^{\alpha k}$, we have

$$\mathbb{P}[x \in W] \leq (1 - \theta)^{\alpha k}.$$

Since Z is a 2-tree, we know that for every two $x = \phi(C_1, t_1), y = \phi(C_2, t_2) \in Z$, we have $\{\mathsf{pred}_w(t_1) : w \in \mathrm{vbl}(C_1) \cap \mathcal{M}\} \cap \{\mathsf{pred}_w(t_2) : w \in \mathrm{vbl}(C_2) \cap \mathcal{M}\} = \emptyset$ (as otherwise C_1 and C_2 would share a variable, and the union of these two sets would span an interval of size at most $2(m-1)+1 = 2m-1$, meaning that $x \sim y$ in G, which contradicts the fact that Z is an independent set in G). In particular, the sets

$$\{\{\mathsf{pred}_w(t') : w \in \mathrm{vbl}(C) \cap \mathcal{M}\} : \phi(C, t') \in Z\}$$

are disjoint. Thus, for every fixed 2-tree $|Z| = \eta + 1$ in $V(G)$, we have

$$\mathbb{P}[Z \subseteq W] \leq \mathbb{P}[x \in W \text{ for all } x \in Z] \leq (1-\theta)^{\alpha k \eta}.$$

Let $\mathcal{T}$ denote the set of 2-trees of $V(G)$ of size $\eta + 1$ that intersects with W'. Since $|W'| \leq d$ and G has maximum degree at most $6k^2d - 1$, by Observation 28, we have

$$|\mathcal{T}| \leq d \cdot \frac{(e(6k^2d)^2)^\eta}{2} \leq (36ed^3k^4)^\eta.$$

Thus, we have

$$\mathbb{P}[|W| \geq 6dk^2(\eta+1)] \leq \sum_{Z \in \mathcal{T}} \mathbb{P}[Z \subseteq W] \leq (36ed^3k^4)^\eta (1-\theta)^{\alpha k \eta} \leq 2^{-\eta},$$

where the last step used the assumption that $\theta \geq 0.4$ and $36ed^3k^4 \cdot 0.6^{\alpha k} \leq 1/2$. This implies that

$$\mathbb{P}[|R| \geq 24dk^4(\eta+1)] \leq \mathbb{P}[|W| \geq 6dk^2(\eta+1)] \leq 2^{-\eta}. \qquad \blacktriangleleft$$

Setting $\eta = (3+c)\log n$, we get the following correctness statement on Algorithm 5.

▶ **Corollary 35.** *For every $t \leq 0$, we have*

$$\mathbb{P}[\mathsf{Glauber}(t) \neq X_t(u)] \overset{25}{\leq} \mathbb{P}[|R| > 80dk^4 \log n] \leq \mathbb{P}[|R| > 24dk^4((3+c)\log n + 1)] \overset{29}{\leq} n^{-(3+c)}.$$

In particular, for every $u \in \mathcal{M}$, we have

$$\mathbb{P}[\mathsf{Glauber}(\mathsf{pred}_u(0)) \neq X_0(u)] = \mathbb{P}[\mathsf{Glauber}(\mathsf{pred}_u(0)) \neq X_{\mathsf{pred}_u(0)}(u)] \leq n^{-(3+c)}.$$

Taking a union bound over all $u \in \mathcal{M}$, we get that

$$\mathbb{P}[(\mathsf{MarginSample}(u))_{u \in \mathcal{M}} \neq X_0] \leq n^{-(2+c)}.$$

Since we have picked T in Algorithm 3 sufficiently large so that $d_{\mathsf{TV}}(X_0, \nu) \leq n^{-(2+c)}$, we get that the joint output $(\mathsf{MarginSample}(u))_{u \in \mathcal{M}}$ satisfies $d_{\mathsf{TV}}((\mathsf{MarginSample}(u))_{u \in \mathcal{M}}, \nu) \leq n^{-c}$, proving Proposition 20.

B Efficiency of Algorithm 5

We now move on to show the efficiency of $\mathsf{Glauber}(t)$ for all $t \leq 0$. Observe that for every $r \geq 48dk^4$, by Proposition 29, we have

$$\mathbb{P}[|R| \geq r] \leq 2^{-\frac{r}{48dk^4}}. \tag{4}$$

Moreover, we terminate $\mathsf{Glauber}(t)$ once we reach $|R| \geq 80dk^4 \log n$. We will use these information to give an upper bound on the expected cost of $\mathsf{Glauber}(t)$.

We start by upper bounding the size of the final sets V' and Λ in Algorithm 5 in terms of $|R|$.

▶ **Lemma 36.** *The V' and Λ in Line 19 of Algorithm 5 satisfy*

$$|\Lambda| \le kd|V'| \le 2kd^2|R|/\alpha.$$

Proof. By Line 10 of Algorithm 5, we know that for all $u \in \Lambda$, there exists clause $C \in \mathcal{C}$ such that $u \in \mathrm{vbl}(C)$ and $\mathrm{vbl}(C) \cap V' \neq \emptyset$. This shows that $|\Lambda| \le kd|V'|$. Observe that V' contains at least $|V'|/(\Delta+1) \ge |V'|/(2dk)$ clauses with disjoint variable sets. Since every marked variable in the clauses added to V' have gone through at least one round of LB-Sample (per Line 5 and Line 12), we get that $|R| \ge \alpha k \cdot |V'|/(2dk) = \alpha|V'|/(2d)$. ◀

To upper bound the expected cost of Line 19, we further need the result of He–Wang–Yin [18] on the existence of a "Bernoulli factory" algorithm $\mathsf{BF}(\cdot)$ such that for every locally uniform CNF and variable v, the Bernoulli factory efficiently draws a $\{0,1\}$-random variable according to the padding distribution of v.

▶ **Proposition 37** ([18, Lemma 3.10 and Appendix A])**.** *Let $\Psi = (V_\Psi, \mathcal{C}_\Psi)$ be a k-CNF, σ be a feasible partial assignment of Ψ, and $\Psi^\sigma = (V_\Psi^\sigma, \mathcal{C}_\Psi^\sigma)$ be the reduced CNF (see Definition 5). Suppose we have*

$$\mathbb{P}[\neg C \mid \sigma] \le \zeta \qquad \text{for every } C \in \mathcal{C}_\Psi^\sigma.$$

Then there exists an algorithm $\mathsf{BF}(\cdot)$ such that for every $v \in V^\sigma$,

- *$\mathsf{BF}(v)$ with probability 1 returns $\xi \sim (\mu_\Psi)_v^{\mathrm{pad},\sigma}$;*
- *$\mathsf{BF}(v)$ has expected cost $O(k^9 d^6 |\mathcal{C}_\Psi^\sigma|(1-e\zeta)^{-|\mathcal{C}_\Psi^\sigma|})$.*

We remark that in Algorithm 5, our partial assignment σ_Λ is always supported on the marked variables. Since every clause has at least αk unmarked variables that are not assigned by σ_Λ, we get that $\mathbb{P}[\neg C \mid \sigma_\Lambda] \le 2^{-\alpha k}$ for every C. Thus when applying Proposition 37, we will take $\zeta = 2^{-\alpha k}$.

We can now give an upper bound on the expected cost of Algorithm 5, proving Lemma 21.

Proof of Lemma 21. Let A be the set of all t' such that $\mathsf{Glauber}(t')$ is executed at least once as subroutine of $\mathsf{Glauber}(t)$. Since we run $\mathsf{LB\text{-}Sample}(t')$ at the beginning (Line 7) of each round $\mathsf{Glauber}(t')$, we know that $|A| \le |R|$.

Observe that for every t', if $\mathsf{Glauber}(t')$ has been computed once, then we will permanently assign $M(t') = \mathsf{Glauber}(t') \in \{0,1\}$, and later executions of $\mathsf{Glauber}(t')$ will terminate at Line 4. Thus, it suffices to upper bound the cost of every first execution of $\mathsf{Glauber}(t')$ before entering another $\mathsf{Glauber}(t'')$. Multiplying this by $|A|$ would give an upper bound on the cost of $\mathsf{Glauber}(t)$.

Suppose we are at the first time of executing $\mathsf{Glauber}(t')$ for some t'. We first estimate the cost of the while loop in Lines 10–18, which is a constant multiple of the number of executions of Line 14 and Line 18. Note that every time we execute Line 14 or Line 18, some w is added to Λ due to a clause C chosen in Line 11, with $w \in \mathrm{vbl}(C)$. Moreover, each clause C can be chosen in Line 11 at most once. Thus, the number of pairs (w, C) that correspond to an execution of Line 14 or Line 18 is at most $d|\Lambda|$. Consequently, the cost of the while loop in Lines 10–18 is $O(d|\Lambda|) = O(kd^3|R|/\alpha)$.

We now estimate the cost of Line 19. Observe that in Line 19, we can apply the Bernoulli factory in Proposition 37 to compute the padding distribution $(\nu_\Psi)_u^{\mathrm{pad},\sigma_\Lambda}$. Since the connected component Ψ in Line 19 has at most $d|V'| \le 2d^2|R|/\alpha$ clauses, by Proposition 37, the expected cost of Line 19 is at most

$$O(k^9 d^8 |R|/\alpha (1 - e2^{-\alpha k})^{-2d^2|R|/\alpha}).$$

Let $R_{\max} = 80dk^4 \log n$. Combining the above and applying Equation (4), we get that the expected cost of $\mathsf{Glauber}(t)$ is at most

$$\begin{aligned}
&\mathbb{E}[|A| \cdot O(kd^3|R|/\alpha + k^9d^8|R|/\alpha(1-e2^{-\alpha k})^{-2d^2|R|/\alpha})] \\
&\leq \mathbb{E}[|R| \cdot O(kd^3|R|/\alpha + k^9d^8|R|/\alpha(1-e2^{-\alpha k})^{-2d^2|R|/\alpha})] \\
&\leq \sum_{r=0}^{R_{\max}} \mathbb{P}[|R| \geq r] \cdot r \cdot O(kd^3 r/\alpha + k^9d^8 r/\alpha(1-e2^{-\alpha k})^{-2d^2 r/\alpha}) \\
&\leq \sum_{r=0}^{R_{\max}} O(2^{-\frac{r}{48dk^4}} \cdot r \cdot kd^3 r/\alpha + k^9d^8 r/\alpha(1-e2^{-\alpha k})^{-2d^2 r/\alpha}) \\
&\leq \sum_{r=0}^{R_{\max}} O\left(2^{-\frac{r}{48dk^4}} \cdot r \cdot \left(\frac{kd^3 r}{\alpha} + \frac{k^9d^8 r}{\alpha} \cdot e^{\frac{2d^2 r/\alpha}{2^{\alpha k}}}\right)\right) = O(k^9d^8R_{\max}^2/\alpha) = O(k^{17}d^{10}\log^2 n/\alpha),
\end{aligned}$$

where in the last step we used the information that

$$2^{-\frac{1}{48dk^4}} \cdot e^{\frac{2d^2/\alpha}{2^{(1-\alpha)k}}} \leq 0.9 \quad \Longrightarrow \quad \sum_{r=0}^{R_{\max}} \left(2^{-\frac{1}{48dk^4}} \cdot e^{\frac{2d^2/\alpha}{2^{\alpha k}}}\right)^r = O(1). \qquad \blacktriangleleft$$

Learn to Unlearn

Bernhard Gstrein
University of Freiburg, Germany

Florian Pollitt
University of Freiburg, Germany

André Schidler
University of Freiburg, Germany

Mathias Fleury
University of Freiburg, Germany

Armin Biere
University of Freiburg, Germany

Abstract

Clause learning is a significant milestone in the development of SAT solving. However, keeping all learned clauses without discrimination gradually slows down the solver. Thus, selectively removing some learned clauses during routine database reduction is essential. In this paper, we reexamine and test several long-standing ideas for clause removal in the modern solver Kissat. Our experiments show that retaining all clauses alters performance in all instances. For satisfiable instances, periodically removing all learned clauses surprisingly yields near state-of-the-art performance. For unsatisfiable instances, it is vital to always keep some learned clauses. Building on the influential Glucose paper, we find that it is crucial to always retain the clauses most likely to help, regardless of whether they are ranked by size or LBD in practice. Another key factor is whether a clause was used recently during conflict resolution steps. Eagerly keeping used clauses improves all unlearning strategies.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Satisfiability solving, learned clause recycling, LBD

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.14

Supplementary Material *Software (Source Code)*: `https://github.com/texmex76/kissat-cr` [16] archived at `swh:1:dir:b6a189824ab8c4ceab3f1a2f31c349423724cc19`
Dataset (Log Files): `https://doi.org/10.5281/zenodo.15146408` [17]

Funding This work was supported by the state of Baden-Württemberg through bwHPC, the German Research Foundation (DFG) grant INST 35/1597-1 FUGG, and a gift from Intel Corp.

1 Introduction

Modern SAT solvers following the conflict-driven clause learning (CDCL) paradigm [21] or related algorithms [28], reduce the search space by learning clauses that boost propagation power. Thanks to solver efficiency and hardware advances, they can now learn many more clauses and solve much larger instances. However, these clauses (also called "no-goods" [28]) gradually slow down the solver because the efficiency of most algorithms depends on the size of their data structures. Furthermore, retaining every clause turns CDCL into an algorithm requiring exponential space, similar to reverting from DPLL [6] to DP [7]. Thus, reducing the number of clauses by *unlearning* is crucial for good performance. In this paper, we revisit established clause unlearning strategies and evaluate their impact on our state-of-the-art SAT solver Kissat [4], winner of all tracks of the SAT Competition 2024.


© Bernhard Gstrein, Florian Pollitt, André Schidler, Mathias Fleury, and Armin Biere;
licensed under Creative Commons License CC-BY 4.0



Several research efforts have tried to mitigate this problem. Already in 1993, in parallel to the seminal work on CDCL [24], Ginsberg [12] presented a polynomial space variant of conflict-driven no-good learning, which he later adapted in 2015 to allow arbitrary restarts [13] without losing completeness. However, state-of-the-art solvers have not adopted this approach.

On the practical side, various unlearning schemes have been proposed: keeping clauses up to a fixed size [8], using relevance-based criteria [20], combining different approaches [14,18,24] and more. In our experiments, we revisit size-based unlearning in particular. More recently, CDCL solvers have started to predict the *future* usefulness of a clause by measuring either how often it is used (its activity) [9,14] or how recently it has been used (its age) [11,14,15] during conflict analysis. This information is then used to rank clauses for retention or deletion. The seminal Glucose [1] introduced the new ranking scheme *literal block distance* (LBD) and, more importantly, the idea of keeping specific (glue) clauses unconditionally.

We highlight the importance of clause unlearning in our first experiment (Section 3), where we evaluate Kissat without any unlearning (a configuration previously studied by Chanseok Oh [23]) as well as periodically unlearning all learned clauses. The experiment shows that keeping all clauses degrades performance on all instance types. Interestingly, while periodic unlearning of all clauses yields nearly state-of-the-art performance on satisfiable instances, it significantly harms performance on unsatisfiable ones.

Our second experiment (Section 4) revisits the favored approach of MiniSat [9], where clause *activity* is used to decide which clauses to unlearn. In this scheme, a clause's activity increases each time it is used to derive a new clause, while unused clauses are penalized. In our experiments, the performance of Kissat with activity-based unlearning is unsatisfactory.

In 2009, Glucose [1] introduced a new usefulness criterion based on the propagation power of clauses, approximated by LBD (the number of different decision levels in a clause when it is learned or propagates). While MiniSat ranks clauses based on activity and unlearns a fixed percentage, Glucose sorts clauses into two tiers and unconditionally keeps the lower-tier (critical) clauses. Our experiments in Section 5 confirm that retaining only critical clauses is more important than the difference between using clause size or LBD as the ranking metric. Similar findings regarding the indifference between ranking schemes have also been observed in a related experiment [10] focusing on combinatorial benchmarks.

In 2019, a machine-learning-based approach was used to analyze clause usefulness, which we revisit in Section 6. In that work, the usefulness of a clause was estimated by generating proofs, reconstructing dependencies, and classifying core clauses as useful. Soos et al. [27] identified, that clauses which have been *used* to derive a new clause since the last unlearning round should be kept. This effectively reduces clause activity to a single binary indicator. While this metric alone performs poorly, combining it with other criteria consistently improves solver performance. In fact, such a combination is one of our best configurations and achieves performance similar to the base version of Kissat while being much easier to implement.

Finally, in Section 7, we discuss the state-of-the-art and Kissat. Since Kissat uses LBD as a key metric for estimating a clause's usefulness, we analyze the actual distribution of LBDs in used clauses. While the distribution usually shows peaks that match our definition of critical clauses, some instances have a minimal LBD far above our critical threshold. We address this by proposing a new dynamic tier system with adaptive thresholds to identify critical clauses, though this scheme does not improve performance.

Algorithm 1 Unlearning is part of the *unlearn* function, scheduled regularly within the CDCL loop after a certain number of conflicts occurred and clauses have been learned. We want to understand which clauses in *unlearn* should be thrown-away, i.e., "unlearned", and which should be kept. More aspects on the KISSAT implementation of *unlearn* (in `reduce.c`) are discussed in appendix A.

```
unlearn (some sub-set of the learned clauses F)
1  backtrack to decision level zero (root-level)          // make sure all clauses can be deleted
2  optionally update critical tier limits dynamically      // Sect. 7
3  initialize G as an empty list of candidate clauses to unlearn
4  for all learned clauses C ∈ F
5      check the used flag of C, i.e., whether C was used during learning since the last unlearn
6      if used continue but reset used flag of C in any case      // Sect. 6
7      if C is considered critical continue                       // Sect. 5 (2-tier system)
8      otherwise add C to candidate set G
9  sort G by "predicted usefulness" ranking function          // Sect. 4 (activity), Sect. 5 (LBD)
10 remove from F the least useful fraction f · |G| of clauses in G      // Sect. 3
       // where f is the target fraction to remove, e.g., f = 50% removes half of G,
       // f = 0% no clause (no-unlearn) and f = 100% all remaining candidates (unlearn-all)
11 Schedule next unlearning
```

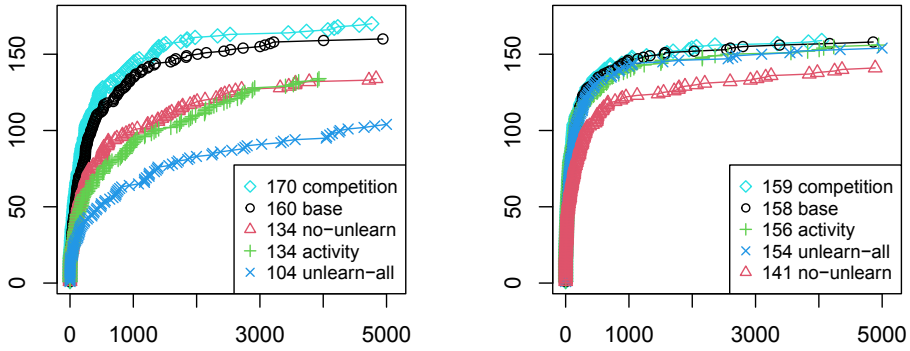
2 Background and Experimental Setup

We refer the reader to the corresponding chapter of the *Handbook of satisfiability* [21] for an introduction to conflict-driven clause learning (CDCL), i.e., CDCL applies unit propagation after decisions, finds conflicts and then learns new clauses to change the focus of the search, interleaved with restarts, inprocessing [19], as well as unlearning, the focus of this paper.

In Alg. 1 we show pseudo-code for the implementation of *unlearn*, which is called periodically from the CDCL loop according to some unlearning schedule. Clauses are only removed during unlearning, so even unlearn-all keeps clauses for some time: it only throws away all learned clauses in each unlearning round. In this paper, we vary different parts of this Alg. 1. Changes and their effects are outlined in the respective sections below.

As initial starting point of our investigations we used the competition version of the award-winning SAT solver KISSAT [4] from the SAT Competition 2024. It won all main-tracks of this most recent competition and thus is considered the state-of-the-art. In order to reduce influence on and by other heuristics while varying aspects of unlearning, we have simplified the competition version in two ways. First, we let KISSAT restart before unlearning clauses, as otherwise, the solver is forced to keep clauses which are used as reasons for assigned variables. Second, we fix the fraction of removed clauses to be the same for the entire run of the solver, specifically 75%. In the competition version it slowly increases from 60% to 90%. The resulting base version is the actual starting point for the other variants considered.

Throughout this paper, all experiments are conducted with the 400 problems from the SAT Competition 2024, following *The SAT Practitioner's Manifesto* [5] and solved on the bwForCluster Helix with AMD Milan EPYC 7513 CPUs, a memory limit of 16 GB per solver run and a time limit of 5000 s. We consistently present results as cumulative distribution functions (CDFs), where the x-axis gives the time and the y-axis shows the number of solved problems (400 total for the entire benchmark set or 200 total when only UNSAT or SAT).



(a) UNSAT instances only. **(b)** SAT instances only.

Figure 2 Comparing unlearning strategies: full competition; simplified base; no-unlearn (retain all clauses); unlearn-all (periodically unlearn all clauses); activity-based (*cf.* Sect. 4).

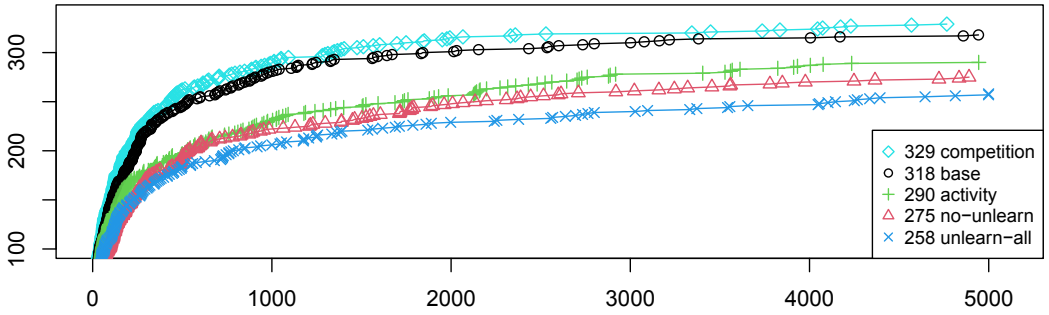


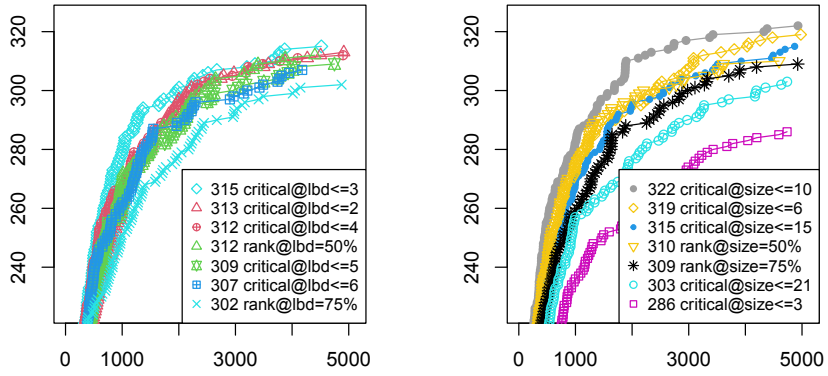
Figure 3 Comparing the same unlearning strategies as in Fig. 2 but on all competition instances.

3 Importance of Unlearning

We first evaluate the configurations unlearn-all and no-unlearn which periodically unlearn all and never unlearn any clauses, respectively. We observe an interesting effect: unlearning all clauses is only very modestly harmful on satisfiable instances (Fig. 2b). It appears sufficient to guess a model to solve those instances and learned clauses only seem to be instrumental to locally guide the solver towards this model, and should be unlearned fast. However, unlearn-all is detrimental to solving unsatisfiable instances (Fig. 2a).

Keeping all clauses (no-unlearn) reduces KISSAT's performance for both classes, but is not as harmful on unsatisfiable instances as expected. It further has a negative impact on memory usage of course as for no-unlearn accumulated memory usage goes from 167 GB to 240 GB. On the other hand, the number of conflicts on the 96 shared solved unsatisfiable instances of the no-unlearn configuration is reduced to 0.7 the number of conflicts in the base configuration and goes up for unlearn-all by more than factor of 10. Chanseok Oh [23] reached similar conclusions for unlearn-all.

In general, we obverse that configurations retaining more clauses perform better on unsatisfiable instances and worse on satisfiable instances, while unlearning more clauses tends to improve performance on satisfiable instances but hinders solving unsatisfiable instances. This might be exploited if the expected result for a particular instance is known.



(a) LBD **(b)** size

Figure 4 Comparing critical thresholds with rank (using two target fractions f for lbd and size).

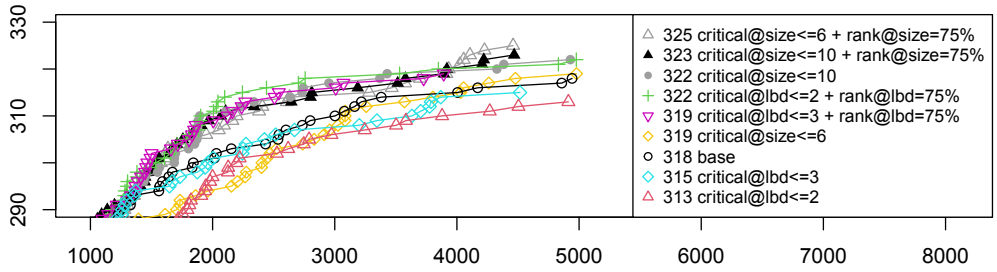


Figure 5 Effect of combining critical and rank with the best configurations from Fig. 4.

4 Activity as Usefulness Prediction

The solver MiniSat [9] implements a common strategy for clause retention: A clause's *activity* – reflecting how often it contributes to conflict analysis – determines its usefulness. Each time a clause participates in deriving a learned clause, its activity increases; new clauses get the current global clause activity increment. The global clause activity increments grows exponentially with each conflict (penalizing unused clauses), and when values become too large, scores are rescaled. This is akin to the (E)VSIDS heuristic [9, 22] for variables. Although alternative metrics like propagation counts have been explored, activity-based heuristics perform best. This aligns with Simon's observation [25] that roughly 90% of propagations are unused, indicating resolved clauses in conflict analysis drive the search.

We implemented an activity branch on top of the base branch (one of 42 for this study [16]) to test how well activity predicts clause usefulness in Kissat. It keeps the aggressive scheduling used in all experiments but modifies the unlearn function (see Alg. 1): the target fraction is set to 50% as in MiniSat, but the *used* flag and critical metrics are ignored.

Figure 3 shows that activity-based unlearning performs worse than our baseline. On satisfiable instances (Fig. 2b) its performance mirrors unlearn-all, on unsatisfiable instances (Fig. 2a) it resembles no-unlearn. Since satisfiable instances gain little from learned clauses, activity-based unlearning gives mixed results and is a weak predictor by modern standards.

Table 6 Number of solved instances for different configurations w/ and w/o the *used* flag.

configuration		+ used	configuration		+ used	configuration		+ used
critical@size<=10	322	**328**(+6)	critical@size<=6	319	**322**(+3)	critical@lbd<=3	315	**324**(+9)
rank@size=50%	310	**322**(+12)	rank@lbd=50%	312	**325**(+13)	activity	290	**294**(+4)

5 Critical Clauses and LBD as Usefulness Prediction

In 2009, Audemard and Simon [1] introduced two key modifications to MiniSat, resulting in Glucose. These include the LBD score and a policy of always retaining high-quality (LBD ≤ 2) clauses – termed *glue clauses* by the authors, but called *critical* here to avoid confusion. These changes enabled more aggressive restarts and unlearning by keeping fewer clauses. This yields a *2-tier system*: top-ranked clauses (tier 1) are always retained; others (tier 2) are ranked and selectively unlearned.

The term "LBD" has two definitions. In [1], it counts the number of decision levels in a learned clause before backjumping. Later work [2, 3] updates LBD upon propagation, based on levels after backjumping (one less than the original). To reconcile this, Glucose adds 1 to updated scores, while Kissat adopts the propagating definition, calling it "glue" in code.

We evaluate LBD and critical clause ideas independently (Fig. 4). One set of experiments compares different critical thresholds (Alg. 1, line 7) across target fractions using rank (line 10); another uses clause size instead. Although the average size/LBD ranged from 2.05 to 2.15, we expected glue<=5 to behave like size<=10 – but it did not.

Still, surprisingly, LBD and size are nearly interchangeable. Retaining clauses up to size 10 yields the best results, while rank configurations trail the best critical ones. This suggests that setting an appropriate critical threshold suffices to identify valuable clauses, whereas rank tends to retain more unhelpful ones, hurting performance.

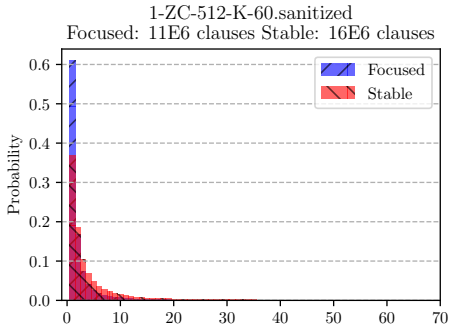
We also evaluated combinations of critical and rank (Fig. 5) by enabling both lines 7 and line 10 in Alg. 1. Since critical clauses are always retained, we reverted to the default unlearn target of $f = 75\%$. This significantly improved performance for lower thresholds, such as lbd<=2, lbd<=3 or size<=6, with smaller gains for the higher size<=10 threshold.

6 Used Clauses are Useful

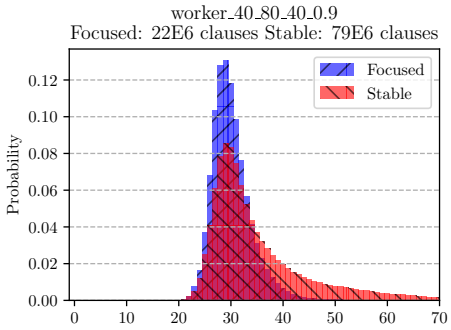
In 2019, Soos et al. [27] and, more recently, Yang et al. [29] applied machine learning techniques to identify effective criteria for clause usefulness. Their approach involved running the solver without performing any clause unlearning while logging detailed statistical data for each learned clause. Subsequently, a proof checker trimmed the proof and extracted resolution chains using a heuristic that favored reusing existing clauses over introducing new ones during derivation. Their analysis confirms that the LBD is an important predictor of a clause's usefulness – with LBD slightly outperforming clause size – and demonstrated that a clause's recent use in deriving a conflict is a strong estimator of its future utility.

Various solvers, including CaDiCaL, implement a *used* flag for clauses, set during conflict analysis and reset during unlearning (*cf.* Alg. 1). In essence, the *used* flag reduces clause activity to a single bit recording whether the clause was used since the last unlearning. It is very similar to the *second chance algorithm* or *clock algorithm* [26] used in page replacement policies, as it gives the clause a second chance to be useful before unlearning.

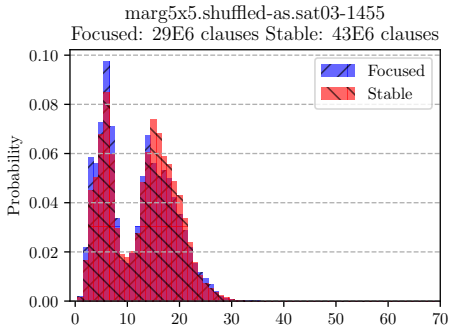
We combine this *used* flag with each of the configurations from the previous sections and compare the difference in the solved instances when retaining *all* used clauses (+ used) versus the configuration without *used* flag (Tab. 6). Subsequently, we refer to these configurations



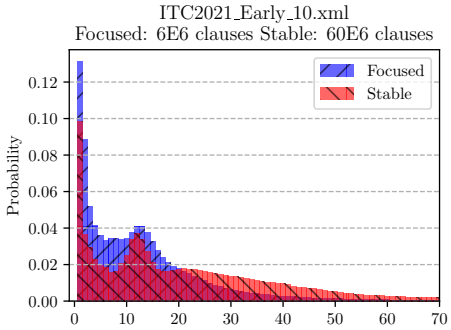
(a) Peak at LBD 1 (common distribution).



(b) Peak offset at LBD 30 (rare distribution).



(c) Two clear peaks (rare distribution).



(d) Most frequent distribution with long tail.

Figure 7 LBD distribution for selected benchmarks in no-reduce (keep all clauses). Benchmark name is shown above each graph together with how many clauses were learned during the run.

as "<*configuration*>+used". Keeping only used clauses performs slightly better than activity-based unlearning, showing that it is indeed a useful abstraction (Fig. 9). However, it still performs worse than the baseline. The full potential of the *used* flag is only realized in conjunction with other unlearning methods: it *consistently* improves each configuration's performance. Hence, used clauses are important, but less so than critical clauses.

The *used* flag inspired us to give clauses one more chance, namely a *third* chance. When clauses are used, we set *used* to 2 and decrement it during unlearning. We consider the clause for deletion only when we reach zero. This in essence simulates a three-tier system [23], but in our experiments does not yield any improvements (see Fig. 9 in the appendix).

Another interesting question is how to update the *used* flag during inprocessing. We recommend *not* updating the *used* flag during forward subsumption, as doing so resulted in significant performance degradation in CaDiCaL. Specifically, in version 1.9.4 we altered the policy for updating the *used* flag on successfully strengthened learned clauses – changing it from granting a second chance to providing a third chance. This modification led to a severe performance regression in incremental solving, and was reverted in CaDiCaL 1.9.5.

7 Odd LBD Distributions and Dynamic Tier Calculations

In order to link LBD to usefulness, we further recorded the LBD of all clauses used during conflict analysis on all benchmarks from the SAT Competition 2024. Note that Kissat runs a portfolio of two modes; *focused* mode (many restarts, shallow exploration of the search

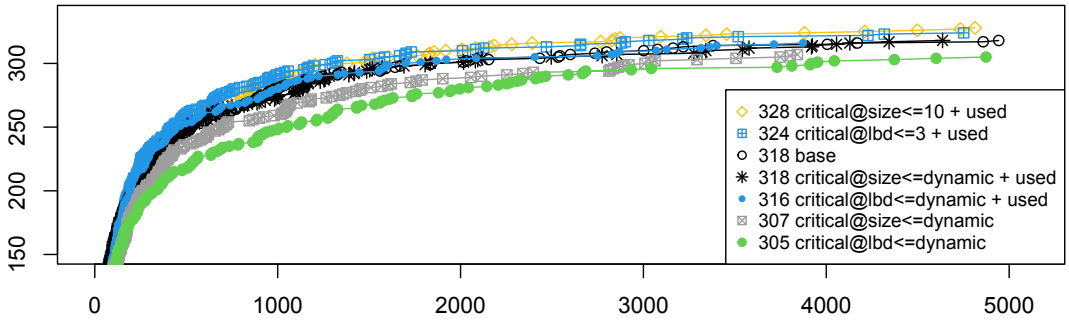


Figure 8 Runs with dynamically calculated tier 1 limit (50% clauses below the limit were used).

space), and *stable* mode (few restarts, and deep exploration of the search space with long assignments). Chanseok Oh [23] calls these modes "UNSAT" and "SAT" modes. Due to these different search characteristics, we recorded LBD statistics for both modes separately.

We show 4 representative instances from the SAT Competition 2024 in Fig. 7. The most common LBD distribution shows a sharp peak at very low LBD values, followed by a steep decline. However, some problem families exhibit a broader distribution, or the peak is shifted to much higher LBD values. In certain cases, there are *no* used clauses with low LBDs at all. As a result, *no* critical clauses would be retained unconditionally. In such situations, unconditionally keeping *used* clauses (and ranking) helps to compensate for this issue.

The skewed LBD distributions motivated us to replace the static tier limit for identifying critical clauses with a dynamic approach based on either LBD or size. Specifically, we set the threshold so that 50% of all clauses used since the beginning fall below this limit.

We also observed some instances where most learned clauses have an LBD of 1. In such cases, using a tier threshold of LBD 2 – typically seen as very good – would actually be problematic, as it would prevent those clauses from ever being unlearned.

The dynamic unlearning scheme (Fig. 8) does not achieve the same performance as the static schemes, suggesting that the usefulness of critical clauses comes from the absolute size or LBD, and not the comparatively low size or LBD. The performance of dynamic configurations is similar to the non-dynamic configurations on satisfiable instances but notably worse on unsatisfiable instances. This, surprisingly, indicates that the in the dynamic configurations too few useful clauses are kept.

The *used* flag of the previous experiment also improves dynamic configurations. Further, size and LBD in combination with dynamic tier limit computation perform similarly. Nonetheless, the dynamic configurations together with the *used* flag still perform worse. While we see as usual that the used flag improves the performance and that size is similar to LBD, the resulting configuration is still worse than the version where we keep all clauses of size 10. We have observed during earlier runs on SAT Competition 2023 bencharks that there were more outliers for the glue distributions compared to SAT Competition 2024 benchmarks. We leave the investigation of this observation as future work.

8 Conclusion and Future Work

We investigated which unlearning schemes are useful for the state-of-the-art SAT solver Kissat on SAT Competition 2024 instances. Our results show that, overall, activity performs the worst. Furthermore, while the difference between LBD and size based ranking is limited, it is important to keep either low LBD or low size clauses unconditionally, as well as *used* clauses, which consistently gives a boost for all our considered unlearning strategies. A

vaild configuration which could replace the current strategy in KissAT could therefore be critical@size<=6 + rank@size=75% + used. Probably the most striking result is that frequently unlearning all clauses completely is in essence not harmful for solving satisfiable instances.

As future work it would be interesting to understand whether these findings can be extended to other SAT solvers and persist on other benchmark sets, as the diversity of solved instances for the strategies is perhaps limited by the benchmark set (the virtual best solves 351 instances, only 22 more than the single best strategy).

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. Refining restarts strategies for SAT and UNSAT. In *CP*, volume 7514 of *Lecture Notes in Computer Science*, pages 118–126. Springer, 2012. doi:10.1007/978-3-642-33558-7_11.

3 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.

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

5 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.

6 Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. doi:10.1145/368273.368557.

7 Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. doi:10.1145/321033.321034.

8 Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cut-set decomposition. *Artificial Intelligence*, 41(3):273–312, 1990. doi:10.1016/0004-3702(90)90046-3.

9 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.

10 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 Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1300–1308. ijcai.org, 2018. doi:10.24963/IJCAI.2018/181.

11 Roman Gershman and Ofer Strichman. Haifasat: a SAT solver based on an abstraction/refinement model. *J. Satisf. Boolean Model. Comput.*, 6(1-3):33–51, 2009. doi:10.3233/SAT190061.

12 Matthew L Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993. doi:10.1613/JAIR.1.

13 Matthew L Ginsberg. Satsisfiability and systematicity. *Journal of Artificial Intelligence Research*, 53:497–540, 2015. doi:10.1613/JAIR.4684.

14 Eugene Goldberg and Yakov Novikov. BerkMin: A fast and robust Sat-solver. *Discret. Appl. Math.*, 155(12):1549–1561, 2007. doi:10.1016/J.DAM.2006.10.007.

15 Evguenii I. Goldberg and Yakov Novikov. Berkmin: A fast and robust Sat-solver. In *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France*, pages 142–149. IEEE Computer Society, 2002. doi:10.1109/DATE.2002.998262.

16 Bernhard Gstrein, Florian Pollitt, André Schidler, Mathias Fleury, and Armin Biere. Kissat Clause Reduction Version. Software, swhId: swh:1:dir:b6a189824ab8c4ceab3f1a2f31c349423724cc19 (visited on 2025-07-16). URL: https://github.com/texmex76/kissat-cr, doi:10.4230/artifacts.24025.

17 Bernhard Gstrein, Florian Pollitt, André Schidler, Mathias Fleury, and Armin Biere. Learn to unlearn, April 2025. doi:10.5281/zenodo.15146408.

18 Said Jabbour, Jerry Lonlac, Lakhdar Sais, and Yakoub Salhi. Revisiting the learned clauses database reduction strategies. *International Journal on Artificial Intelligence Tools*, 27, February 2014. doi:10.1142/S0218213018500331.

19 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.

20 Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In Benjamin Kuipers and Bonnie L. Webber, editors, *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA*, pages 203–208. AAAI Press / The MIT Press, 1997. URL: http://www.aaai.org/Library/AAAI/1997/aaai97-032.php.

21 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*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2nd edition, 2021. doi:10.3233/FAIA200987.

22 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.

23 Chanseok Oh. Between SAT and UNSAT: the fundamental difference in CDCL SAT. 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 *LNCS*, pages 307–323. Springer, 2015. doi:10.1007/978-3-319-24318-4_23.

24 João P. Marques Silva and Karem A. Sakallah. GRASP – A new search algorithm for satisfiability. In Rob A. Rutenbar and Ralph H. J. M. Otten, editors, *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, pages 220–227. IEEE Computer Society / ACM, 1996. doi:10.1109/ICCAD.1996.569607.

25 Laurent Simon. Post mortem analysis of SAT solver proofs. In Daniel Le Berre, editor, *POS-14. Fifth Pragmatics of SAT workshop*, volume 27 of *EPiC Series in Computing*, pages 26–40. EasyChair, 2014. doi:10.29007/gpp8.

26 Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.*, 3(3):223–247, 1978. doi:10.1145/320263.320276.

27 Mate Soos, Raghav Kulkarni, and Kuldeep S. Meel. CrystalBall: Gazing in the black box of SAT solving. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019 – 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 371–387. Springer, 2019. doi:10.1007/978-3-030-24258-9_26.

28 Richard M Stallman and Gerald J Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial intelligence*, 9(2):135–196, 1977. doi:10.1016/0004-3702(77)90029-7.

29 Jiong Yang, Arijit Shaw, Teodora Baluta, Mate Soos, and Kuldeep S. Meel. Explaining SAT solving using causal reasoning. In *SAT*, volume 271 of *LIPIcs*, pages 28:1–28:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.SAT.2023.28.

A Appendix: More Details on Unlearning in Kissat

A simplified version of the unlearning algorithm implemented in Kissat is shown in Fig. 1 in Sect. 2. Below, we elaborate on aspects omitted from the simplified exposition. In our experiments, these aspects were either disabled in the evaluated base version or remained unchanged throughout the study.

A key aspect not yet discussed is the scheduling of the *unlearn* function – the limit on conflicts and learned clauses required before triggering the next *unlearn* call. In MiniSat, this interval increases *geometrically* (e.g., starting at 10k conflicts and growing by 50% each time). The Glucose paper [1] suggests adopting an *arithmetic* schedule by retaining critical clauses. For example, one may start unlearning after 1000 conflicts and increment the interval by a fixed amount (e.g., 300 conflicts per round), lowering the frequency of unlearning.

Under such an arithmetic scheme, the conflict interval between the n-th and next *unlearn* is $300 \cdot n + 1000$, or more generally $\mathit{period} \cdot n + \mathit{initial}$. Yet, even assuming a constant target fraction ($f = 50\%$ in Glucose) of removed clauses and no tier system, analyzing its effect on the expected number of retained clauses after n rounds is non-trivial. The competition version instead schedules unlearning after $\mathit{period} \cdot \sqrt{n} + \mathit{initial}$ conflicts, with both parameters set to 1000 – diverging from Glucose 's arithmetic scheme. As scheduling adds another investigative dimension, we leave it unchanged and defer further analysis to future work.

In the competition configuration version of Kissat, the target fraction f increases dynamically from 50% to 90%, scaling logarithmically with the number of conflicts. The base version uses a constant $f = 75\%$ and lacks this dynamic update.

Additionally, the competition version adopted a three-tier system, where clauses in tier 3 (e.g., LBD > 6) get only a second chance to survive unlearning, while medium-critical clauses (e.g., LBD > 2) receive a third chance, as in Sect. 6. The base version omits the extended *used* range for critical clauses in tier 1. This encoding allows even critical clauses (LBD ≤ 2) to be eventually unlearned if unused across many rounds. In competition, 5 bits encode *used*, so such clauses are removed after 31 unlearning rounds without use. In contrast, base sets *used* to 2, limiting the range to three values.

Moreover, like many other solvers, the competition configuration does *not* restart (i.e., backtrack to level zero) during unlearning (see also Sect. 3). Thus, clauses used as reasons on the trail must be retained. Such clauses, along with subsumed and root-level satisfied ones, are skipped in the loop in Fig. 1. Since restarts are more frequent in focused mode than stable mode, average conflict levels are lower in focused mode. Not restarting during unlearning thus increases the chance of retaining reason clauses in stable mode. To avoid this bias, the base version performs a restart before unlearning.

Finally, as discussed in Sect. 7 – the study's starting point – the default Kissat version *dynamically* computes LBD limits for tiers. However, despite diverse observed distributions, we found no measurable benefit from this technique.

B Appendix: Additional Experiments

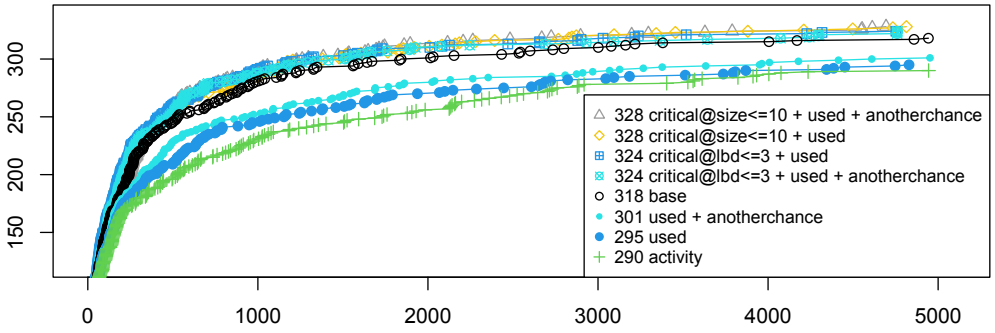


Figure 9 Comparing the effect of used, activity and the effect of giving clauses another chance.

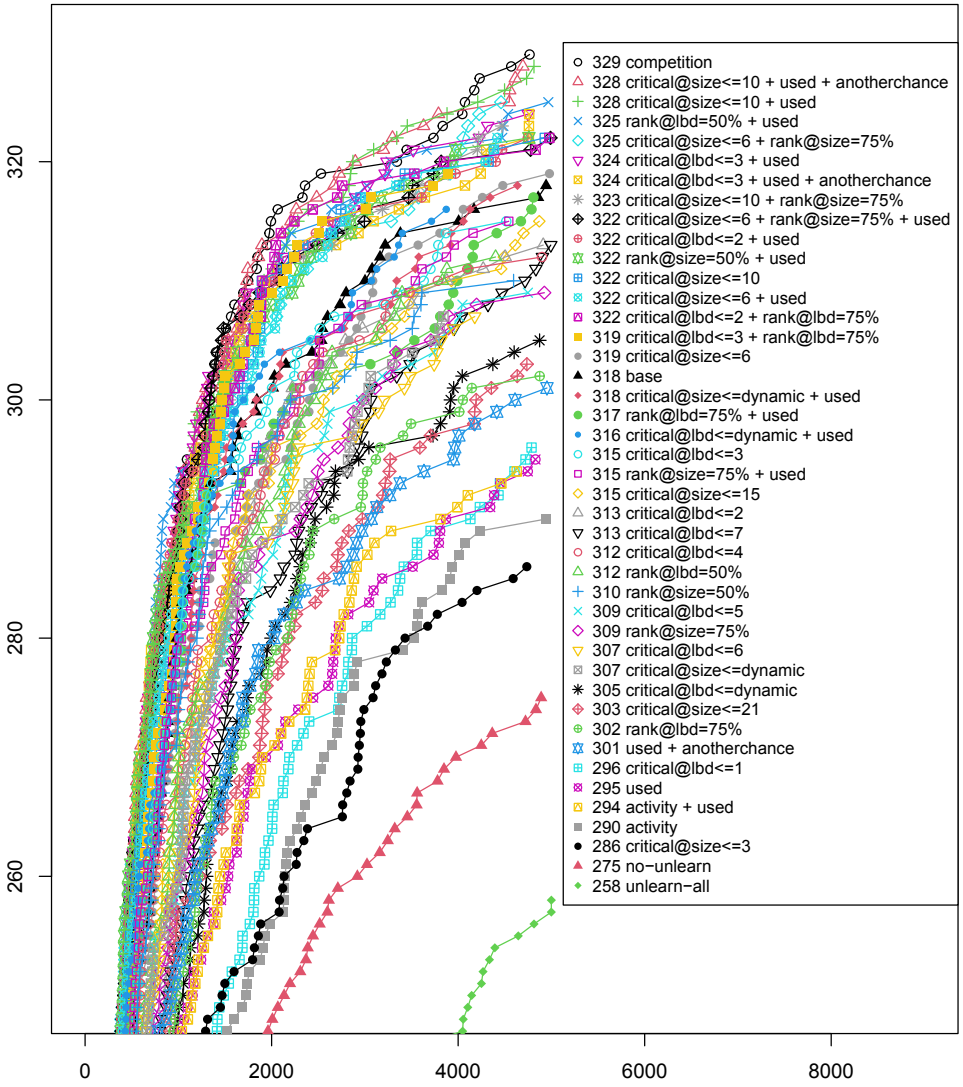


Figure 10 Cumulative distribution function (CDF, 100% = 400 instances) of all interesting runs.

RustSAT: A Library for SAT Solving in Rust

Christoph Jabs
University of Helsinki, Finland

Abstract

State-of-the-art Boolean satisfiability (SAT) solvers constitute a practical and competitive approach for solving various real-world problems. To encourage their widespread adoption, the relatively high barrier of entry following from the low level syntax of SAT and the expert knowledge required to achieve tight integration with SAT solvers should be further reduced. We present RustSAT, a library with the aim of making SAT solving technology readily available in the Rust programming language. RustSAT provides functionality for helping with generating (Max)SAT instances, writing them to, or reading them from files. Furthermore, RustSAT includes interfaces to various state-of-the-art SAT solvers available with a unified Rust API. Lastly, RustSAT implements several encodings for higher level constraints (at-most-one, cardinality, and pseudo-Boolean), which are also available via a C and Python API.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Rust, library, SAT solvers, constraint encodings

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.15

Supplementary Material
Software (RustSAT repository): `https://github.com/chrjabs/rustsat` [32]
archived at `swh:1:rel:ae4bbca3303bbd4d263bca42f01bf737638b36ce`
Software (Code & Data for Benchmarks): `https://github.com/chrjabs/rustsat-benchmarks` [33]
archived at `swh:1:dir:dd8bdc5aea20abd3ac594d6f24e58d1f8f05b480`

Funding This work is funded by Research Council of Finland (grant 356046).

Acknowledgements Thanks to the people who have already contributed to RustSAT on GitHub. Especially Christopher Jefferson, Jukka Suomela, and Noah Bruns have helped greatly in providing feedback based on different usecases for RustSAT. I also want to thank Matti Järvisalo and Jeremias Berg, who have supported me in working on RustSAT during my PhD research.

1 Introduction

Boolean satisfiability (SAT) solving is a significant success story of recent years [21]. State-of-the-art SAT solvers are highly optimized reasoning engines, able to solve propositional formulas with millions of variables and clauses, as demonstrated, e.g., by recent SAT Competitions [23, 28]. This has resulted in SAT solvers being commonly used as the underlying reasoning engine for solving a wide variety of problems, ranging from model checking [12, 9] over computer-assisted mathematics [50, 27, 51] to constraint satisfaction solving [44], and many more. Through their commonly supported incremental solving functionality [18, 38], SAT solvers also find application in problem-solving beyond NP [15, 6], and in optimization [3].

In order to achieve the level of performance optimization required, state-of-the-art SAT solvers are written in low-level languages, with C++ being the most common choice. This can pose some challenges when building tools based on SAT solving technology in different programming languages, as it often requires writing additional "boilerplate" code in order to interface with SAT solvers, which has to be repeated for each project and SAT solver. The standardized IPASIR interface [5] for incremental SAT solvers removes some of the required work, by unifying the API for the most-basic solver functionality. For the


© Christoph Jabs;
licensed under Creative Commons License CC-BY 4.0

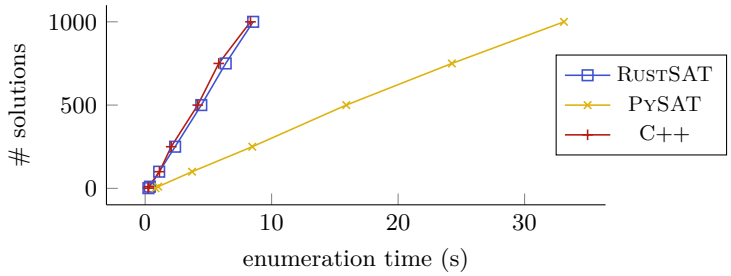


Figure 1 Runtime comparison of solution enumerator with RustSAT, PySAT, and C++ for the `AProVE11-12.cnf` instance.

Python ecosystem, the PySAT library [29, 31] has improved the accessibility of SAT solving technology significantly by packaging interfaces to state-of-the-art SAT solvers with helpful functionality for modelling problems as propositional logic. For interacting with SAT solvers from Python, OptiLog [1] defines a standardized `iSAT` C++ interface which includes more functionality than IPASIR. However, as many languages (e.g., Rust) offer direct interaction only with C APIs, `iSAT` is not universally applicable.

We present RustSAT, a library for the Rust programming language, that includes functionality for building and working with SAT instances, interfaces to common state-of-the-art SAT solvers, and CNF encodings for higher-level constraints. RustSAT is geared towards performance-sensitive applications written in Rust, where either SAT solvers are employed as part of the application, or propositional encodings for problem settings are encoded and then written to file. Some applications where RustSAT is already being used are a product configuration tool for the automotive industry [14], explaining pen and paper puzzles [20] (originally based on PySAT, but now using RustSAT), the Scuttle multi-objective MaxSAT solver [34], all written in Rust, and the Loandra [7] and MallobMax [49] MaxSAT solvers using RustSAT through its C API.

Being implemented in Rust, RustSAT provides a balance between accessibility – through Rust's modern build system and package distribution, as well as tooling to generate convenient API documentation – and performance. To illustrate the performance improvement that RustSAT can yield over PySAT in certain applications, as well as the little overhead incurred compared to not using any library at all, Figure 1 shows the time required to enumerate up-to 1000 solutions to the `AProVE11-12.cnf` instance from the SAT Competition 2022 Anniversary track, using PySAT, RustSAT and a custom C++ implementation.[1] While Python and PySAT can in many cases be competitive [29, 30], the more than 3× speedup observed in Figure 1 shows the potential efficiency improvement from implementing problem-solving tools using SAT solvers in a more efficient programming language such as Rust. Figure 1 also shows that RustSAT incurrs virtually no overhead compared to implementing solution enumeration in C++, employing the SAT solver directly. Additionally, Rust provides the advantages of type and memory safety, detecting many failure cases at compile time rather than at runtime.

All source code for RustSAT can be found at `https://github.com/chrjabs/rustsat`, and detailed and up-to-date API documentation is available at `https://docs.rs/rustsat`. Throughout this paper, when we refer to Rust types, functions or crates, we include hyperlinks to the API documentation online, formatted as in this example: `rustsat`.

[1] Experiment run on an AMD Ryzen 7 1700 (3 GHz) with 16 GB of memory, using the `pysat.examples.models` implementation included in PySAT (1.8.dev17) and the `enumerator` tool from `rustsat-tools`. All three approaches use MiniSat 2.2.0 as the SAT solver.

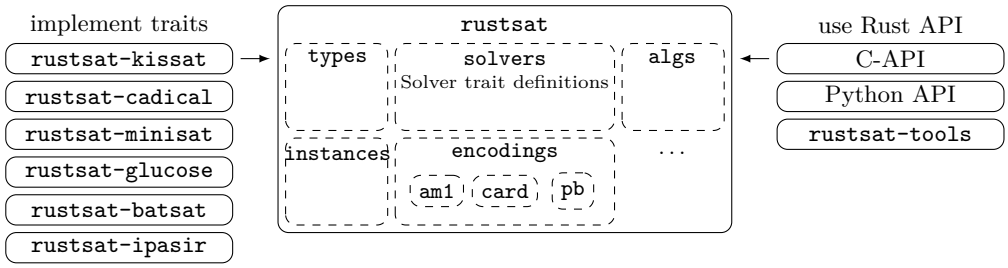


Figure 2 The architecture of the RustSAT project. Crates are represented as solid boxes, and modules as dashed boxes.

2 The Rust Programming Language

We give a brief overview of those features of the Rust programming language and its ecosystem that are relevant for understanding the design-principles of the RustSAT library.

Rust [52] is a programming language emphasizing performance, type safety, and reliability. In contrast to languages like Java or Python, Rust is fully compiled and does not use a garbage collector for memory management. In order to still achieve memory safety, Rust uses a system called the *borrow checker*, which during compilation tracks object lifetimes and ensures that all memory references are valid. For certain operations – e.g., when interfacing external C APIs or working with raw pointers – the Rust compiler cannot ensure memory safety. To enable usecases where such operations are necessary, Rust allows users to write so-called *unsafe* code, where memory safety guarantees need to be manually upheld by the user, as, e.g., when programming in C++.

Code inside a Rust project is typically structured into several (sub)modules, with paths to a type or function in a submodule being separated by two colons (`::`), a notation we also use throughout this paper. To enable generic code written against any type that supports specific functionality, Rust supports abstraction via so-called *traits*. Traits work similarly to interfaces in, e.g., Java: When defining a trait, one abstractly specifies certain methods that need to be provided by implementors of the trait. For example, a trait for SAT solvers might specify an `add_clause` and a `solve` method. A type that implements the trait then supplies the concrete implementations of the methods. Code that is generically written against a trait rather than a specific type can then be compiled with any type implementing that respective trait.

An important part of the Rust ecosystem is the `cargo` package manager. Similarly to the Python package manager `pip`, `cargo` takes care of downloading and compiling dependencies. Additionally, `cargo` provides a build system for projects, similar to `cmake` or GNU Autotools for C++ projects. A `cargo` package is called a *crate*, and crates are typically published on `https://crates.io`. The API documentation for crates on `crates.io` is automatically generated and available at `https://docs.rs/`.

3 Design Principles and Functionality

The architecture of RustSAT is illustrated in Figure 2. RustSAT consists of the main `rustsat` crate, which contains most of the functionality provided. Interfaces to SAT solvers, as well as the C and Python APIs are provided as separate crates. Additionally, the `rustsat-tools` crate contains some helpful executable tools for working with SAT-related instances, e.g., a tool to verify that a given assignment is indeed a solution to a CNF formula.

The aim of the RustSAT project is to be universal, convenient, and reliable to use, while not sacrificing performance. Towards this goal, we use an extensive automated test suite that is extended whenever bugs are discovered, to avoid regressions as much as possible. For reliability, we follow the semantic versioning scheme, where during this quite early stage of development (marked by major version 0), breaking changes will be published as a minor version bump, while any other changes are released as a patch version. All of RustSAT's functionality is accessible without requiring unsafe code. The only unsafe elements in the RustSAT API are intended as slight performance improvements when certain guarantees are already checked externally, and do therefore not need to be verified by RustSAT (e.g., `Lit::new_unchecked`). RustSAT is licensed under the MIT license, and developed entirely open-source. User feedback and contributions have already been very valuable in designing the API to be convenient in a wide range of usecases.

In the following, we detail the main functionality included in RustSAT version 0.7.2 split into functionality for working with SAT (and related) instances, solver interfaces, and higher-level constraint encodings.

3.1 Working with Problem Instances

In the `types` module, RustSAT defines various types that form the basis of working with SAT instances in Rust, and of the typesafe API of RustSAT. The type `Var` wraps a 32-bit integer to represent propositional variables (starting from index 0), and `Lit` represents literals as the variable index shifted one bit to the left, and the last bit signaling negation, i.e., with the same memory representation as in MiniSat [17]. The methods `Lit::from_ipasir` and `Lit::to_ipasir` convert literals to the integer representation used in IPASIR [5] and the DIMACS CNF [25] format. The `Clause` type is a wrapper around a vector providing some helpful functionality for working with clauses. Whenever a clause is referenced, RustSAT accepts any slice of literals (i.e., contiguous sequence in memory), making the `Clause` type optional to use. To build instances of satisfiability or optimization problems, the `SatInstance` and `OptInstance` types are available in the `instances` module.[2] As their constraints, these instance types support clauses, cardinality, and pseudo-Boolean constraints. To convert them to conjunctive normal form (`Cnf`), the constraint encodings described in Section 3.3 can be used. Objectives (`Objective`) can be represented either on the basis of soft clauses or as a linear function over literals. Instance objects can also be crated from DIMACS CNF [25] and WCNF [8], as well as OPB [48] files via parser implementations included in RustSAT. As an illustration of the API, slightly simplified method signatures of the `SatInstance` type are shown in Listing 1.

Variable management in RustSAT is handled by so-called variable managers that implement the `ManageVars` trait. Most central to the `ManageVars` trait is the `new_var` method, which is used for obtaining a variable that has not been used so far. The simplest variable manager simply keeps track of the next free variable index (`BasicVarManager`), while more advanced variable managers can maintain a mapping between propositional variables and any Rust objects (`ObjectVarManager`).

[2] If compiled with an additional feature flag, a `MultiOptInstance` type is available for multi-objective optimization.

Listing 1 Excerpt of the `SatInstance` API.

```
impl SatInstance {
  // Adding constraints
  pub fn add_clause(&mut self, cl: Clause);
  pub fn add_card_constr(&mut self, card: CardConstraint);
  pub fn add_pb_constr(&mut self, pb PbConstraint);
  pub fn add_lit_impl_clause(&mut self, a: Lit, b: &[Lit]);
  // File parsing
  pub fn from_dimacs_path(path: &Path) -> Result<Self>;
  pub fn from_opb_path(path: &Path, opts: opb::Options) -> Result<Self>;
  // Converting to CNF
  pub fn into_cnf(self) -> (Cnf, VarManager);
  // ...
}
```

3.2 SAT Solver Interfaces

The vast majority of state-of-the-art SAT solvers offer APIs in either C or C++. While calling C APIs from Rust is possible, the APIs for solvers often differ, with the IPASIR [5] interface for incremental solver standardizing only the most-central functionality. Furthermore, tightly integrating the build system of each solver with Rust's `cargo` build system to make them accessible in Rust requires additional work. RustSAT standardizes interfaces for common functionality in SAT solvers via a set of traits that are defined in the `solvers` module of the main `rustsat` crate. In separate crates, RustSAT packages various state-of-the-art SAT solvers that can be easily integrated in the `cargo` build system. These solver interface crates implement the traits defined in the `rustsat` crate, which enables easily swapping out a given SAT solver for another one that provides the same required functionality.

We first describe the traits defined by RustSAT, which unify the solver interfaces. Afterwards, we list the solvers RustSAT currently provides interfaces for, and which functionality they support.

All traits capturing SAT solver functionality are defined and documented in the `solvers` module. The documentation of this module also gives an up-to-date overview of the SAT solvers that are supported. The most basic functionality that every SAT solver supports is captured in the `Solve` trait. This trait includes methods for adding clauses, solving without assumptions, and obtaining solutions. *Incremental* solving [18, 38] is supported via the `SolveIncremental` trait, which contains the `solve_assumps` and `core` methods for solving under assumptions and obtaining a core. An overview of the most important methods in the `Solve` and `SolveIncremental` traits is given in Listing 2. Additional traits provide functionality for terminating solvers via callbacks (`Terminate`, compare `ipasir_set_terminate` [5]) or an asynchronous interrupt signal (`Interrupt`), tracking learned clauses (`Learn`, compare `ipasir_set_learn` [5]), setting preferred phases for variables (`PhaseLit`), excluding variables from being eliminated by pre/inprocessing (`FreezeVar`), flipping literals in found solutions (`FlipLit`), propagating assumptions (`Propagate`), setting resource limits (`LimitConflicts`, `LimitDecisions`, `LimitPropagations`), and getting solver statistis (`SolveStats`, `GetInternalStats`).

RustSAT provides crates for the state-of-the-art SAT solvers Kissat [11], CaDiCaL [10], MiniSat [17], and Glucose [2]. The detailed solver versions can be seen in Table 1; for some solvers multiple versions are supported, with the concrete version being selected via feature flags at compilation time. All except the first are incremental solvers and therefore implement the `SolveIncremental` trait. CaDiCaL supports the most functionality, implementing all

Listing 2 Excerpt of the `Solve` and `SolveIncremental` traits.

```rust
pub trait Solve {
  fn signature(&self) -> &'static str;
  fn solve(&mut self) -> Result<SolverResult>;
  fn lit_val(&mut self, lit: Lit) -> Result<TernaryVal>;
  fn add_clause(&mut self, clause: Clause) -> Result<()>;
  // ...
}

pub trait SolveIncremental {
  fn solve_assumps(&mut self, assumps: &[Lit]) -> Result<SolverResult>;
  fn core(&mut self) -> Result<Vec<Lit>>;
}
```

mentioned traits except for `LimitPropagations`. Details on which additional traits each solver implements can be found in the online documentation of the solver crates. In addition to those well-known solvers, RustSAT also implements an interface to BatSat [16], which is a reimplementation of MiniSat in Rust. Choosing BatSat allows for projects to be pure Rust, and therefore, e.g., compile to web assembly.

In addition to solver interfaces usable "out of the box", via the `rustsat-ipasir` crate RustSAT allows for linking to any user-provided library which implements IPASIR [5]. While CaDiCaL does implement the IPASIR interface, using CaDiCaL through its dedicated interface gives access to more functionality: the `rustsat-ipasir` crate only implements the `Solve`, `SolveIncremental`, `Learn`, `Term`, and `SolveStats` traits. Lastly, the `ExternalSolver` type allows for calling an executable solver binary with DIMACS CNF input, and parsing the output written to the standard output interface, given that it follows the output specification of the SAT competition [26]. This can be convenient for using solvers such as Gimsatul [22] that are not intended to be used as libraries.

3.3 Constraint Encodings

In the `encodings` module, RustSAT implements CNF encodings for higher-level constraints. Beyond simple encodings (e.g., a literal implying a clause, `atomics::lit_impl_clause`), mainly intended to increase code readability, encodings for at-most-one, cardinality, and pseudo-Boolean constraints are available. Providing efficiently implemented constraint encodings in RustSAT allows users to employ state-of-the-art constraint encodings when solving real-world problems without having to go through the error-prone process of implementing complex encodings themselves. Interfaces to the different constraint encodings are unified via

Table 1 SAT solver interfaces available for RustSAT.

Solver	Versions	Crate	Incremental
Kissat [11]	3.0.0 – 4.0.2	`rustsat-kissat`	no
CaDiCaL [10]	1.5.0 – 2.1.3	`rustsat-cadical`	yes
MiniSat [17]	2.2.0	`rustsat-minisat`	yes
Glucose [2]	4.2.1	`rustsat-glucose`	yes
Batsat [16]	0.6.0	`rustsat-batsat`	yes
IPASIR	–	`rustsat-ipasir`	yes
Call solver binary	–	`ExternalSolver`	no

Table 2 The constraint encodings implemented in RustSAT. Bound types supported by a specific encoding are indicated as upper bounds ($\leq$) and lower bounds ($\geq$).

Constraint	Encoding	Bounds	Incrementality
Pseudo-Boolean	`GeneralizedTotalizer` [36]	$\leq$	yes
	`BinaryAdder` [54, 19]	$\geq, \leq$	yes
	`DynamicPolyWatchdog` [43]	$\leq$	only changing bounds
Cardinality	`Totalizer` [4, 39]	$\geq, \leq$	yes
At-most-one	`Pairwise` [47]	≤ 1	no
	`Ladder` [24]	≤ 1	no
	`Bitwise` [46]	≤ 1	no
	`Commander` [37]	≤ 1	no
	`Bimander` [41]	≤ 1	no

traits: for at-most-one constraint encodings, the `am1::Encode` trait captures all functionality, while for cardinality and pseudo-Boolean constraint encodings upper and lower bounding, as well as incremental and non-incremental use are split into separate traits in the `card` and `pb` submodules. With incremental use of encodings we refer to either adding additional input literals, or changing the enforced bound, both while reusing previously built parts of the encoding [39]. Table 2 lists the encodings that are currently implemented and which functionality they support. To the best of our knowledge, RustSAT is currently the only readily available constraint encoding library providing an implementation of the dynamic polynomial watchdog encoding [43], which is used in state-of-the-art MaxSAT solvers [42].

For all constraint encodings implemented in RustSAT, we aim to produce the smallest number of clauses required to enforce the constraint that is being encoded. Towards this end, we employ the *cone of influence* strategy [43] which removes clauses from the encoding which contain pure literals – i.e., literals that only appear in one polarity – either directly, or after having already removed other clauses. Removing these clauses preserves both the correctness and the propagation properties of the encoding.

To illustrate the effect that the cone of influence strategy has, in Figure 3, we compare the number of clauses in CNF encodings generated by RustSAT (0.7.2), PySAT (1.8.dev17, note that the pseudo-Boolean constraint encodings provided are reexported from PbLib [45] via the PyPBLib interface), and Open-WBO (2.1) [40]. For the totalizer cardinality encoding [4], Figure 3 (left) shows the number of clauses in a totalizer encoding produced for 300 input literals and an enforced upper bound ranging from 1 to 300. It can be observed that PySAT seems to be generating the clauses required for enforcing *any* bound at all times. Open-WBO omits clauses only required to define the semantics for output variables corresponding to higher values than the currently enforced bound. In RustSAT, we also omit encoding output variables that are lower than the required range, which results in smaller encodings for higher bounds. Figure 3 also shows the number of generated clauses for binary adder (middle) and generalized totalizer (right) pseudo-Boolean encodings generated for 100–600 input literals with random weights in $[1, 100]$ and an enforced bound of 300. Also for the pseudo-Boolean encodings, RustSAT produces the fewest clauses, with PbLib and Open-WBO producing both more than $3\times$ the number of clauses for the binary adder, while the difference for the generalized totalizer encoding is smaller. Note that neither PbLib nor PySAT implement the generalized totalizer encoding.

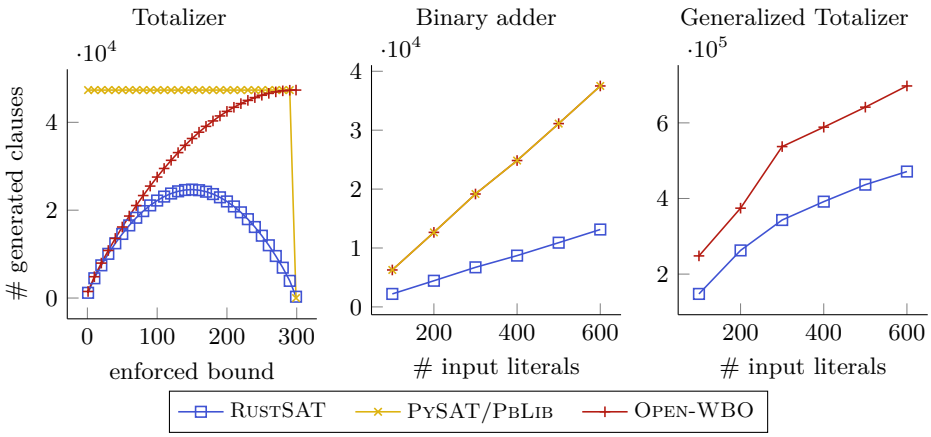


Figure 3 Left: Number of clauses in a totalizer cardinality encoding over 300 input literals for a given bound. Middle/Right: Number of clauses for a binary adder (middle) and generalized totalizer (right) pseudo-Boolean encoding for a given number of input literals with random weight in $[1, 100]$ and enforced bound of 300.

In addition to the CNF encodings listed in Table 2, in the `card::simulators` and `pb::simulators` submodules RustSAT provides helpers for inverting encodings (i.e., encoding an upper bound constraint with a lower-bounding encoding by inverting the constraint, and vice versa; `Inverted`), combining an upper and a lower-bounding encoding to get an encoding for both bound types at the same time, (`Double`) and encoding a pseudo-Boolean constraint by expanding it into a cardinality constraint with repeated input literals and using a cardinality encoding (`Card`).

Since the recent 0.7.0 release of RustSAT, the `Totalizer` and `GeneralizedTotalizer` encodings provide functionality for certifying the correctness of the generated CNF encoding by producing a proof in VeriPB format [13, 53, 35]. With this feature, the encodings can be employed in certified MaxSAT solvers, or for building a tool that produces certified translations from OPB to CNF. Currently, other encoding libraries such as PySAT or PbLib do not support generating certificates for the produced encodings.

3.4 C and Python API

While the primary intended way of using RustSAT is from the Rust programming language, some of its functionality is also exposed via a C and a Python API.

In version 0.7.2, the C API contains access for all higher-level constraint encodings listed in Table 2. In the C API, literals are represented as IPASIR-style `int`s, and clauses are returned via callbacks that work similarly to `ipasir_add` [5]. The main usecase for the C API is using the (incremental) encoding implementations in solvers written in a different language, as is for example the case for the Loandra MaxSAT solver [7]. To use the RustSAT C API in a project, the `rustsat-capi` crate needs to be compiled, which produces a statically linkable library. The full documentation of the C API can be found in `rustsat.h`.

Similarly to the C API, the Python API exposes all constraint encodings included in RustSAT. Additionally, the `Lit` and `Cnf` types, as well as a variable manager are included. At this point, the Python API of RustSAT does not include solver interfaces, but the RustSAT encodings can be used together with the solver interfaces in PySAT. The Python API is published on PyPI (`https://pypi.org/project/rustsat`) and its documentation is available online (`https://christophjabs.info/rustsat/pyapi`).

4 Conclusions

We presented RustSAT version 0.7.2, a library with the aim of making SAT solving technology more accessible from the Rust programming language. RustSAT packages tools for dealing with satisfiability and optimization instances in Rust, unified interfaces to state-of-the-art SAT solvers, and CNF encodings for at-most-one, cardinality, and pseudo-Boolean constraints. We have given an overview of the design principles of RustSAT– providing an easy-to-use API while not sacrificing performance – and illustrated key features empirically. RustSAT is available in open source, and detailed and up-to-date API documentation can be found online.

References

1 Josep Alos, Carlos Ansótegui, Josep M. Salvia, and Eduard Torres. OptiLog v2: Model, solve, tune and run. 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 25:1–25:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.25.

2 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11–17, 2009*, pages 399–404, 2009. URL: http://ijcai.org/Proceedings/09/Papers/074.pdf.

3 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiability. 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.

4 Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 – October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003. doi:10.1007/978-3-540-45193-8_8.

5 Tomas Balyo and Armin Biere. IPASIR: The standard interface for incremental satisfiability solving. https://github.com/biotomas/ipasir.

6 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.

7 Jeremias Berg, Christoph Jabs, Hannes Ihalainen, and Matti Järvisalo. Loandra in the 2024 MaxSAT evaluation. In Jeremias Berg, Matti Järvisalo, Ruben Martins, Andreas Niskanen, and Tobias Paxian, editors, *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*, volume B-2024-2 of *Department of Computer Science Report Series B*, pages 9–10. University of Helsinki, 2024.

8 Jeremias Berg, Matti Järvisalo, Ruben Martins, Andreas Niskanen, and Tobias Paxian. MaxSAT Evaluation 2024: Rules. https://maxsat-evaluations.github.io/2024/rules.html.

9 Armin Biere. Bounded model checking. 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 739–764. IOS Press, 2021. doi:10.3233/FAIA201002.

10 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification – 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

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

12 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.

13 Bart Bogaerts, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. VeriPB and CakePB in the SAT Competition 2023. In Tomas Baylo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, volume B-2023-1 of *Department of Computer Science Report Series B*, pages 86–88. University of Helsinki, 2023.

14 Noah Frederik Bruns. Application of multi-objective MaxSAT-solvers for optimizing highly configurable products. Master's thesis, Technische Universität Wien, 2024.

15 Edmund M. Clarke. SAT-based counterexample guided abstraction refinement. In Dragan Bosnacki and Stefan Leue, editors, *Model Checking of Software, 9th International SPIN Workshop, Grenoble, France, April 11–13, 2002, Proceedings*, volume 2318 of *Lecture Notes in Computer Science*, page 1. Springer, 2002. doi:10.1007/3-540-46017-9_1.

16 Simon Cruanes and Masaki Hara. BatSat: A (parametrized) rust sat solver originally based on minisat. https://github.com/c-cube/batsat.

17 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.

18 Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In Ofer Strichman and Armin Biere, editors, *First International Workshop on Bounded Model Checking, BMC@CAV 2003, Boulder, Colorado, USA, July 13, 2003*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 543–560. Elsevier, 2003. doi:10.1016/S1571-0661(05)82542-3.

19 Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *J. Satisf. Boolean Model. Comput.*, 2:1–26, 2006. doi:10.3233/sat190014.

20 Joan Espasa, Ian P. Gent, Ruth Hoffmann, Christopher Jefferson, and Alice M. Lynch. Using small MUSes to explain how to solve pen and paper puzzles. *CoRR*, abs/2104.15040, 2021. arXiv:2104.15040.

21 Johannes Klaus Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. The silent (r)evolution of SAT. *Commun. ACM*, 66:64–72, 2023. doi:10.1145/3560469.

22 Mathias Fleury and Armin Biere. Scalable proof producing multi-threaded SAT solving with gimsatul through sharing instead of copying clauses. *CoRR*, abs/2207.13577, 2022. doi:10.48550/arXiv.2207.13577.

23 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artif. Intell.*, 301:103572, 2021. doi:10.1016/j.artint.2021.103572.

24 Ian P Gent and Peter Nightingale. A new encoding of alldifferent into sat. In *International Workshop on Modelling and Reformulating Constraint Satisfaction*, volume 3, pages 95–110, 2004.

25 Maijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT Competition 2024: Benchmarks. https://satcompetition.github.io/2024/benchmarks.html.

26 Maijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT Competition output format. https://satcompetition.github.io/2024/output.html.

27 Marijn J. H. Heule and Manfred Scheucher. Happy ending: An empty hexagon in every set of 30 points. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part I*, volume 14570 of *Lecture Notes in Computer Science*, pages 61–80. Springer, 2024. doi:10.1007/978-3-031-57246-3_5.

28 Marijn J. K. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *SAT Competition 2024*, volume B-2024-1 of *Department of Computer Science Report Series B*. University of Helsinki, 2024. URL: http://hdl.handle.net/10138/584822.

29 Alexey Ignatiev, António Morgado, and João Marques-Silva. PySAT: A python toolkit for prototyping with SAT oracles. 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 428–437. Springer, 2018. doi:10.1007/978-3-319-94144-8_26.

30 Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient MaxSAT solver. *J. Satisf. Boolean Model. Comput.*, 11:53–64, 2019. doi:10.3233/SAT190116.

31 Alexey Ignatiev, Zi Li Tan, and Christos Karamanos. Towards universally accessible SAT technology. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21–24, 2024, Pune, India*, volume 305 of *LIPIcs*, pages 16:1–16:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SAT.2024.16.

32 Christoph Jabs. RustSAT. Software, version 0.7.2., swhId: swh:1:rel:ae4bbca3303bbd4d263bca42f01bf737638b36ce (visited on 2025-07-22). URL: https://github.com/chrjabs/rustsat, doi:10.4230/artifacts.24081.

33 Christoph Jabs. RustSAT Benchmarks. Software, swhId: swh:1:dir:dd8bdc5aea20abd3ac594d6f24e58d1f8f05b480 (visited on 2025-07-22). URL: https://github.com/chrjabs/rustsat-benchmarks, doi:10.4230/artifacts.24082.

34 Christoph Jabs. Scuttle: A multi-objective MaxSAT solver. https://bitbucket.org/coreo-group/scuttle.

35 Christoph Jabs, Jeremias Berg, Bart Bogaerts, and Matti Järvisalo. Certifying pareto optimality in multi-objective maximum satisfiability. In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3–8, 2025, Proceedings, Part II*, volume 15697 of *Lecture Notes in Computer Science*, pages 108–129. Springer, 2025. doi:10.1007/978-3-031-90653-4_6.

36 Saurabh Joshi, Ruben Martins, and Vasco Manquinho. Generalized totalizer encoding for pseudo-boolean constraints. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming – 21st International Conference, CP 2015, Cork, Ireland, August 31 – September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2015. doi:10.1007/978-3-319-23219-5_15.

37 William Klieber and Gihwon Kwon. Efficient CNF encoding for selecting 1 from n objects. In *Workshop on Constraints in Formal Verification*, 2007.

38 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.

39 Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. 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 531–548. Springer, 2014. doi:10.1007/978-3-319-10428-7_39.

40 Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver,. 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 438–445. Springer, 2014. doi:10.1007/978-3-319-09284-3_33.

41 Van-Hau Nguyen and Son Thai Mai. A new method to encode the at-most-one constraint into SAT. In Huynh Quyet Thang, Le Anh Phuong, Luc De Raedt, Yves Deville, Marc Bui, Truong Thi Dieu Linh, Thi-Oanh Nguyen, Dinh Viet Sang, and Nguyen Ba Ngoc, editors, *Proceedings of the Sixth International Symposium on Information and Communication Technology, Hue City, Vietnam, December 3–4, 2015*, pages 46–53. ACM, 2015. doi:10.1145/2833258.2833293.

42 Tobias Paxian and Bernd Becker. Pacose: An iterative SAT-based MaxSAT solver. In Jeremias Berg, Matti Järvisalo, Ruben Martins, Andreas Niskanen, and Tobias Paxian, editors, *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*, volume B-2024-2 of *Department of Computer Science Report Series B*, page 26. University of Helsinki, 2024.

43 Tobias Paxian, Sven Reimer, and Bernd Becker. Dynamic polynomial watchdog encoding for solving weighted MaxSAT. 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 37–53. Springer, 2018. doi:10.1007/978-3-319-94144-8_3.

44 Laurent Perron, Frédéric Didier, and Steven Gay. The CP-SAT-LP solver (invited talk). In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27–31, 2023, Toronto, Canada*, volume 280 of *LIPIcs*, pages 3:1–3:2. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CP.2023.3.

45 Tobias Philipp and Peter Steinke. PBLib – A library for encoding pseudo-boolean constraints into CNF. 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 9–16. Springer, 2015. doi:10.1007/978-3-319-24318-4_2.

46 Steven D. Prestwich. Finding large cliques using sat local search. In Barry O'Sullivan Frédéric Benhamou, Narendra Jussien, editor, *Trends in Constraint Programming*, chapter 15, pages 269–274. John Wiley & Sons, Ltd, 2007. doi:10.1002/9780470612309.ch15.

47 Steven D. Prestwich. CNF encodings. 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 75–100. IOS Press, 2021. doi:10.3233/FAIA200985.

48 Olivier Roussel. General opb format. https://www.cril.univ-artois.fr/PB24/OPBgeneral.pdf.

49 Dominik Schreiber, Christoph Jabs, and Jeremias Berg. From scalable SAT to MaxSAT: Massively parallel solution improving search. In Maxim Likhachev, Hana Rudová, and Enrico Scala, editors, *Eighteenth International Symposium on Combinatorial Search, SoCS 2025, Glasgow, UK, August 12–15, 2025*. AAAI Press, 2025. to appear.

50 Bernardo Subercaseaux and Marijn J. H. Heule. The packing chromatic number of the infinite square grid is at least 14. 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 21:1–21:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.21.

51 Bernardo Subercaseaux, Wojciech Nawrocki, James Gallicchio, Cayden R. Codel, Mario Carneiro, and Marijn J. H. Heule. Formal verification of the empty hexagon number. In

Yves Bertot, Temur Kutsia, and Michael Norrish, editors, *15th International Conference on Interactive Theorem Proving, ITP 2024, September 9–14, 2024, Tbilisi, Georgia*, volume 309 of *LIPIcs*, pages 35:1–35:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.ITP.2024.35.

52 The Rust Team. Rust: A language empowering everyone to build reliable and efficient software. https://www.rust-lang.org/.

53 Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. In Georg Gottlob, Daniela Inclezan, and Marco Maratea, editors, *Logic Programming and Nonmonotonic Reasoning – 16th International Conference, LPNMR 2022, Genova, Italy, September 5–9, 2022, Proceedings*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, 2022. doi:10.1007/978-3-031-15707-3_33.

54 Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Inf. Process. Lett.*, 68:63–69, 1998. doi:10.1016/S0020-0190(98)00144-6.

Depth-Optimal Quantum Layout Synthesis as SAT

Anna B. Jakobsen
Department of Computer Science, Aarhus University, Denmark

Anders B. Clausen
Department of Computer Science, Aarhus University, Denmark

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

Irfansha Shaik
Department of Computer Science, Aarhus University, Denmark
Kvantify Aps, Copenhagen S, Denmark

Abstract

Quantum circuits consist of gates applied to qubits. Current quantum hardware platforms impose connectivity restrictions on binary CX gates. Hence, Layout Synthesis is an important step to transpile quantum circuits before they can be executed. Since CX gates are noisy, it is important to reduce the CX count or CX depth of the mapped circuits.

We provide a new and efficient encoding of Quantum-circuit Layout Synthesis in SAT. Previous SAT encodings focused on gate count and CX-gate count. Our encoding instead guarantees that we find mapped circuits with minimal circuit depth or minimal CX-gate depth. We use incremental SAT solving and parallel plans for an efficient encoding. This results in speedups of more than 10-100x compared to OLSQ2, which guarantees depth-optimality. But minimizing depth still takes more time than minimizing gate count with Q-Synth.

We correlate the noise reduction achieved by simulating circuits after (CX)-count and (CX)-depth reduction. We find that minimizing for CX-count correlates better with reducing noise than minimizing for CX-depth. However, taking into account both CX-count and CX-depth provides the best noise reduction.

2012 ACM Subject Classification Hardware → Quantum computation; Computing methodologies → Planning for deterministic actions

Keywords and phrases Quantum Layout Synthesis, Transpiling, Circuit Mapping, Incremental SAT, Parallel Plans

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.16

Related Version *Extended Version*: `https://arxiv.org/abs/2506.06752` [8]

Supplementary Material *Software (Github)*: `https://github.com/anbclausen/quills`
archived at `swh:1:dir:ae986beceee3051ca8ec16feed3ddc0b2afd6aab`
Software (Zenodo link): `https://doi.org/10.5281/zenodo.15606051` [9]

Funding This research was partially funded by the Innovation Fund Denmark project "Automated Planning for Quantum Circuit Optimization" and by the European Innovation Council through Accelerator grant no. 190124924.

Acknowledgements This paper is based on the MSc-thesis of Anna B. Jakobsen and Anders B. Clausen. They contributed equally to this work. We thank Kostiantyn V. Milkevych for porting Quills to Q-Synth v4.

1 Introduction

With recent advances in quantum hardware, there is a surge of interest in scalable compilation techniques. In order to execute a quantum program, it must be compiled to a circuit with unary and binary gates, while handling hardware restrictions. The quantum compilation


© Anna B. Jakobsen, Anders B. Clausen, Jaco van de Pol, and Irfansha Shaik;
licensed under Creative Commons License CC-BY 4.0



pipeline broadly consists of two steps: Circuit Synthesis and Layout Synthesis. In this paper, we focus on the Quantum Layout Synthesis (QLS) problem [13, 24, 15, 21] that transforms a "logical" quantum circuit consisting of native gates into an equivalent circuit that respects the connectivity of a physical quantum platform. Many quantum platforms do not support all-to-all connectivity of physical qubits, thus restricting the application of binary gates to only neighboring qubits. We assume that the only gates in the input circuit are unary gates and the binary CX (Conditional NOT) gate. The physical platform is specified by an (undirected) graph on physical qubits, the coupling map. The output is an initial mapping of the logical qubits onto the physical qubits, as well as a new, equivalent circuit, in which CX gates are only applied to physical qubits that are connected in the coupling map. To ensure this, so-called SWAP gates may be inserted to route interacting logical qubits towards connected physical qubits. During routing, the order of the gates must also be respected.

Note that the QLS problem may admit many possible solutions. One can easily schedule all gates by eagerly adding SWAP gates to bring distant logical qubits closer together. Although the primary objective of QLS is to handle connectivity restrictions, the secondary objective is to maximize hardware performance. In the current Noisy Intermediate Scale Quantum (NISQ) era, qubits are noisy, and every gate adds to the error. Reducing noise is crucial for practical quantum computing. While there does not exist a single metric that predicts actual hardware performance, there exist some simple metrics like gate-count that are indicative. Since binary CX gates can be up to 10 times more error-prone than unary gates, the number of CX gates (CX-count metric) is more relevant than just gate count [14]. Since SWAP gates are usually implemented by 3 CX gates each, solutions with too many SWAPs should be avoided. Several heuristic approaches have been proposed that reduce the additional SWAP gates. The most famous is SABRE [13] (available in IBM's Qiskit), which implements a very fast heuristic to find a correct layout mapping. The tool MQT QMAP provides a heuristic approach based on the A*-algorithm [27]. There exist a myriad of other heuristic approaches that focus on CX-count optimization, we refer to [23] for a detailed survey. While heuristic approaches are fast, their solutions are often far from optimal [15]. Unfortunately, the optimal QLS problem is NP-hard [23]. The problem remains hard when considering natural restrictions on the coupling graph, like planar graphs [5] and bipartite graphs [11], and it is even conjectured to be NP-hard for simple linear neighbor graphs [7]. Thus, unless P=NP, one needs exhaustive exact approaches for (near)-optimal solutions.

Several exact approaches based on SMT [24, 15], Classical Planning [20], and SAT [21, 26] have been proposed to optimize CX-count. Synthesizing a k-SWAP optimal circuit involves refuting the existence of a mapped circuit of at most $k-1$ SWAPs. Recent SAT approaches proposed encodings whose makespan corresponds to the number of inserted SWAP gates instead of the number of CX-gates. Since there are typically fewer SWAP gates than CX gates, such SAT encodings significantly outperform earlier approaches [21].

Despite the progress in CX-count optimization, CX-count alone is not sufficient to predict the hardware performance. Note that qubits become unstable over time due to so-called *decoherence*. It is essential to execute gates in the circuit before the qubits become unstable. To reduce execution time, multiple gates can be executed in parallel, provided they are independent, i.e., do not act on the same qubits. This makes circuit depth an important metric, since it captures this notion of parallel execution. Since CX-gates are up to 10x slower compared to unary gates, we focus on reducing the CX-depth of the circuit. This metric counts the longest chain of dependent CX-gates.

Optimization of circuit depth has been explored in [15] in the context of SMT solving. Their tool OLSQ2 is the state-of-the-art tool for depth-optimal synthesis. In practice, optimization of (CX)-depth is harder than (CX)-count. Unlike (CX)-count optimization,

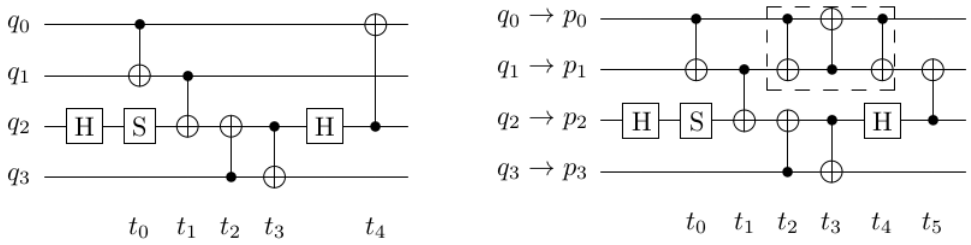
the makespan of an encoding optimizing (CX)-depth must be at least the (CX)-depth of the initial circuit. Since the (CX)-depth is typically higher than the number of SWAPs, depth optimal synthesis becomes much harder in practice. Compared to the SAT based state-of-the-art tool Q-Synth v2 for CX-count optimization, OLSQ2 is indeed up to 5 orders of magnitude slower [21, 15]. Even the best SMT-based near-optimal tool TB-OLSQ2 is up to 2 orders of magnitude slower compared to Q-Synth v2 [21]. SAT-based tools are shown to significantly outperform SMT-based tools for QLS [21, 26]. The natural next step is to investigate if an efficient SAT-based depth optimal encoding is feasible.

Contributions

In this paper, we provide an encoding of (CX)-depth-optimal QLS in SAT for the first time. Additionally, we provide a tool, QuilLS[1], which implements the encoding. Our approach is based on increasing the time horizon, where in each time step one layer of independent gates is added. The first SAT solution found corresponds to a mapped circuit with optimal depth. We use incremental SAT solving (reusing partial information from previous calls) [16] and parallel plans to encode depth as makespan for efficient solving.

A second contribution is an extensive experiment, comparing various tools with respect to their running time and their performance in actual (CX)-count and (CX)-depth metrics. We compare our new tool with SABRE, Q-Synth and (TB)-OLSQ2. We run these tools on a standard set of quantum circuit benchmarks, obtained from [15, 21] and on the coupling graph of a number of standard quantum platforms (with 5-80 qubits). Our tool is often more than 10-100x faster compared to previous tools with guaranteed depth-optimality, but still slower than size-optimal tools.

Finally, we investigate how the various optimization metrics correlate with a reduction in noise, which is the ultimate goal of optimal QLS. Following the approach in [12, 25], we simulate all mapped circuits on a quantum platform with and without simulated noise, using the Qiskit simulator, and compute the Hellinger distance [3, 6, 19] of these runs. Surprisingly, we find that a reduction in CX-count correlates better with noise reduction than CX-depth. However, taking both measures, CX-count + CX-depth, into account leads to the best noise reduction, motivating the importance of our contribution.



(a) An Example circuit.

(b) CX-depth optimal circuit with 1 extra SWAP.

Figure 1 Example circuit, before and after mapping on a linear neighbour graph. The dotted box represents the extra SWAP gate implemented as 3 CX gates. Time steps t_0 to t_5 indicate the CX gate parallel steps.

[1] Available at `https://github.com/anbclausen/quills`, also ported to Q-Synth v4 at `https://github.com/irfansha/Q-Synth`

2 Preliminaries

2.1 Layout Synthesis of an Example Circuit

As discussed previously in the introduction, the Quantum Layout Synthesis (QLS) problem takes as inputs a quantum circuit and a platform coupling map. For example, consider the 4-qubit circuit in Figure 1a with 5 CX-gates and 3 unary gates. For simplicity, consider a quantum platform of 4 qubits $\{p_0, p_1, p_2, p_3\}$ with linear connections i.e., only $(p_0, p_1), (p_1, p_2), (p_2, p_3)$ are connected. Notice that CX gates at t_0, t_1, t_4 form a triangle in Figure 1a, however our linear platform does not have a triangle. Thus, there is no mapping from logical to physical qubits such that all the CX gates can be scheduled. In such cases, SWAP gates can be inserted into the circuit, which allow physical qubits to swap their states. A SWAP gate can be constructed of three CX gates [17]. Indeed, as shown in Figure 1b, by using one extra SWAP gate we can schedule all gates respecting connectivity restrictions.

Layout synthesis can be optimized using various metrics. One is size optimality, where the number of gates in the output circuit is minimized. In practice, this means minimizing the number of SWAP gates. Another is minimizing the depth of the output circuit, which is defined as the longest path in the dependency graph of the gates in the circuit. A similar metric is the CX-depth, where only the dependencies between the CX gates are considered. Our original circuit has CX-depth 5 where as the mapped circuit has optimal CX-depth of 6. In Figure 1, we indicate the parallel steps with CX-gates before and after mapping. The number of parallel steps with CX-gates essentially corresponds to CX-depth. Optimization of CX-count differs from CX-depth largely due to the interleaving of SWAP gates with other CX gates. The choice of optimization goal can have a large impact. For example, one of the benchmark circuits used in our experimental evaluation (‘`tof_5`’), has a depth-optimal solution on the platform Guadalupe of depth 64 using 31 SWAP gates, while a size-optimal circuit uses only 5 SWAP gates, but has depth 71.

2.2 Layout Synthesis as SAT

Given a propositional formula, the SAT problem is to find an assignment to the variables such that the formula evaluates to true. While the general problem is NP-complete, in practice several instances can be solved efficiently by modern SAT solvers [2]. This motivates encoding other NP-complete problems to SAT. However, finding an efficient problem encoding is highly non-trivial. As discussed earlier, recent SAT based approaches [21, 26] perform well on CX-count metric. Unlike (CX)-depth optimization, a SAT encoding optimizing (CX)-count only requires a makespan of 2 on the example in Figure 2. The key idea is to group CNOTs at each parallel step and add a single SWAP gate from time step t_1. Thus, the makespan scales with the number of SWAPs which is typically much lower than the (CX)-count or (CX)-depth.

While the disadvantage in the larger required makespan is unavoidable, we can still apply parallel plans [10] to schedule parallel gates in a mapped circuit. It is possible to encode depth in a sequential encoding, however allowing independent gates in the same time step can reduce the makespan. The makespan refers to the circuit (CX)-depth of the solution. Reducing makespan indeed avoids many variable copies, thus impacting performance. We adapt another main technique, incremental SAT solving, used in the earlier SAT encodings for our (CX)-depth optimal encoding. Remember that an optimal synthesis of k (CX)-depth circuit requires refuting the existence of $k-1$ or less (CX)-depth circuits. Instead of checking for circuits with (CX)-depth from 0 to $k-1$ independently, we reuse information from earlier SAT calls. By using incremental SAT solving, we can maintain a single CNF formula and

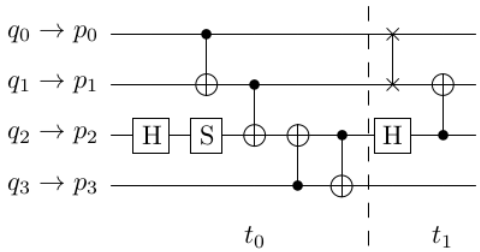


Figure 2 Mapped circuit with only 2 parallel steps in the context of CX-count optimization.

"turn clauses on or off" instead of rebuilding the entire CNF formula every time. This also increases the performance during *inprocessing* (the actual work of the SAT solver). Finally, for a (CX)-depth optimal encoding, we need to handle the execution of SWAP gates over 3 time steps instead of just 1.

3 Depth-optimal SAT encoding

In the previous sections, we observed that SAT-based encodings in Q-Synth perform very well in the case of size-optimal synthesis. In this section, we formulate a similar SAT encoding of depth-optimal layout synthesis. In contrast to size-optimal synthesis, depth-optimal synthesis might produce solutions with a sub-optimal number of CX gates to find a solution with optimal circuit depth. Circuit depth refers to the longest path in the gate dependency graph. We employ two techniques, parallel plans, and incremental SAT solving for an efficient encoding. Unlike Q-Synth, a parallel step in our encoding encodes a single layer in a circuit, consisting of parallel gates applied to different qubits. Thus, the makespan of the encoding corresponds to the depth of the mapped circuit. The novelty of our encoding comes from the handling of SWAP gates while maintaining depth-optimality, which is non-trivial.

The overall structure of the encoding can be seen in Algorithm 1. Since the output cannot have smaller depth than the input, we only test the existence of a mapping when the makespan is at least as large as the input depth. The first time step where the instance is satisfied is the optimal depth. Much of the structure of the encoding follows that of [21]. We will focus the in-depth explanation to those parts that are different.

Algorithm 1 Building and solving the SAT encoding.

Input: Coupling map $\mathcal{P}$ and circuit $\mathcal{C}$
  let D be the depth of $\mathcal{C}$
  let instance be an empty CNF
  for $t = 1, 2, \ldots$ **do**
    Add $MappingConstraints(t)$ to instance
    Add $ConnectivityConstraints(t)$ to instance
    Add $GateConstraints(t)$ to instance
    Add $SwapConstraints(t)$ to instance
    if $t \geq D$ **then**
      Add $Assumptions(t)$ to instance
      Solve instance with assumption asm^t
      if instance is satisfied **then**
        return satisfying assignment

Table 1 Encoding variables and descriptions.

Variable	Description
$\mathrm{mp}^t_{q,p}$	logical qubit q is mapped to the physical qubit p at time t.
oc^t_p	physical qubit p is occupied by some logical qubit at time t.
$\mathrm{e}^t_{q,q'}$	logical qubits q, q' are mapped to connected qubits at time t.
u^t_p	physical qubit p is not part of a SWAP gate at time t.
c^t_g	gate g is currently applied at time t.
a^t_g	gate g was applied at a time strictly before t.
d^t_g	gate g is delayed and has not been applied yet at time t.
$\mathrm{sw}^t_{p,p'}$	physical qubits p, p' are swapped at times t, $t-1$ and $t-2$.
$Sw(p)^t$	the set of variables $\mathrm{sw}^t_{p,p'}$ where $(p,p') \in P_{conn}$
st^t_p	physical qubit p is touched by a SWAP at t, $t-1$ and $t-2$.
asm^t	assumption to indicate that t is the final time step.

Table 1 presents the variables used in the encoding. For each time step, a set of variables is generated for the constraints of that time step. The asm^t variable is special and is used for incremental SAT solving; it provides context to the SAT solver about previous attempts to find a solution, essentially reducing the search space of the solver. SWAP gates are scheduled over 3 time steps since they are commonly implemented as three consecutive CX gates. In the following, we describe each part of the constraints added for each time step.
The Mapping and Connectivity constraints are exactly as in [21], but are repeated here for the sake of completeness. We use P for the set of physical qubits and Q for the set of logical qubits. P_{conn} indicates the edges between physical qubits in the connectivity graph. We use $g.q$ to indicate the logical qubit that a unary gate must be applied on, and $g.q_0, g.q_1$ for the qubits that a CX gate must be applied on. G_{CX} denotes the set of CX gates in the input circuit while G_{unary} denotes the set of unary gates.

MappingConstraints

$$\bigwedge_{q \in Q} \mathrm{ExactlyOne}(\mathrm{mp}^t_{q,p_0}, \ldots, \mathrm{mp}^t_{q,p_n}) \tag{1}$$

$$\bigwedge_{p \in P} \mathrm{AtMostOne}(\mathrm{mp}^t_{q_0,p}, \ldots, \mathrm{mp}^t_{q_m,p}) \tag{2}$$

$$\bigwedge_{p \in P} \left(\mathrm{oc}^t_p \iff \left(\bigvee_{q \in Q} \mathrm{mp}^t_{q,p}\right)\right) \tag{3}$$

The above constraints ensure that the $\mathrm{mp}^t_{q,p}$ and oc^t_p variables are set correctly. Here $\mathrm{ExactlyOne}(t_1, \ldots, t_n) \triangleq \sum_{i=1}^{n} t_i = 1$ denotes that exactly one of $\{t_1, \ldots, t_n\}$ is true while $\mathrm{AtMostOne}(t_1, \ldots, t_n) \triangleq \sum_{i=1}^{n} t_i \leq 1$ denotes that at most one of $\{t_1, \ldots, t_n\}$ is true.

ConnectivityConstraints

$$\bigwedge_{q,q' \in Q} \left(\bigwedge_{(p,p') \in P_{conn}} \left((\mathrm{mp}^t_{q,p} \wedge \mathrm{mp}^t_{q',p'}) \Rightarrow \mathrm{e}^t_{q,q'}\right) \quad \wedge \bigwedge_{p,p' \notin P_{conn}} \left((\mathrm{mp}^t_{q,p} \wedge \mathrm{mp}^t_{q',p'}) \Rightarrow \neg \mathrm{e}^t_{q,q'}\right)\right) \tag{4}$$

$$\bigwedge_{g \in G_{CX}} \left(\mathrm{c}^t_g \implies \mathrm{e}^t_{g.q_0, g.q_1}\right) \tag{5}$$

Constraint 4 ensures that the $e^t_{q,q'}$ predicates are true if and only if the physical qubits assigned to q and q' at time t are connected. Constraint 5 ensures that the CX gates are applied to connected physical qubits.

GateConstraints

The Gate constraints follow a similar pattern to those in [21], but are substantially different because in this encoding at most one gate can be scheduled on each qubit at a given time step. For a given time t, a gate is either currently scheduled denoted c_g^t, it has been scheduled for some $t' < t$ denoted a_g^t (advanced), or it will be scheduled for some $t' > t$ denoted d_g^t (delayed) as encoded in constraint 6. Furthermore, no gate can be "advanced" on the first time step since nothing has been applied yet as encoded in constraint 7.

$$\bigwedge_{g \in G} \mathrm{ExactlyOne}(\mathrm{c}_g^t, \mathrm{a}_g^t, \mathrm{d}_g^t) \tag{6}$$

$$\bigwedge_{g \in G} \neg \mathrm{a}_g^1 \tag{7}$$

The main idea from the size-optimal encoding was that of two-way propagation. This is adapted to a depth-optimal encoding by ensuring that if a gate is "current" or "delayed" its direct successors must be "delayed", while if a gate is "current" or "advanced" its direct predecessors must be "advanced". This information is propagated throughout the circuit, giving rise to the name. Constraint 8 enforces that the dependencies of the input circuit are preserved in the output circuit and that two gates are not applied on the same qubit at the same time. We use $S(g)$ (and $P(g)$) to denote the successors (predecessors) of gate g in the dependency graph of the input circuit.

$$\bigwedge_{g \in G} \Big(\bigwedge_{g' \in S(g)} (\mathrm{c}_g^t \vee \mathrm{d}_g^t) \implies \mathrm{d}_{g'}^t \ \wedge \bigwedge_{g' \in P(g)} (\mathrm{c}_g^t \vee \mathrm{a}_g^t) \implies \mathrm{a}_{g'}^t \Big) \tag{8}$$

To ensure that gates are not skipped or applied twice, constraint 9 ensures that a gate is "advanced" if and only if it was "current" or "advanced" in the previous time step. Likewise, a gate is "current" or "delayed" if and only if it was "delayed" in the previous time step.

$$\bigwedge_{g \in G} \Big(\big((\mathrm{c}_g^{t-1} \vee \mathrm{a}_g^{t-1}) \iff \mathrm{a}_g^t \big) \wedge \big(\mathrm{d}_g^{t-1} \iff (\mathrm{c}_g^t \vee \mathrm{d}_g^t) \big) \Big) \tag{9}$$

Constraint 10 is added to speed up solving time by reducing the search space. This constraint express that when a gate is "current" none of its predecessors and successors can be "current". We use $\mathit{fullS}(g)$ (and $\mathit{fullP}(g)$) to denote the transitive closure of the successor (predecessor) relation in the dependency graph. We do this to improve the automatic propagation of the dependency constraints.

$$\bigwedge_{g \in G} \bigwedge_{g' \in \mathit{fullS}(g) \cup \mathit{fullP}(g)} \mathrm{c}_g^t \implies \neg \mathrm{c}_{g'}^t \tag{10}$$

Whenever a gate is "current", the physical qubit(s) that it is being applied on must be "usable", i.e., not undergoing a SWAP. This is encoded for binary gates in constraint 11 and for unary gates in constraint 12. These constraints ensure that a gate and a SWAP are not applied at the same time step.

$$\bigwedge_{g \in G_{CX}} \bigwedge_{p \in P} \mathrm{c}_g^t \implies ((\mathrm{mp}_{g.q_0,p}^t \vee \mathrm{mp}_{g.q_1,p}^t) \implies \mathrm{u}_p^t) \tag{11}$$

$$\bigwedge_{g \in G_{unary}} \bigwedge_{p \in P} \mathrm{c}_g^t \implies (\mathrm{mp}_{g.q,p}^t \implies \mathrm{u}_p^t) \tag{12}$$

SwapConstraints

Constraint 13 encodes that a given physical qubit can participate in at most one SWAP gate for any three consecutive time steps. This is to ensure that SWAP gates are not applied in conflict with each other. We use $Sw(p)^t$ to mean the set of variables $\mathrm{sw}^t_{p,p'}$ where $(p, p') \in P_{conn}$.

$$\bigwedge_{p \in P} \mathrm{AtMostOne}(Sw(p)^t \cup Sw(p)^{t-1} \cup Sw(p)^{t-2}) \tag{13}$$

The "swap" variables indicate the last time step of a SWAP gate. To ensure that gates are not scheduled on top of SWAPs we require that if the "swap touched" variable is set for a qubit, it is not available for gates to be scheduled on it for this and the preceding two time steps as seen in constraint 14. Also, as encoded in constraint 15, the "swap" variables must all be set to false for the first three time steps.

$$\bigwedge_{p \in P} \bigwedge_{t' \in \{t, t-1, t-2\}} \left(\mathrm{st}^t_p \implies \neg \mathrm{u}^{t'}_p\right) \tag{14}$$

$$\bigwedge_{p,p' \in P_{conn}} \left(\neg \mathrm{sw}^1_{p,p'} \wedge \neg \mathrm{sw}^2_{p,p'} \wedge \neg \mathrm{sw}^3_{p,p'}\right) \tag{15}$$

Nothing stops the SAT solver from swapping unmapped physical qubits. Such SWAPs are referred to as 'spurious SWAPs'. To ensure that no spurious SWAPs are scheduled, we require that at least one physical qubit in a SWAP must have a logical qubit mapped to it in constraint 16. We denote a SWAP with only one mapped physical qubit 'ancillary SWAP'. Ancillary SWAPs can be disabled by requiring that both physical qubits participating in a SWAP be occupied. We only present results allowing ancillary SWAPs.

$$\bigwedge_{p,p' \in P_{conn}} \left(\mathrm{sw}^t_{p,p'} \implies (\mathrm{oc}^t_p \vee \mathrm{oc}^t_{p'})\right) \tag{16}$$

Three final constraints are added to encode the effect of a SWAP gate. Constraint 17 says that if a "swap" variable is set which includes the physical qubit p, the "swap touched" variable for p is also set. Constraint 18 says that if the swap touched variable is not set for a qubit, its mapping is preserved from the last time step. Finally, constraint 19 says that when the swap variable for two qubits is set, the mappings for them switch compared to the previous time step.

$$\bigwedge_{p \in P} \left(\mathrm{st}^t_p \iff \left(\bigvee_{p' \in conn(p)} \mathrm{sw}^t_{p,p'}\right)\right) \tag{17}$$

$$\bigwedge_{p \in P} \bigwedge_{q \in Q} \left(\neg \mathrm{st}^t_p \implies (\mathrm{mp}^{t-1}_{q,p} \iff \mathrm{mp}^t_{q,p})\right) \tag{18}$$

$$\bigwedge_{p,p' \in P_{conn}} \bigwedge_{q \in Q} \left(\mathrm{sw}^t_{p,p'} \implies \left((\mathrm{mp}^{t-1}_{q,p} \iff \mathrm{mp}^t_{q,p'}) \wedge (\mathrm{mp}^{t-1}_{q,p'} \iff \mathrm{mp}^t_{q,p})\right)\right) \tag{19}$$

Assumptions

We also encode an assumption for incremental SAT solving in constraint 20: For each t exceeding the depth of the input circuit, we add this constraint expressing that no gate is delayed at the last time step, which ensures that all gates from the input circuit are scheduled. This tells the SAT solver that we expect a solution at time step t, and consequently not at any time step prior to t.

$$\bigwedge_{g \in G} \left(\mathrm{asm}^t \implies \neg \mathrm{d}^t_g\right) \tag{20}$$

3.1 Encoding variations

There are two goal variations that are easily implemented. The first one is finding optimal CX-depth instead of depth. This can be achieved by the same encoding, by preprocessing the circuit to only include CX gates. After the optimal placement of SWAPs has been found using the encoding, the unary gates can be reinserted to get the final circuit.

The other variation is finding a circuit with minimal depth and locally minimal SWAP count. After a circuit with minimal depth has been found, we count the number of SWAPs S and force it to decrease in increments of one by adding new assumption variables sasm^i for $i = S-1, \ldots, 0$. To force the number of SWAPs to be at most i, the following constraint is added to the instance: $\text{sasm}^i \implies \text{AtMostN}(i, \text{sw}^0_{p_0,p_1}, \ldots, \text{sw}^t_{p_{n-1},p_n})$, where $\text{sw}^0_{p_0,p_1}, \ldots, \text{sw}^t_{p_{n-1},p_n}$ represents all "swap" variables that have been created in total over all layers. The first i where it is not possible to solve an instance, tells us that the locally minimal number of SWAP gates is $i+1$. This is because the search space is monotone in the number of SWAP gates [15].

3.2 Count Optimal Encoding vs Depth Optimal Encoding

We have employed several effective techniques from SAT based SWAP-count optimal synthesis in [21] for a scalable depth optimal encoding in this paper. In this Section, we summarize the similarities and differences between the count optimal and depth optimal encodings.

The following list describes the techniques adapted from count optimal encoding:

- Incremental solving to reuse learned clauses from earlier bounds with simple assumptions.
- Bi-directional propagation of gate constraints using "delayed" and "advanced" variables.
- Similar cardinality and connectivity constraints in both encodings.

The following list summarizes the main differences from the count optimal encoding:

- A SWAP gate (with depth 3) can interleave with other 1-qubit and 2-qubit gates in depth optimal encoding unlike a count optimal setting. Equations 13 to 19 present the new constraints for enabling interleaving of SWAP gates.
- The makespan of a count optimal encoding corresponds to the number of SWAPs needed. However, the makespan of a depth optimal encoding is at least the depth of the original circuit. Since the circuit depth/CNOT-depth is often several times more than the number of SWAPs, scalability of encodings differs greatly.
- As discussed in Section 3.1, a depth optimal encoding also enables local SWAP count optimization with simple additional constraints. On the other hand, one cannot easily optimize local depth in a count optimal encoding.

4 Experimental evaluation

Our SAT encoding is implemented in the tool QuilLS[2]. Remember that optimizing depth is harder than optimizing count due to larger makespan. For an evaluation, we are interested in the performance penalty by depth optimization instead of count. Further, we are also interested in the quality of results and their impact on noise reduction which is the ultimate goal for circuit optimization. In particular, we structure our experiments to answer the following research questions:

[2] QuilLS is available at `https://github.com/anbclausen/quills`. It is also ported to Q-Synth v4 at `https://github.com/irfansha/Q-Synth`. The experiments reported here are obtained with QuilLS.

A How does the runtime performance of QuilLS compare to Q-Synth, TB-OLSQ2, and OLSQ2?

B How do the depth, CX-depth, SWAP count of QuilLS compare to SABRE, Q-Synth, TB-OLSQ2, and OLSQ2?

C How do different optimization strategies affect the quality of the synthesized circuits in terms of noise?

Our experiments are run on a Linux machine with a 2.4GHz CPU and 60GB of memory. We always use a timeout of 2 hours (7200 seconds). For our benchmarks, we use a set of 33 standard benchmarks (15 arithmetic and 18 QUEKO [23]) with up to 54 qubits (q) and 270 CX gates from the literature [15, 21]. Furthermore, we use 10 8q Variational Quantum Eigensolver (VQE) circuits with up to 79 CX gates from [21] representing practically applicable quantum circuits. Each of these benchmark circuits is synthesized on the following platforms: Tenerife (5q), Melbourne (14q), Guadalupe (16q), Tokyo (20q), and Cambridge (32q) by IBM; Google's Sycamore (54q); and Rigetti's Aspen-M (80q). Given a test circuit, we only synthesize it on a platform if the platform has the required number of qubits. The exact data on the circuits can be found in Section A.

For our evaluation, we use the following tools:

- QuilLS: Our (CX)-depth-optimal tool, with CaDiCaL (v1.5.3) [1] SAT solver as our backend.
- SABRE [13]: A heuristic algorithm that is part of Qiskit. We use the same experimental setting as in [21].
- Q-Synth (v2) [21]: A SAT based state-of-the-art size-optimal tool with CaDiCal (v1.5.3) as backend.
- OLSQ2 [15]: An SMT-based state-of-the-art depth-optimal tool with Z3 (v4.13.0.0) [4] as a backend.
- TB-OLSQ2 [15]: A near-optimal version of OLSQ2 with better runtime performance. We report a timeout if the tool does not terminate within the time limit.

For both OLSQ2 and TB-OLSQ2, we use the best settings, i.e., allowing them to use SABRE to get an upper bound.

All measurements are available on Zenodo [9][3] and an overview of the data is given in Appendix B in the full technical report [8]. Next, we will discuss the setup and the results of each experiment in detail.

4.1 Performance of QuilLS Compared to Existing Tools

Experimental setup

We are interested in both depth and CX-depth optimization. As mentioned in Section 3.1, we can optimize either depth or CX-depth in our SAT encoding. We also ensure that QuilLS always computes locally minimal SWAPs. The OLSQ2 and TB-OLSQ2 tools optimize depth, and we can make them optimize CX-depth instead, by removing all unary gates from the input circuit before passing it to these tools. For a fair comparison, we set options such that OLSQ2 and TB-OLSQ2 also compute locally minimal SWAP count. We run SABRE, Q-Synth and the (CX)-depth-optimal versions of (TB)-OLSQ2 and QuilLS on all the benchmarks on every platform and measure running time.

[3] Full code and data is available at `https://doi.org/10.5281/zenodo.15606051`

Results

First, we focus on depth optimization for tools QuilLS and OLSQ2. The times for computing a heuristic solution with SABRE are excluded since they are all smaller than 0.5 seconds. Figure 3 shows the running times comparison of QuilLS and the other tools. The full data set with running times for all tools can be found in the appendix.

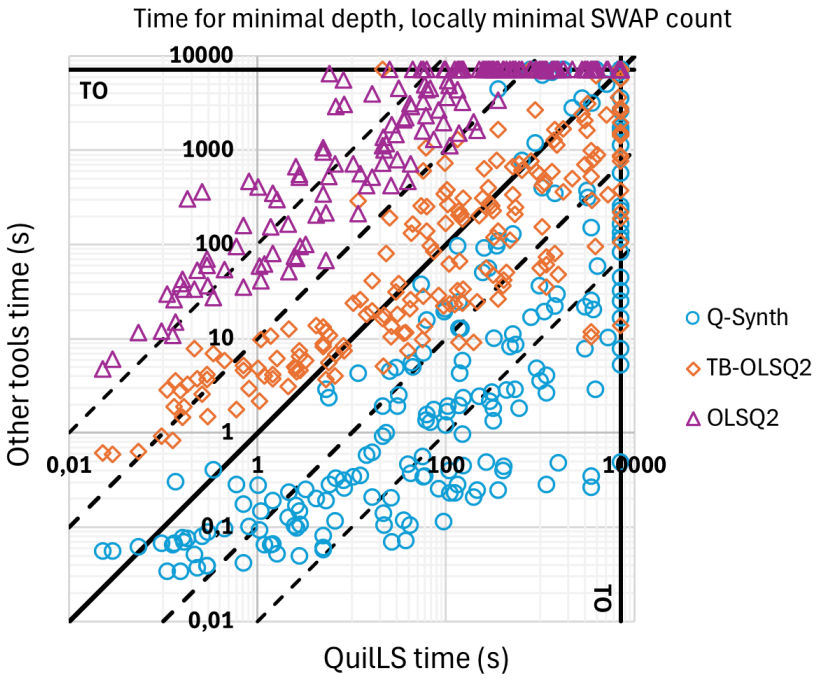


Figure 3 Time comparison between QuilLS and other tools (with their own optimization metrics). A mark on the bold diagonal line shows equal performance between the tools, a mark above shows better performance by QuilLS, and a mark below shows worse performance by QuilLS than the other tool. Dotted diagonal lines indicate performance ratios of 10 and 100. Bold horizontal and vertical lines indicate timeouts (TO).

In Table 2 the number of solved instances, and time-outs are shown for each tool/metric. We also include the results with CX-depth optimization by QuilLS and OLSQ2.

QuilLS vs OLSQ2 and TB-OLSQ2

OLSQ2 is our direct competitor that optimizes the same metric, i.e., depth and locally minimal SWAP count. From the results, it is clear that we significantly outperform OLSQ2. OLSQ2 is almost always at least 10 times slower than QuilLS, and sometimes up to 3 orders of magnitude slower. TB-OLSQ2 on the other hand, being a near-optimal tool performs better on circuits with larger synthesis times. This is consistent with the observations between optimal and near-optimal synthesis in the literature [15].

QuilLS vs Q-Synth

Q-Synth is generally 10 times faster than QuilLS and more than 100 times faster on many circuits. Despite being similar encodings and using the same SAT solver as a backend, we see the hardness of optimizing depth compared to SWAP count. Interestingly, on some circuits with large synthesis times Q-Synth uses more time than QuilLS.

Table 2 Timeouts (TO), and solved instances for QLS tools.

Tool	TO	Solved	Solved %
SABRE	0	220	100%
TB-OLSQ2 (CX-depth)	26	194	88%
TB-OLSQ2 (depth)	30	190	86%
Q-Synth (SWAP count)	31	189	86%
QuilLS (CX-depth)	35	185	84%
QuilLS (depth)	48	172	78%
OLSQ2 (CX-depth)	102	118	54%
OLSQ2 (depth)	125	95	43%

Depth vs CX-depth optimization in QuilLS

It is faster in general to compute a CX-depth-optimal circuit than a depth-optimal circuit. This is expected, since the instances are smaller. This is also the case for (TB)-OLSQ2, and the performance for CX-depth-optimal synthesis gives a picture very similar to Figure 3.

Q-Synth, TB-OLSQ2, and the CX-depth-optimal version of QuilLS solve 84-88% of instances, while the depth-optimal version of QuilLS solves 78% of instances. OLSQ2 solves around 50% of instances. SABRE solves every instance within a second. However, in the next experiment we will see that the quality of mappings from SABRE are indeed far from optimal.

4.2 Quality of Results for QuilLS Compared to Existing Tools

In the previous experiment, we have seen a performance penalty on optimizing depth instead of SWAP count. In this experiment, we look at the quality of the results reported by all the tools. We are interested in finding a trade-off between performance and quality of results. So far, we have considered the two metrics SWAP count and depth with locally optimal SWAP count. Since it is quite cheap to compute SWAP optimal mappings, one could also consider optimizing depth for the optimal SWAP count. To do this, one can use e.g. Q-Synth to find the optimal SWAP count, and then use our tool QuilLS for optimizing depth given a bound on SWAP count (we call this combination "Q-S+Qui"). The experimental setup is the same as for the previous section, except that we include this new variant: size optimal with locally minimal (CX)-depth. We now measure the depth, CX-depth, and SWAP count for each synthesized circuit.

Results

Table 3 shows for each tool and optimization metric the average ratio of the depth, CX-depth and CX count compared to optimal tools, over a set of benchmarks that all tools could solve (93 for all tools, 129 if we discard OLSQ2). We also show the success rate (number of solved circuits over all 220 benchmarks). We report the number of CX gates in the synthesized circuit rather than the SWAP gates, since the CX gate count also reflects the size of the original circuit. We set the tool and metric combination that guarantees optimal values to **1.00** and provide the ratios of the results from other tools. A higher ratio means a worse result.

Table 3 Average ratios between depth (d), CX-depth (CX-d) and CX gate count (CX) of circuits mapped by various tool/metric combinations. Bold numbers indicate guaranteed optimality. 93 circuits are solved by all tools; 129 circuits are solved by all tools except OLSQ2.

			Avg. over N=129			Avg. over N=93		
Tool/metric/%success			d.	CX-d	CX	d.	CX-d	CX
SABRE	heur	100	1.48	1.46	1.15	1.29	1.27	1.10
TB-O2	CX-d	88	1.09	1.06	1.01	1.08	1.06	1.01
TB-O2	d.	86	1.08	1.07	1.01	1.08	1.07	1.01
Q-Synth	CX	86	1.06	1.05	**1.00**	1.04	1.03	**1.00**
QuilLS	CX-d	84	1.03	**1.00**	1.04	1.02	**1.00**	1.03
QuilLS	d.	78	**1.00**	1.03	1.09	**1.00**	1.02	1.04
Q-S+Qui	CX-d	67	1.04	1.01	**1.00**	1.03	1.01	**1.00**
Q-S+Qui	d.	60	1.02	1.02	**1.00**	1.01	1.02	**1.00**
OLSQ2	CX-d	54	–	–	–	1.02	**1.00**	1.03
OLSQ2	d.	43	–	–	–	**1.00**	1.01	1.04

Discussion

For all three measures, SABRE gets the worst ratios, which is to be expected since SABRE is a heuristic tool; it is also the only tool that can map all instances. Comparing N=129 vs N=93, it is also clear that for harder instances, SABRE inserts relatively more SWAP gates.

Only OLSQ2 and QuilLS guarantee minimal (CX)-depth. Q-Synth and TB-OLSQ2 get (CX)-depths that are less than 10% off from the optimal depths. Q-Synth (including Q-S+Qui) guarantees optimal CX-count. QuilLS (optimizing for depth) is on average 9% off from the CX-count optimum (for N=129). It performs the same on CX-count as OLSQ2 (for N=93). The other tool combinations have 1-4% worse CX-count than Q-Synth, except the heuristic tool SABRE (15% off).

We conclude that QuilLS finds 5-9% better depths and CX-depths on average than the optimal tools with different optimization goals, while finding 46% better (CX)-depths on average than SABRE. This comes at the cost of larger CX-counts than other tools. We observe that first optimizing SWAP count and then depth produces very good results overall.

4.3 Impact of Optimization Metrics on Noise Reduction

Our ultimate motivation for circuit optimization is noise reduction. The noise in a quantum circuit comes from noisy gates, interference from the environment, and measurement errors. One way to evaluate the impact of these errors is by running the same circuit many times and comparing the distribution of the measurement outcomes to the expected distribution. These distributions can be obtained, for instance by Qiskit's simulator with and without noise models.

Each quantum platform in the Qiskit library comes with a noise model, providing noise probabilities for native gates. To use these, we first transpile our circuits to IBM's native gates, using Qiskit (without optimization).

The noise profile for gates varies significantly between the different qubits of the platform. This poses a problem for us: The same synthesized circuit with different initial mappings would have very different levels of noise. Noise models are represented as a set of probability distributions over quantum operations, so we created an averaged version of each noise model by averaging the noise distribution per operation over all qubits. In this way, the noise model is still realistic, but does not depend on the initial mapping.

Qiskit's noise models specify gate errors and measurement errors, but not the interference from the environment when qubits are idle (also called decoherence). To incorporate this, we use the same approach as in [12]: Qiskit provides a noise distribution for the `I`-gate, which is meant to model decoherence [18]. Hence, before simulating a circuit, we insert I-gates for all idle positions in that circuit.

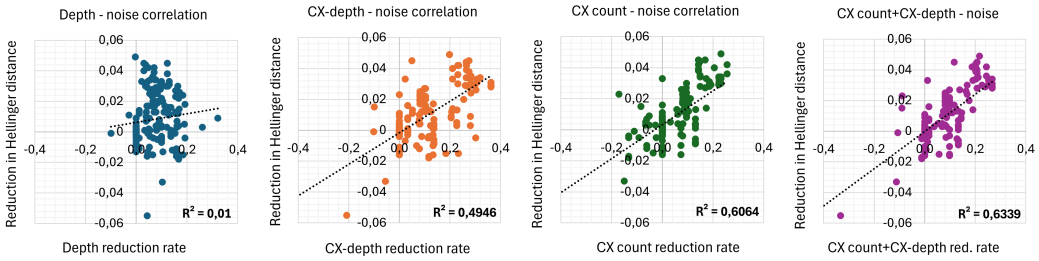
We apply all the different QLS tools (with the variations discussed in earlier experiments) to the transpiled circuits, and simulate the mapped circuits under the averaged noise model.

Experimental setup

We simulate each synthesized circuit 100,000 times with and without noise. When simulating a circuit, we measure the mapped qubits at the very end of the circuit to obtain a probability distribution. We use Qiskit to compute the Hellinger distance between the two distributions obtained from simulations with and without the noise model. This is standard in the literature [3, 6, 19]. Given the resource intensity of simulation, we only use the medium-sized platform IBM Guadalupe (16q) with the standard and VQE benchmarks.

Results

Figure 4 shows the correlations between reduction rates for depth, CX-depth, CX count, and the sum of CX-depth and CX count and the reduction in the Hellinger distance as a measure for noise. We consider SABRE as baseline and report reductions in various tools. The depth reduction rate for tool X is normalized as $(d(SABRE) - d(X))/d(SABRE)$, and similar for the other metrics. This presentation and analysis style follows the example of [25].



(a) Noise coefficient 1%. **(b)** Noise coefficient 49%. **(c)** Noise coefficient 61%. **(d)** Noise coefficient 63%.

Figure 4 Noise correlation coefficients between reduction rate for the optimization goal and reduction in Hellinger distance.

Discussion

Figure 4 shows that the depth reduction rate is not correlated with noise reduction at all, while the CX-depth reduction rate and the CX gate count reduction rate are quite strongly correlated. Surprisingly, CX gate count reduction correlates more with noise than CX-depth reduction. The correlation is strengthened further when looking at the sum of CX-depth and CX gate count.

This result is surprising since it is generally assumed that noise scales with depth [25, 12, 14, 15]. For example, the authors of [14] stress the importance of considering unary gates when minimizing depth. In [22], it is argued that minimizing CX-depth provides a better noise reduction than minimizing depth, since the binary CX gates are more noisy. But seeing no correlation between depth and noise is unexpected.

We conclude that reducing CX gate count and CX-depth reduces noise in output circuits. Further, CX gate count correlates better than CX-depth. CX count + CX-depth provides the best correlation so far. Surprisingly, we see no correlation between reducing depth and reducing noise.

5 Conclusion

In this paper, we have presented an efficient SAT encoding of (CX)-depth-optimal quantum layout synthesis. The SAT encoding has variations that allow ancillary SWAPs and finding locally minimal SWAP counts. We have implemented these encodings in the tool QuilLS and tested them on standard benchmarks. We compare QuilLS to the state-of-the-art depth-optimal tool, OLSQ2 [15], its near-optimal variant TB-OLSQ2, the state-of-the-art size optimal tool, Q-Synth [21], and a heuristic, fast but sub-optimal tool, SABRE [13]. QuilLS and OLSQ2 synthesize circuits with better (CX)-depths than the other tools, and QuilLS vastly outperforms the previous best depth-optimal tool OLSQ2, by solving more instances, and often being more than 10-100x faster.

We also evaluate the amount of noise in the circuits synthesized by each tool, by measuring the Hellinger distance between ideal and noisy simulations of the circuits. We see that reducing the number of SWAPs and reducing CX-depth (and in particular their combination) is strongly correlated with reducing noise. Contrary to what is generally assumed in the literature, we do not see a correlation between reducing depth and reducing noise. However, this should be explored further, by testing whether these results carry over to larger quantum platforms and when executing the circuits on real quantum computers.

References

1 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`.

2 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`.

3 Samudra Dasgupta and Travis S. Humble. Characterizing the reproducibility of noisy quantum circuits. *Entropy*, 24(2):244, February 2022. `doi:10.3390/e24020244`.

4 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`.

5 M. R. Garey, D. S. Johnson, and R. Endre Tarjan. The planar Hamiltonian circuit problem is NP-complete. *SIAM Journal on Computing*, 5(4):704–714, 1976. `doi:10.1137/0205049`.

6 Konstantinos Georgopoulos, Clive Emary, and Paolo Zuliani. Modeling and simulating the noisy behavior of near-term quantum computers. *Physical Review A*, 104(6), December 2021. `doi:10.1103/physreva.104.062432`.

7 Yuichi Hirata, Masaki Nakanishi, Shigeru Yamashita, and Yasuhiko Nakashima. An efficient conversion of quantum circuits to a linear nearest neighbor architecture. *Quantum Inf. Comput.*, 11:142–166, 2011. `doi:10.26421/QIC11.1-2-10`.

8 Anna B. Jakobsen, Anders B. Clausen, Jaco van de Pol, and Irfansha Shaik. Depth-Optimal Quantum Layout Synthesis as SAT (full version). *CoRR*, 2025. `arXiv:2506.06752`.

9 Anna Blume Jakobsen, Anders B. Clausen, J.C. van de Pol, and Irfansha Shaik. Code, benchmarks and data for "depth-optimal quantum layout synthesis as sat" paper at sat25, June 2025. `doi:10.5281/zenodo.15606051`.

10 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.

11 M. S. Krishnamoorthy. An NP-hard problem in bipartite graphs. *SIGACT News*, 7(1):26, January 1975. doi:10.1145/990518.990521.

12 Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. QASMBench: A low-level quantum benchmark suite for NISQ evaluation and simulation. *ACM Transactions on Quantum Computing*, 4(2), February 2023. doi:10.1145/3550488.

13 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.

14 Sanjiang Li, Ky Dan Nguyen, Zachary Clare, and Yuan Feng. Single-qubit gates matter for optimising quantum circuit depth in qubit mapping. In *ICCAD*, pages 1–9. IEEE, 2023. doi:10.1109/ICCAD57390.2023.10323863.

15 Wan-Hsuan Lin, Jason Kimko, Bochen Tan, Nikolaj Bjørner, and Jason Cong. Scalable optimal layout synthesis for NISQ quantum processors. In *DAC'23*, pages 1–6, July 2023. doi:10.1109/DAC56929.2023.10247760.

16 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.

17 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

18 Qiskit contributors. Qiskit documentation, 2023. URL: https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.IGate.

19 Salonik Resch and Ulya R. Karpuzcu. Benchmarking quantum computers and the impact of quantum noise. *ACM Comput. Surv.*, 54(7):142:1–142:35, 2022. doi:10.1145/3464420.

20 Irfansha Shaik and Jaco van de Pol. Optimal layout synthesis for quantum circuits as classical planning. In *2023 IEEE/ACM International Conference on Computer Aided Design*, ICCAD'23. IEEE, 2023. doi:10.1109/ICCAD57390.2023.10323924.

21 Irfansha Shaik and Jaco van de Pol. Optimal layout synthesis for deep quantum circuits on NISQ processors with 100+ qubits. In *SAT'24*, volume 305, pages 26:1–18. LIPIcs, 2024. doi:10.4230/LIPIcs.SAT.2024.26.

22 Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket⟩: a retargetable compiler for nisq devices. *Quantum Science and Technology*, 6(1):014003, November 2020. doi:10.1088/2058-9565/ab8e92.

23 Bochen Tan and Jason Cong. Optimality study of existing quantum computing layout synthesis tools. *IEEE Transactions on Computers*, 70(9):1363–1373, 2021. doi:10.1109/TC.2020.3009140.

24 Robert Wille, Lukas Burgholzer, and Alwin Zulehner. Mapping quantum circuits to IBM QX architectures using the minimal number of SWAP and H operations. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19. ACM, June 2019. doi:10.1145/3316781.3317859.

25 Xin-Chuan Wu, Marc Grau Davis, Frederic T. Chong, and Costin Iancu. Qgo: Scalable quantum circuit optimization using automated synthesis. *CoRR*, 2022. arXiv:2012.09835.

26 Jiong Yang, Yaroslav A. Kharkov, Yunong Shi, Marijn J. H. Heule, and Bruno Dutertre. Quantum Circuit Mapping Based on Incremental and Parallel SAT Solving. In *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:18, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SAT.2024.29.

27 Alwin Zulehner, Alexandru Paler, and Robert Wille. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 38(7):1226–1236, 2019. doi:10.1109/TCAD.2018.2846658.

A Benchmarks Information

The circuit data for all the circuits used in the experiments can be found in Table 4.

Table 4 Circuit data for all standard, VQE and QUEKO benchmarks.

Circuit name	Qubits	Gates	CX gates	Depth	CX-depth
4gt13_92	5	66	30	38	26
4mod5-v1_22	5	21	11	12	10
adder	4	23	10	11	6
barenco_tof_4	7	72	34	68	34
barenco_tof_5	9	104	50	95	48
mod_mult_55	9	91	40	47	25
mod5mils_65	5	35	16	21	16
or	3	17	6	8	6
qaoa5	5	22	8	14	8
qft_8	8	106	56	42	26
rc_adder_6	14	140	71	83	40
tof_4	7	55	22	46	21
tof_5	9	75	30	61	28
toffoli	3	15	6	11	6
vbe_adder_3	10	89	50	58	33
vqe_8_0_5_100	8	96	52	79	44
vqe_8_0_10_100	8	129	63	92	51
vqe_8_1_5_100	8	43	18	32	16
vqe_8_1_10_100	8	100	47	76	41
vqe_8_2_5_100	8	90	48	80	44
vqe_8_2_10_100	8	153	79	136	75
vqe_8_3_5_100	8	76	40	61	35
vqe_8_3_10_100	8	151	78	119	68
vqe_8_4_5_100	8	76	39	60	32
vqe_8_4_10_100	8	136	71	102	56
16QBT_05CYC_TFL_0	16	37	15	5	5
16QBT_10CYC_TFL_0	16	73	29	10	7
16QBT_15CYC_TFL_0	16	109	44	15	11
16QBT_20CYC_TFL_0	16	145	58	20	14
16QBT_25CYC_TFL_0	16	180	72	25	15
16QBT_30CYC_TFL_0	16	217	87	30	18
16QBT_35CYC_TFL_0	16	253	101	35	25
16QBT_40CYC_TFL_0	16	289	116	40	27
16QBT_45CYC_TFL_0	16	325	130	45	30
54QBT_05CYC_QSE_0	54	192	54	5	5
54QBT_10CYC_QSE_0	54	384	108	10	10
54QBT_15CYC_QSE_0	54	576	162	15	12
54QBT_20CYC_QSE_0	54	767	216	20	16
54QBT_25CYC_QSE_0	54	959	270	25	21
54QBT_30CYC_QSE_0	54	1151	324	30	24
54QBT_35CYC_QSE_0	54	1342	378	35	32
54QBT_40CYC_QSE_0	54	1534	432	40	33
54QBT_45CYC_QSE_0	54	1727	487	45	38

Core-Guided Linear Programming-Based Maximum Satisfiability

George Katsirelos
Université Paris-Saclay, AgroParisTech, INRAE, UMR MIA Paris-Saclay, 91120, Palaiseau, France

Abstract

The core-guided algorithm OLL is the basis of some of the most successful algorithms for MaxSAT in recent evaluations. It works by iteratively finding cores of the formula and transforming it so that it exhibits a higher lower bound. It has recently been shown to implicitly discover cores of the original formula, as well as a compact representation of its reasoning within a linear program. In this paper, we use and extend these results to design a practical MaxSAT solver. We show an explicit linear program which matches and usually exceeds the bound computed by OLL. We show that OLL can be restated as an algorithm that explicitly computes a feasible dual solution of this linear program. This restated algorithm naturally works with an arbitrary dual solution. It can in fact be used to improve *any* LP representation of the MaxSAT instance. This presents a large increase of the potential design space for such algorithms. We describe some potential improvements from this insight and show that an implementation outperforms the state of the art algorithms on the set of instances from the latest MaxSAT evaluation.

2012 ACM Subject Classification Theory of computation → Discrete optimization; Mathematics of computing → Combinatorial optimization; Theory of computation → Logic; Mathematics of computing → Solvers

Keywords and phrases maximum satisfiability, core-guided solvers, linear programming

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.17

Supplementary Material *Software (Source Code)*: `https://github.com/gkatsi/OLLLP` archived at `swh:1:dir:85dd891d9fd590dabd0fb6c2900c1877e0b17b81`

Funding Part of this work was funded by the AI Interdisciplinary Institute ANITI. ANITI is funded by the French "Investing for the Future – PIA3" program under the Grant agreement n°ANR-23-IACL-0002.

1 Introduction

MaxSAT is the optimization variant of the Boolean propositional satisfiability problem. The performance of MaxSAT solvers has increased significantly in the last few years, as evidenced by the results of recent MaxSAT evaluations. There exist two main classes of algorithms: implicit hitting set (IHS) solvers [9, 11, 10, 4, 5] and core-guided solvers [14, 3, 21, 20, 19, 15]. However, despite the improvement in performance over the last few years, there are few new algorithms. The main core-guided algorithm, OLL [1, 19], was introduced more than 10 years ago.

In this paper, we do not quite propose a completely different algorithm. We propose a new algorithm, OL^{3}P, which is quite similar to OLL. However, it uses a linear program as its main reasoning engine for bounds. The flexibility of the linear program allows us to ignore the subtleties of figuring out a correct reformulation and rely on an LP solver to do it. This means that the OL^{3}P algorithm can accommodate constraints coming from different contexts, which opens up possibilities for further algorithmic development. Specifically, our contributions are:

- We show an LP that can capture the lower bound computed by OLL. More formally, it admits a dual solution which gives the same lower bound. While the existence of this was known [17], we give here an explicit and compact LP.


© George Katsirelos;
licensed under Creative Commons License CC-BY 4.0



- We discuss the necessary condition for using an LP to compute bounds in this algorithmic framework, namely that the LP is in augmented form.
- We discuss the subtleties and surprises of using LP reduced costs to reformulate MaxSAT instances.
- We show how to integrate a new source of constraints, by seeding the LP with clauses from the original formula.
- We integrate all these into a solver that clearly outperforms the previous state of the art of core-guided MaxSAT solving.

2 Background

A propositional formula F in conjunctive normal form (CNF) is a conjunction of clauses, where each clause c is a disjunction of literals. A literal is either a propositional variable or its negation. We write $vars(c)$ for the set of variables whose literals appear in a clause c and $vars(F)$ for the union of $vars(c)$ over all the clauses of F. Given an assignment A of variables to true or false, a clause c is satisfied by an assignment if it maps at least one of its literals to true, which we write $A \models c$. Otherwise it is falsified. The CNFSAT (or simply SAT) problem asks to determine whether there exists an assignment such that all clauses are satisfied.

The Weighted Partial MaxSAT (WPMS) problem is an extension of SAT to optimization. A WPMS instance is a tuple $\langle H, S, w\rangle$ where H is a set of *hard* clauses, S is a set of *soft* clauses and $w : S \to \mathbb{R}_{\geq 0}$ is a weight function mapping soft clauses to non-negative weights. The cost of an assignment A, written $w(A)$ is infinite if it falsifies any of the hard clauses, otherwise it is the sum of the weights of the soft clauses it falsifies: $\sum_{c \in S, A \not\models c} w(c)$. The objective of WPMS is to find an assignment whose cost is finite and minimum among all assignments.

Equivalently, we can define WPMS as a pair $\langle H, w\rangle$, where H is a set of hard clauses and w is a function $w : vars(H) \to \mathbb{R}_{\geq 0}$. Alternatively, we can write w as an n-dimensional vector, where n is the number of variables, so that we can write the WPMS problem as

$$\begin{aligned} & \min w^T x \\ s.t. & \\ & x \models H \end{aligned}$$

This restatement of WPMS is consistent with recent work [17, 16]. Moreover, it is equivalent to the formulation with soft clauses. To convert from soft clauses to this formulation, we can reify each soft clause using so-called *blocking* variables, i.e., introduce a fresh variable $b \iff c$ for each soft clause $c \in S$ and set the cost of b to be the weight of c. This is what most WPMS solvers do anyway in order to use assumption-based SAT solvers.

A core C of a WPMS is a subset of its soft clauses such that the CNF formula $C \cup H$ is unsatisfiable. In terms of WPMS solving, this means that at least one of the soft clauses in C must be satisfied. In our preferred formulation of WPMS, where an instance is $\langle H, w\rangle$, a core is a set of literals such that at least one of these literals must be true in any feasible solution, i.e., any solution that satisfies H. In other words, a core is a positive clause over blocking variables.

As we shall see later, it is useful to ignore the correspondence between literals of blocking variables and soft clauses. Indeed, the "cores" extracted by OL^{3}P need not contain exclusively positive literals of blocking variables. In fact, both of these conditions are violated: the clauses extracted by OL^{3}P may contain literals of any sign; moreover, those may be literals

of blocking variables, of sum variables [1], and even of variables of the original formula that are not blocking variables. Despite the difference with standard terminology, we use the term core throughout, as it is the main piece of information given by the SAT solver to the MaxSAT algorithm. We emphasize however, that in our setting a core is simply a clause or, equivalently, a linear inequality.

2.1 OLL

OLL is an algorithm for solving MaxSAT that belongs to a class of iterative SAT-based algorithms known as core-guided. It was introduced by Andres et al. [1] in the context of ASP and adapted to MaxSAT by Morgado et al. [19]. Its pseudocode is shown in Algorithm 1.

OLL optimistically tries to find a solution with 0 cost at each iteration, by asking a SAT solver to solve the CNF instance that results from setting all literals in the objective function to false (line 5). If that succeeds, this is optimal and it returns in line 7. If it fails, it extracts a core from that formula and uses it to increase the lower bound by c^i, the minimum cost of any variable in the core (line 8). It then reformulates the instance in line 9 so that each feasible assignment of F has its cost reduced by exactly c^i in F^{i+1}. The reformulation involves introducing new variables in the objective function, modifying the costs of existing variables, as well as building a subformula f^i_{OLL} which logically defines the new variables. As long as the reformulation step is correct, OLL increases the lower bound at each iteration, hence it is guaranteed to eventually reach the true optimum and terminate.

In order to avoid confusion between cores of the original formula and cores of the reformulated formulas - which can be different - we follow [22] and refer to cores of $H \cup f^i_{OLL}$ as metas.

Algorithm 1 Core-guided MaxSAT.

```
1 Procedure Core-Guided(F = ⟨H, w⟩)
2     lb = 0
3     f^0_OLL = ∅
4     for iteration i = 1, ... do
5         (a, m^i) = solve((H ∪ f^{i-1}_OLL) |_{w=0})
6         if a ≠ ∅ then
7             return a
8         lb ← lb + min{w^{i-1}(j) | x_j ∈ m^i}
9         (f^i_OLL, w^i) = Reformulate(f^{i-1}_OLL, w^{i-1}, m^i)
```

We complete the description of OLL with a description of the reformulation step. Let the meta at iteration i be $m^i = \{x^i_1, x^i_2, \ldots, x^i_{k^i}\}$. By the definition of $F^i|_{w=0}$, all literals that appear in m^i have positive cost. We abuse notation and write $w^{i-1}(m^i) = \min_{x \in m^i} w^{i-1}(x)$ for the minimum cost among the literals that appear in m^i. OLL increases the lower bound by $w^{i-1}(m^i)$ and introduces *sum variables*, defined as $o^i_j \iff \sum_{x \in m^i} \geq j$ for $2 \leq j \leq k^i$. Finally, it reduces by $w^{i-1}(m^i)$ the cost of all literals in m^i and sets to $w^{i-1}(m^i)$ the cost of the positive literals of all o^i_j. Formally,

[1] Sum variables are introduced by OLL and will be explained in section 2.1

$$w^i(x) = \begin{cases} w^{i-1}(x) - w^{i-1}(m^i) & \text{if } x \in m^i \\ w^{i-1}(m^i) & \text{if } x \in \{o^i_2, \ldots, o^i_{k^i}\} \\ w^{i-1}(x) & \text{otherwise} \end{cases}$$

To see why this is a reformulation, observe that we can combine the fact that $\sum_{x \in m^i} x \geq 1$ with the definition of the sum variables to get

$$\sum_{x \in m^i} x \quad = \quad 1 + \sum_{j=2}^{k^i} o^i_j \ , \tag{1}$$

because when d of the k^i variables in m^i are true, exactly the $d-1$ sum variables $o^i_2, \ldots, o^i_d$ will be true, by their definition. To complete the reformulation, we multiply equation (1) by $w^{i-1}(m^i)$ and replace its left-hand side in w^{i-1} by its right hand side, which gives w^i. Note that this differs from the usual proof of correctness of OLL, but is useful to better understand the new algorithm OL^{3}P.

2.2 Linear programming

Linear programs are problems of the form $\min c^T x : Ax = b \wedge x \in \mathbb{R}^n_{\geq 0}$, where x is a vector of n real (rational) valued variables, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$. The problem can be solved in polynomial time. We restrict our attention here to the case where $x \in [0,1]^n$.

This form of linear programs, where all constraints are equalities, is called the augmented form. It is possible to convert any linear program which includes inequalities by introducing *slack variables*. For example the constraint $a^T x \geq b$ becomes $[a \ -1]^T [xs] = b$, where s is a fresh variable and is the slack variable of this constraint.

For each linear program P (called the *primal*), we can define a *dual* problem $D(P)$, which is $\max b^T y : A^T y \leq c \wedge y \in \mathbb{R}^m$, where y is a vector of m real variables, called the dual variables or *dual multipliers*. An important theorem in linear programming is *strong duality*: the optima of P and $D(P)$ are exactly the same. In fact, for every feasible solution y of $D(P)$, its cost is a lower bound on the optimum of P.

There exists a correspondence between variables of the primal and constraints of the dual. Indeed, the i^{th} column of P contains the coefficients of the i^{th} primal variable and also describes the i^{th} dual constraint. From each dual solution y, we can define the *reduced cost* of each primal variable, $rc^y(x)$. This is the slack of the corresponding dual constraint, i.e., $c_i - A_i^T y$. The theorem of *complementary slackness* states that for every pair of optimal solutions x, y, it holds $x_i(c_i - A_i^T y) = 0$, i.e., if the reduced cost of x is non-zero, then x must be 0 in the primal and vice versa.

Reduced costs are used typically for *reduced cost fixing* when the linear program is a relaxation of a discrete optimization problem. For a given dual solution y, $c^T y + c_i - A_i^T y$ is a lower bound on the cost of any solution that assigns x_i a non-zero value. If that matches or exceeds the cost of an incumbent solution of the discrete problem, we can fix x_i to 0. By convention, if a variable is fixed to its 1 in the primal, its reduced cost is non-positive and $c^T y - c_i + A_i^T y$ is a lower bound on the cost of any solution that assigns x_i less than 1 [2].

[2] Negative reduced costs are not true reduced costs. The would imply negative slack of the corresponding dual constraint, which means a non-feasible dual solution. Instead, they are the negated reduced cost of a slack variable and LP solvers use this convention to avoid exposing this slack variable.

In the case of augmented form linear programs, we can make a stronger statement on reduced costs: for any feasible y, the function $b^T y + (rc^y)^T x$ is equal to $c^T x$ in every feasible point of the primal. We can therefore reformulate P using the reduced costs. We give an elementary proof of this fact below.

▶ **Theorem 1.** *Let P be a linear program $\min c^T x : Ax = b$. For a given dual feasible solution y, let $P' = \min b^T y + (rc^y)^T x : Ax = b$. Then $P = P'$, i.e., they have identical sets of feasible solutions and each feasible solution has the same cost in P and P'.*

Proof. Since the constraints of both problems are identical, we only need to show that the objectives are identical, subject only to $Ax = b$.

$$b^T y + (rc^y)^T x = b^T y + (c^T - A^T y)^T x = c^T x + b^T y - y^T Ax \tag{2}$$

Finally, since $Ax = b$, we have $y^T Ax = y^T b = b^T y$, so $b^T y + (rc^y)^T x = c^T x$, as required. ◀

▶ **Example 2.** Consider the following pair of primal and dual LPs:

(Primal)	(Dual)
$\min x_1 + x_2$	$\max y$
s.t.	
$x_1 + x_2 \geq 1$	$y \geq 0$
$x_1 \geq 0$	$y \leq 1$
$x_2 \geq 0$	$y \leq 1$

The optimum of this is 1, witnessed by the primal solution $x_1 = 1$, $x_2 = 0$ and the dual solution $y = 1$. The reduced cost of both x_1 and x_2 under this dual solution is 0, since the corresponding dual constraint for both is $y \leq 1$, which has slack 0. Suppose that we try to use reduced costs as described in theorem 1. This gives the objective function $\min 1$, which is constant. This is not equal to the objective $\min x_1 + x_2$ at all points. For example, the feasible primal assignment $x_1 = x_2 = 1$ has cost 2 under the original objective. Therefore, the scheme of theorem 1 is not a reformulation scheme for LPs not in augmented form.

Suppose now that we convert this LP to augmented form by adding the slack variable s in the sole constraint:

(Primal)	(Dual)
$\min x_1 + x_2$	$\max y$
s.t.	
$x_1 + x_2 - s = 1$	
$x_1 \geq 0$	$y \leq 1$
$x_2 \geq 0$	$y \leq 1$
$s \geq 0$	$-y \leq 0$

This has the same optimum, witnessed by the primal solution $x_1 = 1, x_2 = 0, s = 0$ and the dual solution $y = 1$. The reduced costs of x_1 and x_2 are still 0, but the reduced cost of s is 1. Therefore the scheme of theorem 1 gives the objective $\min 1 + s$, which we can confirm preserves the costs of all feasible primal assignments. Indeed, the assignment $x_1 = x_2 = 1$ above corresponds to the assignment with $x_1 = x_2 = s = 1$ in the augmented LP. This evaluates to 2 under the new objective, as it does under the original objective. ⌟

3 OL^{3}P

In this section, we describe OL^{3}P. First, we note that it has been previously shown [17] that at iteration i of a run of OLL, we can construct an LP from the metas $m^1, \ldots, m^i$, which is logically equivalent to f^i_{OLL} and whose optimum is at least the lower bound computed by OLL. However, that LP is only given implicitly, as the local polytope of a weighted CSP with desirable properties. Here, we give an explicit construction for such an LP. It has the same size asymptotically - linear in the size of the meta - but it is smaller. A similar construction has been proposed by Berg et al. in the context of proof logging for OLL [6].

A crucial requirement is for the LP to be in augmented form, i.e., with equalities only, so that dual solutions give reformulations. Our starting point is equation (1):

$$LP_{OLL}(m^i) \equiv \sum_{x \in m^i} x = 1 + \sum_{j=2}^{k^i} o^i_j \tag{3}$$

$$\cup \quad o^i_j - o^i_{j+1} - e^i_j = 0 \qquad \forall j \in [2, k^i - 1] \tag{4}$$

$$\cup \quad e^i_{k^i} = o^i_{k^i} \tag{5}$$

We name y^i_{sum} the dual multipliers corresponding to constraint (3) for m^i.

▶ **Theorem 3.** *For any execution of* OLL*, at its i^{th} iteration, there exists an LP which (a) has size linear in f^i_{OLL}, (b) is logically equivalent to f^i_{OLL}, (c) has optimum that is at least as great as the lower bound computed by* OLL*, (d) admits a dual feasible solution such that the reduced costs match the objective function computed by* OLL*.*

Proof. The LP is given by constraints (3)–(5).

(a) is obvious, as the total number of non-zeros is $4k_i - 3$: each of the k_i literals of m^i appears once, each of the $k^i - 1$ sum variables appears twice, and each of the $k^i - 1$ equality variables appears once.

(b) For (3), the fact that it follows from f^i_{OLL} was explained in section 2. The constraints (4)–(5) encode that $o^i_j \geq o^i_{j+1}$. The variable e^i_j is a slack variable that also encodes the fact that $\sum_{x \in m^i} x = j$. Indeed, if $e^i_j = 1$ then $o^i_j = 1$ and $o^i_{j+1} = 0$, so the sum is exactly j. If $e^i_j = 0$, then $o^i_j = o^i_{j+1}$, so the sum is either strictly less or strictly greater than j. The last constraint (5) exists only for uniformity. The proof that these entail the SAT encoding of the sum constraints is similar.

(c), (d). Set the dual variable y^i_{sum} for each iteration i to $w^{i-1}(m^i)$ and every other dual variable to 0. We show that this has the required properties by induction. For the first meta, m^1, $w^0(m^1)$ is at most equal to $w^0(x)$ for the original variables $x \in m^1$ and they appear only in a single new constraint, constraint (3) for m^1. Hence the dual constraint of each variable $x \in m^1$ is $y^1_{sum} \leq w^0(x)$, which is satisfied. Moreover, its reduced cost under this dual assignment is $w^0(x) - w^0(m_i)$. For the sum variables the dual constraint is $-y^1_{sum} \leq w^0(x) \leq 0$, which is satisfied since $w^0(m_i)$ is positive and the reduced cost is exactly $w^0(m_i)$, as required.

For the inductive step, at iteration i observe that the only change made to the dual with the addition of the new constraint is that its dual variable y^i_{sum} is added to the dual constraint of each $x \in m^i$. By the inductive hypothesis, these constraints have slack equal to the cost assigned to the corresponding primal variables by OLL. Therefore, by setting $y^i_{sum} = w^{i-1}(m^i)$, we see with the same reasoning as above that all dual constraints are satisfied and the reduced costs are set as OLL. ◀

▶ **Example 4.** Suppose that OLL runs on an instance with objective function

$$\min 5x_1 + 4x_2 + 3x_3 + 2x_4 + x_5$$

and that it finds the core $x_1 + x_2 + x_3 \geq 1$ at iteration 1 and the meta $o_2^1 + x_2 + x_4 \geq 1$ at iteration 2. Then the LP after iteration 1 is

(Primal)	(Dual)
$\min 5x_1 + 4x_2 + 3x_3 + 2x_2 + x_5$	$\max y^1_{sum}$
s.t.	
$x_1 + x_2 + x_3 - o_2^1 - o_3^1 = 1$	
$o_2^1 - o_3^1 - e_2^1 = 0$	
$x_1 \geq 0$	$y^1_{sum} \leq 5$
$x_2 \geq 0$	$y^1_{sum} \leq 4$
$x_3 \geq 0$	$y^1_{sum} \leq 3$
$o_2^1 \geq 0$	$-y^1_{sum} \leq 0$
$o_3^1 \geq 0$	$-y^1_{sum} \leq 0$

And after iteration 2 it will be

(Primal)	(Dual)
$\min 5x_1 + 4x_2 + 3x_3 + 2x_2 + x_5$	$\max y^1_{sum} + y^2_{sum}$
s.t.	
$x_1 + x_2 + x_3 - o_2^1 - o_3^1 = 1$	
$o_2^1 + x_2 + x_4 - o_2^2 - o_3^2 = 1$	
$o_2^1 - o_3^1 - e_2^1 = 0$	
$o_2^2 - o_3^2 - e_2^2 = 0$	
$x_1 \geq 0$	$y^1_{sum} \leq 5$
$x_2 \geq 0$	$y^1_{sum} + y^2_{sum} \leq 4$
$x_3 \geq 0$	$y^1_{sum} \leq 3$
$o_2^1 \geq 0$	$-y^1_{sum} + y^2_{sum} \leq 0$
$o_3^1 \geq 0$	$-y^1_{sum} \leq 0$
$o_2^2 \geq 0$	$-y^2_{sum} \leq 0$
$o_3^2 \geq 0$	$-y^2_{sum} \leq 0$

All variables in the primal are non-negative, but we only show above those that have a corresponding dual constraint. In both cases, we have in the same line a primal constraint and the bounds of its dual variable (unbounded here because the primal constraints are equalities), and the bounds of primal variable with the corresponding dual constraints. We can confirm that the dual solution $y^1_{sum} = 3, y^2_{sum} = 1$ is feasible and the reduced costs of the variables match what is computed by OLL. ⌟

One consequence of theorem 3 and theorem 1 is that we can change the reformulation step of OLL so that it uses an optimal dual solution to compute a reformulation, rather than that which is statically computed. Indeed, the reformulation step can be discarded altogether and the algorithm can use reduced costs directly.

Algorithm 2 OL^{3}P.

```
1  Procedure OL^3P (F = ⟨H, w⟩)
2      lb = 0
3      f^0_OLL = ∅
4      LP^0 = min(w^0)^T x
5      for iteration i = 1, ... do
6          (a, m^i) = solve((H ∪ f^{i-1}_OLL) |_{rc=0})
7          if a ≠ ∅ then
8              return a
9          f^i_OLL = f^{i-1}_OLL ∪ CNF(o^i_j ⟺ Σ_{x∈m^i} ≥ j) ∪ CNF(e^i_j ⟺ Σ_{x∈m^i} = j)
10         LP^i = LP^{i-1} ∪ LP_OLL(m^i)
11         lb = opt(LP^i)
```

We show the pseudocode for this algorithm, which we call OL^{3}P, in Algorithm 2. This follows the outline of OLL. It starts with lower bound zero and initializes the auxiliary formula f^0_{OLL} and corresponding LP LP^0 to the empty formula. Then, in each iteration of the loop in lines 5-11, it tests satisfiability of the formula $(H \cup f^{i-1}_{OLL})\,|_{rc=0}$ in line 6. This formula sets to false all variables which have non-zero reduced cost. In the first iteration, models of this formula correspond to feasible assignments of $\langle H, w\rangle$ that have cost 0 overall. In subsequent iterations, they correspond to assignments whose cost matches the current lower bound. If that formula is satisfiable, it reports optimality and exits in line 8. Otherwise, the SAT solver gives a meta m^i. In line 9, it adds to the subformula f^{i+1}_{OLL} the standard OLL constraints which introduce variables o^i_j to count how many variables of m^i are true, so that o^i_j is true if at least j variables of m^i are true. It additionally introduces the variables required by OL^{3}P, where e^i_j is true if exactly j variables of m^i are true. It also adds the corresponding linear constraints (3)-(5) to the LP in line 10. In constrast to OLL, it does not update the lower bound using the minimum cost of variables in m^i, nor does it update variable costs. Instead, it resolves the LP at line 11 to get an updated bound, as well as updated reduced costs for the next iteration.

▶ **Theorem 5.** OL^{3}P *is sound and complete*

Proof. For soundness, note that lower bounds are derived from the solution of an LP, which is itself built using constraints given by a SAT solver. Hence, all lower bounds are correct.

For completeness, note that since the LP is in augmented form, reduced costs define an equivalent objective. Hence, if the SAT solver determines that the instance $(H \cup f^{i-1}_{OLL})\,|_{rc=0}$ is satisfiable, i.e., it is satisfiable using only literals with 0 reduced cost, the cost of that assignment is exactly the optimum of the LP. Finally, since the lower bound increases at each iteration and the optimum is finite, the algorithm eventually exits with an optimum solution. ◀

While the transition from precomputed reformulation to one driven by LP dual solutions is straightforward, we need to highlight some subtlety in the notation. The instance given to the SAT solver is $(H \cup f^{i-1}_{OLL})\,|_{rc=0}$, which does not require that literals are positive. If a variable is fixed to 1 and has negative reduced cost, it means that it must be true in solutions. While this is not generally problematic with our formulation of MaxSAT, it is different when solving MaxSAT instances in the WCNF format. Specifically, if this happens

for a blocking variable, we require that the clause be falsified, which means that we must encode full reification, not just $\neg c \implies b$, as is typically done to improve performance. Similarly, if this happens for a sum variable o^i_j, it is required that the sum is *at least* j.

▶ **Example 6.** Suppose OL^{3}P is working on an instance with objective

$$\min 2x_1 + 3x_2 + 4x_3$$

and it has discovered the constraint $x_1 + x_2 + x_3 \geq 2$. The usual OLL reformulation would use this constraint to increase the lower bound to 4. However, we can see that there is no way to make two of these variables true and pay any less than 5. Indeed, the LP solver will find a dual solution in the reduced cost of x_1 is -2. This means that x_1 must be true. If it is false, the true variables must be x_2 and x_3, which have joint cost 7.

Now suppose that x_1, x_2, x_3 are blocking variables for soft clauses. If we set x_1 to false but the clause is not fully reified, the SAT solver can discover a solution that satisfies the clause of x_1, nevertheless sets x_1 to true, and its cost is higher than the bound computed by the LP. It will then terminate with LB $<$ UB, which is erroneous behavior. ⌟

The encoding used by many OLL-based solvers, totalizers [15, 23], is not fully reified. Instead, we have to use a different encoding, which can be made to encode full reification, in this case an encoding based on sorting networks [12]. A feature of the totalizer encoding is that it is incremental. For each meta m^i, only the variable o^i_2 needs to be introduced immediately, because any meta that contains o^i_3 can be transformed into one that contains o^i_2, as long as both have non-zero reduced cost. Therefore, practical implementations of OLL only encode o^i_{j+1} when the cost of o^i_j becomes zero. Sorting networks do not have this feature. This is a drawback of our approach, because it means that the SAT solver needs to solve a more complicated formula earlier than it would if it was possible to use an incremental encoding.

3.1 Seeding the LP

MaxHS [9], an implicit hitting set solver, which encodes the optimization part of MaxSAT as an integer linear program (ILP), makes extensive use of *seeding*. Given a MaxSAT formula $F = \langle H, w \rangle$, before it enters the main loop, MaxHS looks for clauses $c \in H$ which contain only literals of variables that appear in the objective. Then, it adds c to the ILP representation of the problem it solves. In addition, when there is a small number of additional variables that must be added to the ILP in order to give it some clauses, MaxHS adds those variables and corresponding clauses. While this may make the ILP larger and hence more difficult to solve, it can also provide useful information which improves both the bounds and the efficiency of the ILP solver.

We can do the same with OL^{3}P. However, where MaxHS can simply add clauses to the ILP problem it maintains, we are restricted to having LPs in augmented form. Hence, when we find a clause c that is suitable for seeding, we add $LP_{OLL}(c)$ to LP^0. The problem with this is that, while in MaxHS adding a clause to the ILP is just a single constraint, in OL^{3}P we need to add 4 times as many non-zeros and introduce $2|c| - 1$ new variables. This encoding has to be mirrored on the SAT side. However, only a relatively small number of constraints get a non-zero dual multiplier. Even if the LP and SAT solvers can determine that all the extra variables that appear in constraints with 0 dual multiplier are redundant, it still increases preprocessing time.

We can do better than that by reconsidering the proof of theorem 1. We can see that it is not necessary for the LP to have all constraints in the form $Ax = b$. Indeed, suppose that an LP has both equality constraints $Ax = b$ and inequality constraints $A'x \geq b'$ and suppose that the dual multiplier of every constraint in A' is 0. Then we can rewrite equation (2) as follows:

$$b^T y + (rc^y)^T x = b^T y + (c^T - A^T y - A'^T y)^T x \tag{6}$$

But since y assigns 0 to the multiplier of every constraint in A', we have $A'^T y = 0$ and the proof proceeds as before.

This allows us to make a simple but important optimization in seeding: when we initially seed the LP, we can leave all clauses as inequality constraints. This reduces the total number of variables in the LP and removes the need to add the corresponding sum constraints on the SAT side. When we actually solve an LP and find an inequality with non-zero dual multiplier, we can convert it to an equality, add all the necessary slack variables and corresponding sum constraint on the SAT side, then reuse the same dual solution to continue execution. But the number of constraints with non-zero dual multiplier is typically much smaller than the entire set of constraints. For instances where we can seed many or even all the initial clauses of the problem, this technique can represent significant savings.

3.2 Combining with implicit hitting set

Algorithm 2 is presented as extracting reduced cost-based metas at each iteration. As our argument in section 3.1 showed, however, the correctness of the algorithm does not depend in any way on the specific type of constraints. To this end, recall that when viewed as an ILP, LP_{OLL} encodes a hitting set problem over a set of cores derived from the metas discovered so far [17]. Hence, if we solve this minimum hitting set problem, we get a new source of cores. These cores are easier to discover than the metas discovered by the main OL^{3}P loop.

Unfortunately, in early experiments, CPLEX proved to be very poor at solving ILP_{OLL}, the integer version of LP_{OLL}. Even in cases where the optimum of the LP matches the optimum of the ILP, CPLEX struggles to find the optimum solution. Exploring this direction further remains future work.

3.3 Reduced cost fixing

As mentioned earlier, reduced cost fixing is a technique that is widely used in ILP solving, as well as in constraint programming [13] and in MaxHS [4]. In the literature of core-guided solvers, the closest analogue is hardening [15]. In particular, if we use exactly the potentially suboptimal dual solutions described previously [17] that simulate OLL, reduced cost fixing is exactly equivalent to hardening. Here, we use an optimal solution, which means we can potentially fix more variables. Note, however, that finding optimal reduced costs is itself a non-trivial problem [8].

There are two considerations for using reduced cost fixing with OL^{3}P. First, note that a practical implementation of OL^{3}P must implement techniques like stratification and weight-aware core extraction (covered in later sections). These split the execution of the solver into distinct phases of core (meta) extraction, each of which finishes with a satisfiable instance, generating a solution, hence an upper bound. This enables reduced cost fixing in the first place. However, it is now no longer necessary that the execution of the solver finishes with a SAT instance.

▶ **Example 7.** Suppose that a variable x always appears positively in every optimal solution of an instance F. Suppose further that we have already found an optimal solution of F and that the reduced cost of x is equal to its actual marginal cost. Then, x will be set to false by reduced cost fixing. There are three possible outcomes from this: either the LP reports a lower bound that exceeds the upper bound, the LP is infeasible, or the SAT solver reports an empty conflict. While seemingly erroneous, these results are correct. We are looking for a solution that improves on the optimum and we have pruned a literal that appears in all optimum solutions. The system is unsatisfiable, so any inference is correct. ⌟

A second consideration is that, knowing the specific relationship among the variables $o^i_2, \ldots, o^i_{k^i}$, we can compute better reduced costs. Consider o^i_3 for some i. When that is true, it implies also o^i_2, therefore we incur the cost of the reduced cost of both o^i_2 and o^i_3. In other words, the true marginal cost of o^i_3 is $mc(o^i_3) \geq rc^y(o^i_2) + rc^y(o^i_3)$. This generalizes to all indices of a sum. In preliminary experiments, we found that this technique can be very effective in fixing a large number of sum variables, but ends up hindering the performance of the SAT solver. We therefore do not use this technique in section 4.

3.4 Stratification

Stratification [15, 2] is a technique for improving solving of instances with diverse sets of costs. It splits variables of the objective into buckets, such that the higher the bucket index, the smaller the cost of the variable. It then solves the instance ignoring all but the first bucket of variables, i.e., treating all the other variables as if they have reduced cost 0. When that finishes, it adds variables from the next bucket and so on.

Stratification is that much more important for OL^{3}P. The dual solution is not guaranteed to yield integer reduced costs and in fact there typically exist many variables with very small reduced cost. Without stratification, OL^{3}P will be left to find metas with very small cost and make slow progress towards the optimum.

In addition, we implement rank-based stratification. We define the *rank* of an original variable of the instance to be $rank(x) = 0$. For all sum or equality variables corresponding to meta m^i, let r be the maximum rank among all $x \in m_i$. Then we set $rank(o^i_j) = rank(e^i_j) = r + 1$ for $j \in [2, k^i]$. For each cost-based bucket, we first extract all metas involving variables of rank 0, then 1, and so on. Note that metas of rank 0 are cores of the original formula.

The reasoning for this heuristic is that the higher the rank of the variables in a core, the harder it is for the SAT solver to discover, hence we want to extract the easier cores first.

4 Experimental evaluation

In this section, we describe the implementation details of a practical implementation of OL^{3}P and evaluate its performance.

4.1 Experimental Setup

We implemented OL^{3}P as an alternative algorithm on top of MaxHS [3]. We evaluate two variants. The first, OL^{3}P, implements the base algorithm OL^{3}P shown in Algorithm 2, with rank-based stratification, reduced cost fixing, weight aware cost extraction, and seeding as described in section 3.1. The second, OL^{3}P-S is the same but performs no seeding. This variant is the closest to base OLL, as the only major difference is that it uses LP for

[3] https://github.com/gkatsi/OLLLP

reformulation. Both variants use IBM CPLEX version 22.1.1.0 to solve the LP and CaDiCaL version "sc-2021" as a SAT solver. The ILP solver of CPLEX is not used. Additionally, both variants use the MaxPRE preprocessor [18].

We compare against CashWMaxSAT version "DisjCom-S6" [23], the winner of the 2024 MaxSAT evaluation in the weighted track. By default, CashWMaxSAT uses a portfolio strategy whereby, before launching OLL, it gives an instance to the SCIP ILP solver [7] and lets it run for some time. Only if SCIP does not solve the instance, does it execute the OLL algorithm. On the other hand, it does not use MaxPRE by default. We tested four configurations of CashWMaxSAT, varying the use of this feature and the use of MaxPRE:

1. As run in the 2024 evaluation, where the SCIP timeout is 600 seconds
2. As above, but with the SCIP timeout set to 300 seconds to match the lower timeout we used (see below)
3. With SCIP disabled, but using MaxPRE
4. Without either SCIP or MaxPRE

The latter two are more directly comparable to OL^{3}P-S, as they are not portfolios. They all use CaDiCaL version 2.0.0 as a SAT solver.

Finally, we compared against MaxHS, in its configuration from the 2022 MaxSAT evaluation, which is the last time it participated, using IBM CPLEX version 22.1.1.0 to solve the ILP and CaDiCaL version "sc-2021" as a SAT solver. MaxHS is also a portfolio solver. When the number of soft clauses is relatively small, it launches LSU instead. When the problem is small or the number of variables that do not appear in soft clauses is small, it delegates to CPLEX.

We compare the solvers on the instances from the weighted track of the 2024 MaxSAT evaluation. We ran all solvers for 1800 seconds of CPU time on a cluster of 14-core Intel Xeon E5-2695 v3 CPUs running at 2.30GHz.

4.2 Results

Surprisingly, the four different configurations of CashWMaxSAT varied little in their performance, solving 405, 409, 410, and 406 instances, respectively. It is possible that the shorter timeout compared to the evaluation made the portfolio ineffective. We retained configurations 2 and 3, as those exhibiting the best performance.

We show in Figure 1 a plot of the relative performance of the four solvers. Comparing only the core-guided solvers, we see that even though the solvers end at similar numbers of solved instances, both variants of OL^{3}P-S are better than CashWMaxSAT. We give a more detailed breakdown of the number of instances solved by family in Table 1. We see that most of the gains of OL^{3}P come from the JUDGEMENT-AGGREGATION family. There, the seeded LP computes a near optimum bound, then it takes just a few iterations to find the optimum solution and prove optimality. In OL^{3}P-S and CashWMaxSAT, this information is not available and they make slow progress, ending with a significant optimality gap. In the rest of the families, OL^{3}P sometimes exhibits a small advantage, but not consistently.

We also give in Table 2 the average time needed to solve instances (ignoring instances that timed out), as well as the PAR2 score of each solver (the sum of the runtimes over all instances, with instances that timed out or ran out of memory counting as twice the timeout, hence 3600 here). We see that both OL^{3}P-S and OL^{3}P have significantly reduced average solving time and lower PAR2 compared to CashWMaxSAT. This means that, not only does OL^{3}P solve more instances, it does so more quickly on average. The CashWMaxSAT +SCIP configuration does particularly poorly in these metrics because of the sequential portfolio it uses: for every instance that SCIP is unable to solve, it pays a penalty of 300 seconds. These

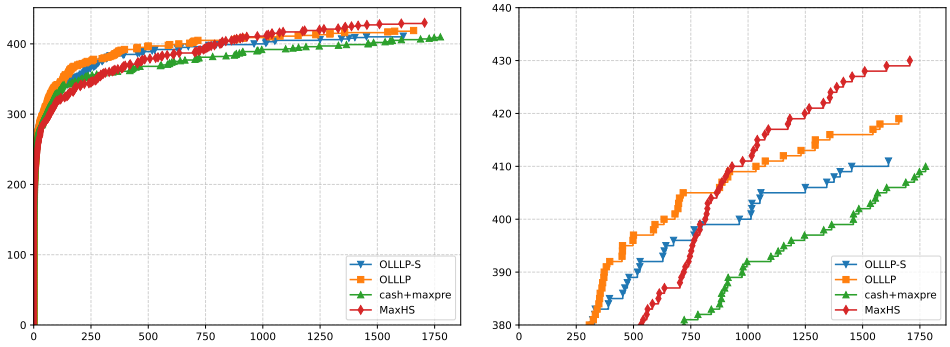


Figure 1 Results on exact weighted track of 2024 MaxSAT evaluation. On the right, zoomed in on the y axis so that the differences are highlighted.

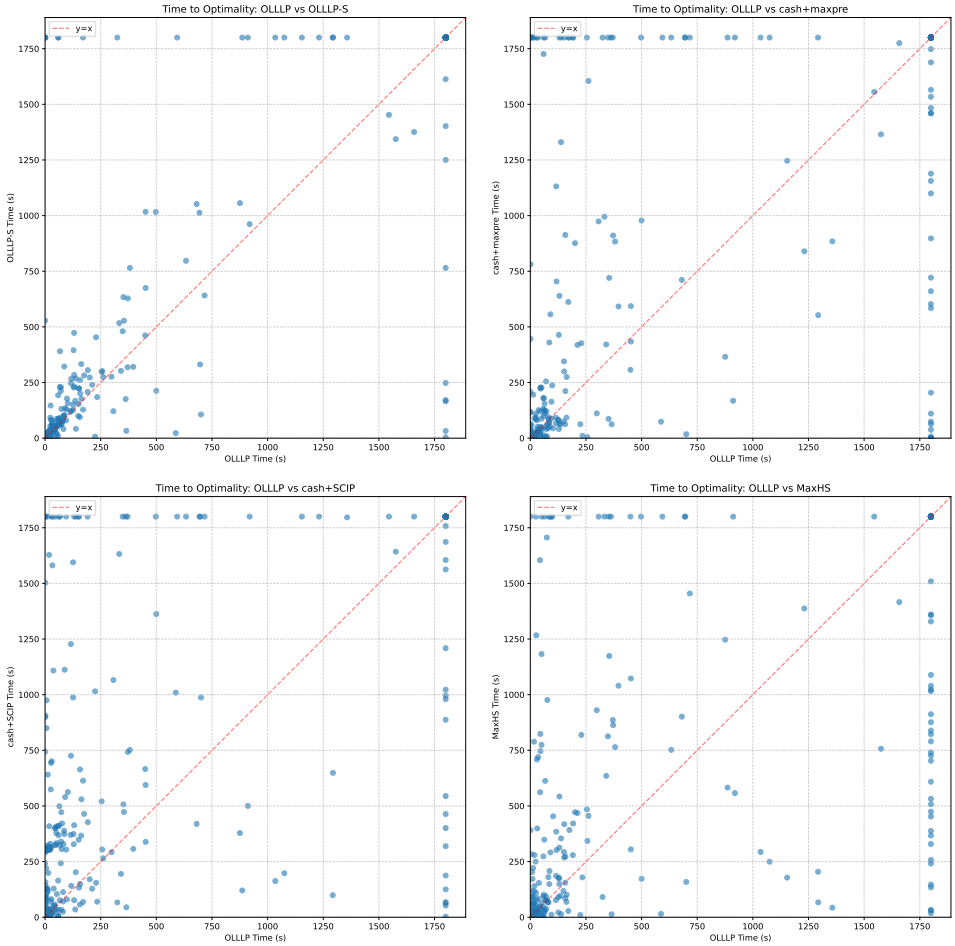


Figure 2 Scatter plots comparing OL^{3}P against the other three solvers. Dots over the $y = x$ line indicate that OL^{3}P was faster.

Table 1 Number of instances solved by each solver per family.

Family	#	Cash	Cash +SCIP	MaxHS	OL^{3}P-S	OL^{3}P
abstraction-refinement	11	11	10	11	11	11
af-synthesis	15	8	4	12	3	4
auctions	15	14	15	15	14	14
BTBNSL	15	3	2	7	3	3
causal-discovery	15	14	14	12	14	14
correlation-clustering	15	9	9	9	9	9
CSG	10	10	10	10	10	10
css-refactoring	11	11	11	8	11	11
dalculus	15	15	13	15	15	15
decision-tree	15	0	0	0	0	0
drmx-cryptogen	15	15	11	15	15	13
frb	15	14	11	14	15	15
haplotyping-pedigrees	15	15	15	15	15	15
IMLIB	16	16	16	16	16	16
judgment-aggregation	15	1	12	14	0	10
lisbon-wedding	15	3	3	1	3	4
max-realizability	15	14	14	13	13	12
MaxSATQueriesinInterpretableClassifiers	15	14	14	13	13	13
metro	15	15	15	15	15	15
MinimumWeightDominatingSetProblem	10	0	0	2	0	0
mpe	15	13	13	13	14	12
mpmcs	15	15	15	15	15	15
ParametricRBACMaintenance_mse20	15	0	0	0	3	3
planning	33	33	33	33	33	33
preference_planning	12	12	12	12	12	12
protein_ins	12	12	12	12	12	12
pseudoBoolean	15	14	14	14	14	14
qcp	15	15	15	15	15	15
quantum-circuit	15	15	15	14	15	15
railway-transport	5	2	2	1	2	1
ramsey	4	2	2	1	2	2
relational-inference	8	4	5	6	5	5
rna-alignment	15	10	10	15	15	15
setcover	6	0	3	3	0	0
spot5	15	13	10	11	11	11
switchingactivitymaximization	9	0	0	1	1	1
synplicate	40	15	15	16	15	15
tcp	15	13	13	13	13	14
timetabling	13	10	10	7	11	9
upgradeability	15	15	15	15	15	15
warehouses	8	3	8	8	5	8

observations are confirmed in the scatter plots in Figure 2. However, these plots additionally show that no solver dominates another, as there exist several instances that are solved by only one solver in each pair.

In comparison to MaxHS, all core-guided solvers perform worse overall. However, in some families, like JUDGEMENT-AGGREGATION and WAREHOUSES, OL^{3}P reduces the performance gap. Additionally, the average solving time of both OL^{3}P-S and OL^{3}P is signficantly lower in solved instances where it does prove optimality, although this is not sufficient to cover the gap in PAR2 score caused by the difference in number of instances solved.

Table 2 PAR2 scores.

Solver	PAR2 score	avg time
MaxHS	576598	160.462 s
CashWMaxSAT	640293	148.031 s
CashWMaxSAT +SCIP	670618	213.735 s
OL^{3}P-S	614729	94.2319 s
OL^{3}P	586971	94.9198 s

4.3 LP overhead

A natural question for both OL^{3}P-S and OL^{3}P is on the overhead of LP solving, since we replace the very cheap reformulation procedure of OLL by solving an LP. For OL^{3}P-S, this overhead is negligible. The solver spent just 16s solving the LP over the entire set of solved instances. The maximum amount of time spent in the LP solver in any one instances was just 1.15 s. In only 3 instances was the LP solver time more than 1 second, and they all were instances where the total time to solve them was more than 300 seconds. For OL^{3}P, the overhead is much greater. Cumulatively, OL^{3}P spent 8163.91 seconds in the LP solver (25 seconds per solved instance on average). The maximum amount of time in the LP solver in any one instance was 1343 seconds. The JUDGEMENT-AGGREGATION instances consume most of that time, with 4 of them spending more than 1000 seconds in the LP solver, each. Part of the reason for this overhead is that the current implementation is not particularly careful to be efficient with its use of the LP solver. These results suggest that more attention should be given to this, especially if other sources of constraints are used to populate the LP. Techniques like column and row generation can also speed up LP solving.

5 Conclusion

We have presented OL^{3}P-S, a core-guided algorithm for MaxSAT that uses a linear program to compute reformulations. This has advantages with respect to the behavior of the algorithm itself, as it computes stronger lower bounds from the same information. Moreover, since it can use constraints generated in any way, it opens many algorithmic possibilities. We explored here two: one that worked fairly well, seeding the LP with constraints of the MaxSAT formula; and one that did not, which was to try to solve the hitting set problem described by the LP as an integer program. Regardless, it is likely that more possible ways of enriching and using the linear program will emerge.

References

1 Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in clasp. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)(2012)*, pages 212–221. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.ICLP.2012.211.

2 Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving sat-based weighted maxsat solvers. In Michela Milano, editor, *Principles and Practice of Constraint Programming – 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2012. doi:10.1007/978-3-642-33558-7_9.

3 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Solving (weighted) partial maxsat through satisfiability testing. In *International conference on theory and applications of satisfiability testing*, pages 427–440. Springer, 2009. doi:10.1007/978-3-642-02777-2_39.

4 Fahiem Bacchus, Antti Hyttinen, Matti Järvisalo, and Paul Saikko. Reduced cost fixing in maxsat. 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 641–651. Springer, 2017. doi:10.1007/978-3-319-66158-2_41.

5 Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set maxsat solving. 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 277–294. Springer, 2020. doi:10.1007/978-3-030-51825-7_20.

6 Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided MaxSAT solving. In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22, 2023. doi:10.1007/978-3-031-38499-8_1.

7 Suresh Bolusani, Mathieu Besançon, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Christoph Graczyk, Katrin Halbig, Ivo Hedtke, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Julian Manns, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Dieter Weninger, and Lixing Xu. The SCIP Optimization Suite 9.0. Technical report, Optimization Online, February 2024. URL: https://optimization-online.org/2024/02/the-scip-optimization-suite-9-0/.

8 Guillaume Claus, Hadrien Cambazard, and Vincent Jost. Arc-consistency and linear programming duality: an analysis of reduced cost based filtering, 2022. doi:10.48550/arxiv.2207.10325.

9 Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming – CP 2011 – 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2011. doi:10.1007/978-3-642-23786-7_19.

10 Jessica Davies and Fahiem Bacchus. Exploiting the power of mip solvers in maxsat. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing – SAT 2013 – 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 166–181. Springer, 2013. doi:10.1007/978-3-642-39071-5_13.

11 Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up MAXSAT solving. In Christian Schulte, editor, *Principles and Practice of Constraint Programming – 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2013. doi:10.1007/978-3-642-40627-0_21.

12 Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *J. Satisf. Boolean Model. Comput.*, 2(1-4):1–26, 2006. doi:10.3233/sat190014.

13 Filippo Focacci, Andrea Lodi, and Michela Milano. Cost-based domain filtering. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming – CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings*, volume 1713 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 1999. doi:10.1007/978-3-540-48085-3_14.

14 Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing – SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006. doi:10.1007/11814948_25.

15 Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient maxsat solver. *J. Satisf. Boolean Model. Comput.*, 11(1):53–64, 2019. doi:10.3233/SAT190116.

16 Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Unifying SAT-based approaches to maximum satisfiability solving. *Journal of Artificial Intelligence Research*, 2024. doi:10.1613/jair.1.15986.

17 George Katsirelos. An Analysis of Core-Guided Maximum Satisfiability Solvers Using Linear Programming. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SAT.2023.12.

18 Tuukka Korhonen, Jeremias Berg, Paul Saikko, and Matti Järvisalo. Clause redundancy and preprocessing in maximum satisfiability. In Serge Gaspers and Toby Walsh, editors, *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing, (SAT '17)*, volume 10491 of *Lecture Notes in Computer Science*, pages 449–456. Springer, 2017. doi:10.1007/978-3-031-10769-6_6.

19 António Morgado, Carmine Dodaro, and João Marques-Silva. Core-guided maxsat with soft cardinality constraints. 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 564–573. Springer, 2014. doi:10.1007/978-3-319-10428-7_41.

20 António Morgado, Alexey Ignatiev, and João Marques-Silva. MSCG: robust core-guided maxsat solving. *J. Satisf. Boolean Model. Comput.*, 9(1):129–134, 2014. doi:10.3233/sat190105.

21 Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 2717–2723, 2014. doi:10.1609/aaai.v28i1.9124.

22 Nina Narodytska and Nikolaj S. Bjørner. Analysis of core-guided maxsat using cores and correction sets. 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 26:1–26:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.26.

23 Shiwei Pan, Yiyuan Wang, Shaowei Cai, Jiangman Li, Wenbo Zhu, and Minghao Yin. CASHWMaxSAT-DisjCad: Solver description. Technical report, Department of Computer Science, University of Helsinki, Helsinki, 2024. URL: http://hdl.handle.net/10138/584878.

Towards Practical First-Order Model Counting

Ananth K. Kidambi
Indian Institute of Technology Bombay, Mumbai, India

Guramrit Singh
Indian Institute of Technology Bombay, Mumbai, India

Paulius Dilkas
University of Toronto, Canada
Vector Institute, Toronto, Canada

Kuldeep S. Meel
Georgia Institute of Technology, Atlanta, GA, USA
University of Toronto, Canada

Abstract

First-order model counting (FOMC) is the problem of counting the number of models of a sentence in first-order logic. Since lifted inference techniques rely on reductions to variants of FOMC, the design of scalable methods for FOMC has attracted attention from both theoreticians and practitioners over the past decade. Recently, a new approach based on first-order knowledge compilation was proposed. This approach, called Crane, instead of simply providing the final count, generates definitions of (possibly recursive) functions that can be evaluated with different arguments to compute the model count for any domain size. However, this approach is not fully automated, as it requires manual evaluation of the constructed functions. The primary contribution of this work is a fully automated compilation algorithm, called Crane2, which transforms the function definitions into C++ code equipped with arbitrary-precision arithmetic. These additions allow the new FOMC algorithm to scale to domain sizes over 500,000 times larger than the current state of the art, as demonstrated through experimental results.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Logic and verification; Mathematics of computing → Combinatorics

Keywords and phrases First-order model counting, knowledge compilation, lifted inference

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.18

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2502.12278` [11]

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.15583934`

Funding This research was funded in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference number RGPIN-2024-05956, and the Digital Research Alliance of Canada (`alliancecan.ca`).

Acknowledgements The first two authors contributed equally. Part of the research was conducted while all authors were at the National University of Singapore. We express our gratitude to the anonymous reviewers for their valuable comments and suggestions.

1 Introduction

First-order model counting (FOMC) is the task of determining the number of models for a sentence in first-order logic over a specified domain. The weighted variant, WFOMC, computes the total weight of these models, linking logical reasoning with probabilistic frameworks [31]. It builds upon earlier efforts in weighted model counting for propositional logic [4] and broader attempts to bridge logic and probability [14, 16, 20]. WFOMC is central to *lifted inference*, which enhances the efficiency of probabilistic calculations by exploiting



© Ananth K. Kidambi, Guramrit Singh, Paulius Dilkas, and Kuldeep S. Meel;
licensed under Creative Commons License CC-BY 4.0



symmetries [10]. Lifted inference continues to advance, with applications extending to constraint satisfaction problems [25] and probabilistic answer set programming [1]. Moreover, WFOMC has proven effective at reasoning over probabilistic databases [8] and probabilistic logic programs [18]. FOMC algorithms have also facilitated breakthroughs in discovering integer sequences [22] and developing recurrence relations for these sequences [6]. Recently, these algorithms have been extended to perform sampling tasks [32].

The complexity of FOMC is generally measured by *data complexity*, with a sentence classified as *liftable* if it admits a polynomial-time solution relative to the domain size [9]. While all sentences with up to two variables are known to be liftable [28, 30], Beame et al. [3] demonstrated that liftability does not extend to all sentences, identifying an unliftable sentence with three variables. Recent work has extended the liftable fragment with additional axioms [23, 27] and counting quantifiers [12], expanding our understanding of liftability.

FOMC algorithms are diverse, with approaches ranging from *first-order knowledge compilation* (FOKC) to cell enumeration [26], local search [15], and Monte Carlo sampling [7]. FOKC-based algorithms are particularly prominent, transforming sentences into structured representations such as circuits or graphs. Even when multiple algorithms can solve the same instance, FOKC algorithms are known to find polynomial-time solutions where the polynomial has a lower degree than other approaches [6]. The recently developed ability of a FOKC algorithm to formulate solutions in terms of recursive functions [6] is also noteworthy since the only other proposed alternative is to guess recursive relations [2]. Notable examples of FOKC algorithms include ForcLift [31] and its extension Crane [6].

The Crane algorithm marked a significant step forward, expanding the range of sentences handled by FOMC algorithms. However, it had notable limitations: it required manual evaluation of function definitions to compute model counts and introduced recursive functions without proper base cases, making it more difficult to use. To address these shortcomings, we present Crane2, a fully automated FOMC algorithm that overcomes the constraints of its predecessor. Crane2 can handle domain sizes over 500,000 times larger than those of previous algorithms and simplifies the user experience by automatically managing base cases and compiling function definitions into efficient C++ programs.

The paper is structured as follows. Section 2 covers the necessary preliminaries, including notation, terminology, and background information about (W)FOMC and FOKC. Section 3 describes all aspects of Crane2:

- its overall structure and interactions with Crane,
- how we identify a sufficient set of base cases for a recursive function,
- how we construct the sentence that describes each base case, and
- how we compile function definitions into a C++ program with caching mechanisms that ensure efficiency.

Then, Section 4 presents our experimental results, demonstrating Crane2's performance compared to other FOMC algorithms. Finally, Section 5 concludes the paper by discussing promising avenues for future work.

2 Preliminaries

We begin this section by describing some notation that we will use throughout the paper. Then, in Sections 2.1 and 2.2, we introduce the basic terminology of first-order logic and formally define (W)FOMC. Section 2.3 provides a brief overview of the different types of first-order logics commonly used in FOMC literature. Section 2.4 outlines the principles of FOKC, particularly in the context of Crane. Finally, in Section 2.5, we introduce the algebraic terminology used to describe the output of Crane, i.e., functions and equations that define them.

Notation

We use $\mathbb{N}_0$ to represent the set of non-negative integers. In both algebra and logic, we write $S[x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n]$ to denote the application of a *substitution* to an expression S, where all instances of x_i are replaced with y_i for all $i = 1, \ldots, n$. Additionally, for any variable n and $a, b \in \mathbb{N}_0$, let $[a \leq n \leq b] \coloneqq \begin{cases} 1 & \text{if } a \leq n \leq b \\ 0 & \text{otherwise} \end{cases}$.

2.1 First-Order Logic

This section reviews the basic concepts of first-order logic used in FOKC algorithms. We focus specifically on the format used internally by ForcLift and its descendants. See Section 2.4.1 for a brief overview of how Crane2 transforms an arbitrary sentence in first-order logic into this internal format.

A *term* can be either a variable or a constant. An *atom* can be either $P(t_1, \ldots, t_m)$ (i.e., $P(\mathbf{t})$) for some predicate P and terms $t_1, \ldots, t_m$, or $x = y$ for some terms x and y. The *arity* of a predicate is the number of arguments it takes, i.e., m in the case of the predicate P mentioned above. We write P/m to denote a predicate along with its arity. A *literal* can be either an atom (i.e., a *positive* literal) or its negation (i.e., a *negative* literal). An atom containing no variables, only constants, is called *ground*. A *clause* is of the form $\forall x_1 \in \Delta_1.\ \forall x_2 \in \Delta_2 \ldots\ \forall x_n \in \Delta_n.\ \phi(x_1, x_2, \ldots, x_n)$, where ϕ is a disjunction of literals that contain only the variables $x_1, \ldots, x_n$ (and any constants). We say that a clause is a *(positive) unit clause* if there is only one literal with a predicate, and it is a positive literal. Finally, a *sentence* is a conjunction of clauses. Throughout the paper, we will use set-theoretic notation, interpreting a sentence as a set of clauses and a clause as a set of literals.

▶ Remark. Conforming to previous work [31], the definition of a clause includes universal quantifiers for all its variables. While it is possible to rewrite the entire sentence with all quantifiers at the front, the format we describe has proven convenient for practical use.

2.2 First-Order Model Counting

In this section, we will formally define FOMC and its weighted variant. Although this work focuses on FOMC, computing the FOMC using Crane2 requires using WFOMC for sentences with existential quantifiers. For such sentences, preprocessing (described in Section 2.4.1) introduces predicates with non-unary weights that must be accounted for to compute the correct model count.

▶ **Definition 1** (Structure, model). *Let ϕ be a sentence. For each predicate P/n in ϕ, let $(\Delta_i^P)_{i=1}^n$ be a list of the corresponding domains. Let σ be a map from the domains of ϕ to their interpretations as finite sets, ensuring the sets are pairwise disjoint and contain the corresponding constants from ϕ. A* structure *of ϕ is a set M of ground literals. It is defined by adding to M either $P(\mathbf{t})$ or $\neg P(\mathbf{t})$ for every predicate P/n in ϕ and every n-tuple $\mathbf{t} \in \prod_{i=1}^n \sigma(\Delta_i^P)$. (Here, $\prod$ is the usual Cartesian product.) A structure is a* model *if it makes ϕ valid.*

▶ **Example 2** (Counting bijections). Let us consider the following sentence (previously examined by Dilkas and Belle [6]) that defines predicate P as a bijection between two domains Γ and Δ:

Table 1 A comparison of the three logics used in FOMC. The 2nd–5th columns refer to the number of sorts, support for constants, the maximum number of variables, and supported quantifiers, respectively. The last column lists supported atoms in addition to those of the form $P(\mathbf{t})$ for a predicate P/n and an n-tuple of terms $\mathbf{t}$. Here, k and m are non-negative integers, where m depends on the domain size, P is a predicate, and x and y are terms.

Logic	Sorts	Constants	Variables	Quantifiers	Add. atoms
FO	one or more	✓	unlimited	$\forall$, $\exists$	$x = y$
C^2	one	✗	two	$\forall$, $\exists$, $\exists^{=k}$, $\exists^{\leq k}$, $\exists^{\geq k}$	–
$\mathsf{UFO}^2 + \mathsf{CC}$				$\forall$	$\lvert P\rvert = m$

$$\begin{aligned}
&(\forall x \in \Gamma.\ \exists y \in \Delta.\ P(x,y)) \wedge \\
&(\forall y \in \Delta.\ \exists x \in \Gamma.\ P(x,y)) \wedge \\
&(\forall x \in \Gamma.\ \forall y,z \in \Delta.\ P(x,y) \wedge P(x,z) \Rightarrow y = z) \wedge \\
&(\forall x,z \in \Gamma.\ \forall y \in \Delta.\ P(x,y) \wedge P(z,y) \Rightarrow x = z).
\end{aligned} \tag{1}$$

Let σ be defined as $\sigma(\Gamma) \coloneqq \{1, 2\}$ and $\sigma(\Delta) \coloneqq \{a, b\}$. Sentence (1) has two models:

$$\{P(1,a), P(2,b), \neg P(1,b), \neg P(2,a)\} \qquad \text{and} \qquad \{P(1,b), P(2,a), \neg P(1,a), \neg P(2,b)\}.$$

▶ **Definition 3** (WFOMC instance). *A* WFOMC instance *comprises:*

- *a sentence ϕ,*
- *two (rational)* weights $w^+(P)$ *and* $w^-(P)$ *assigned to each predicate P in ϕ, and*
- *σ as described in Definition 1.*

Unless specified otherwise, we assume all weights are equal to 1.

▶ **Definition 4** (WFOMC [31]). *Given a WFOMC instance (ϕ, w^+, w^-, σ) as in Definition 3, the* (symmetric) weighted first-order model count *(WFOMC) of ϕ is*

$$\sum_{M \models \phi} \prod_{P(\mathbf{t}) \in M} w^+(P) \prod_{\neg P(\mathbf{t}) \in M} w^-(P), \tag{2}$$

where the sum is over all models of ϕ.

2.3 The Three Logics of FOMC

FOMC commonly utilises three types of first-order logic: FO, C^2, and $\mathsf{UFO}^2 + \mathsf{CC}$. Table 1 summarises the key differences among them. FO is the input format for ForcLift and its extensions Crane and Crane2. C^2 is often used in the literature on FastWFOMC and related methods [12, 13]. (Note that no algorithm accepts C^2 as input.) Finally, $\mathsf{UFO}^2 + \mathsf{CC}$ is the input format supported by the most recent implementation of FastWFOMC [24]. All three logics are function-free, and domains are always assumed to be finite. As usual, we presuppose the *unique name assumption*, which states that two constants are equal if and only if they are the same constant [19].

In FO, each term has a designated *sort*, and each predicate P/n corresponds to a sequence of n sorts. Each sort has its corresponding domain. These assignments to sorts are typically left implicit and follow from the quantifiers, e.g., $\forall x, y \in \Delta.\ P(x,y)$ implies that the variables x and y have the same sort. On the other hand, $\forall x \in \Delta.\ \forall y \in \Gamma.\ P(x,y)$

implies that x and y have different sorts, and it would be improper to write, for example, $\forall x \in \Delta.\ \forall y \in \Gamma.\ P(x, y) \vee x = y$. FO is also the only logic to support constants, sentences with more than two variables, and the equality predicate. While we do not explicitly refer to sorts in the rest of the paper, the many-sorted nature of FO is paramount to the algorithms presented therein.

▶ Remark. In the case of ForcLift and its extensions, support for a sentence as valid input does not imply that the algorithm can compile the sentence into a circuit or graph suitable for lifted model counting. However, ForcLift compilation always succeeds on any FO sentence without constants and with at most two variables [28, 30].

Compared to FO, C^2 and $\mathsf{UFO}^2 + \mathsf{CC}$ lack support for constants, the equality predicate, multiple domains, and sentences with more than two variables. The advantage that C^2 brings over FO is the inclusion of *counting quantifiers*. That is, alongside $\forall$ and $\exists$, C^2 supports $\exists^{=k}$, $\exists^{\leq k}$, and $\exists^{\geq k}$ for any positive integer k. For example, $\exists^{=1} x.\ \phi(x)$ means that there exists *exactly one* x such that $\phi(x)$, and $\exists^{\leq 2} x.\ \phi(x)$ means that there exist *at most two* such x. $\mathsf{UFO}^2 + \mathsf{CC}$, on the other hand, does not support any existential quantifiers but instead incorporates *(equality) cardinality constraints*. For example, $|P| = 3$ constrains all models to have *precisely three positive literals with the predicate* P. Except when stated otherwise, we will write all example sentences in FO.

2.4 Crane and First-Order Knowledge Compilation

As our work builds on Crane, in this section, we will briefly outline the steps Crane goes through to compile a sentence into a set of function definitions. We divide the inner workings of the algorithm into two stages: preprocessing and compilation.

2.4.1 Preprocessing

This stage transforms an arbitrary sentence into the format described in Section 2.1, primarily by eliminating existential quantifiers via a process called *Skolemization* [30]. For example, the first conjunct of sentence (1), i.e.,

$$\forall x \in \Gamma.\ \exists y \in \Delta.\ P(x, y) \tag{3}$$

is transformed into

$$\begin{aligned} &(\forall x \in \Gamma.\ Z(x)) \wedge \\ &(\forall x \in \Gamma.\ \forall y \in \Delta.\ Z(x) \vee \neg P(x, y)) \wedge \\ &(\forall x \in \Gamma.\ S(x) \vee Z(x)) \wedge \\ &(\forall x \in \Gamma.\ \forall y \in \Delta.\ S(x) \vee \neg P(x, y)), \end{aligned} \tag{4}$$

where $Z/1$ and $S/1$ are two new predicates with $w^-(S) = -1$. The idea behind Skolemization is that the output sentence has more models, but for every additional model with a positive weight, there is an equivalent model with a negative weight. Hence, the WFOMC of sentences (3) and (4) is the same. Moreover, note that in this example, preprocessing introduces some redundancy, i.e., the second and third clauses of sentence (4) could be removed. These redundancies are handled at the beginning of the compilation process described in Section 2.4.2.

To illustrate how Skolemization preserves the WFOMC, we let $\Gamma = \{ a \}$ and $\Delta = \{ 1, 2 \}$ and assume that $w^+(P) = w^-(P) = 1$. Here, for conciseness, we omit negative literals when writing models. Sentence (3) has three models: $\{ P(a, 1) \}$, $\{ P(a, 2) \}$, and $\{ P(a, 1), P(a, 2) \}$, all with weight 1, so the WFOMC is 3. Sentence (4) has the following models:

- $\{ Z(a) \}$,
- $\{ Z(a), S(a) \}$,
- $\{ Z(a), S(a), P(a, 1) \}$,
- $\{ Z(a), S(a), P(a, 2) \}$,
- $\{ Z(a), S(a), P(a, 1), P(a, 2) \}$.

The first model has a weight of -1 because $w^-(S) = -1$, while all the other models have a weight of 1. Hence, the WFOMC is $4 - 1 = 3$.

2.4.2 Compilation

After preprocessing, Crane compiles the sentence into the triple $(\mathcal{E}, \mathcal{F}, \mathcal{D})$, where $\mathcal{E}$ is the set of equations, and $\mathcal{F}$ and $\mathcal{D}$ are auxiliary maps. $\mathcal{F}$ maps function names to sentences in such a way that the evaluation of the former corresponds to the FOMC of the latter. $\mathcal{D}$ maps function names and argument indices to domains. The equations in $\mathcal{E}$ can define an arbitrary number of functions, one of which (which we will always denote as f) represents the solution to the FOMC problem. Computing the FOMC for specific domain sizes involves invoking f with those sizes as inputs. $\mathcal{D}$ records this correspondence between function arguments and domains.

▶ **Example 5.** Crane compiles sentence (1) for bijection counting into

$$\mathcal{E} = \left\{ \begin{array}{l} f(m,n) = \sum_{l=0}^{n} \binom{n}{l} (-1)^{n-l} g(l,m), \\ \\ g(l,m) = \sum_{k=0}^{m} [0 \le k \le 1] \binom{m}{k} g(l-1, m-k) \end{array} \right\};$$

$$\mathcal{D} = \{ (f,1) \mapsto \Gamma, (f,2) \mapsto \Delta, (g,1) \mapsto \Delta^\top, (g,2) \mapsto \Gamma \},$$

where $\Delta^\top$ is a newly introduced domain. (We omit the definition of $\mathcal{F}$ as the sentences can become quite verbose.) To compute the number of bijections between two sets of cardinality 3, one would evaluate $f(3,3)$; however, the definition of g is incomplete: g is a recursive function presented without any base cases. $\mathcal{D}$ encodes that in $f(m,n)$, m and n represent $|\Gamma|$ and $|\Delta|$, respectively. Similarly, in $g(l,m)$, l represents $|\Delta^\top|$, and m represents $|\Gamma|$.

Compilation is performed primarily by applying *(compilation) rules* to sentences. Crane has two modes depending on the selection process for compilation rules when multiple alternatives are available. The first option is to use greedy search: there is a list of rules, and the first applicable rule is the one that gets used, disregarding all the others. The second option is to use a combination of greedy and *breadth-first search* (BFS). In this approach, we classify each compilation rule as greedy or non-greedy. Greedy rules are applied as soon as possible at any stage of the compilation process, while BFS is executed over all applicable non-greedy rules, identifying the solution that necessitates the smallest number of non-greedy rules.

2.5 Algebra

In this paper, we use both logical and algebraic constructs. While the rest of Section 2 focused on the former, this section describes the latter. We write `expr` for an arbitrary algebraic expression. In the context of algebra, a *constant* is a non-negative integer. Likewise, a *variable* can either be a parameter of a function or a variable introduced through summation,

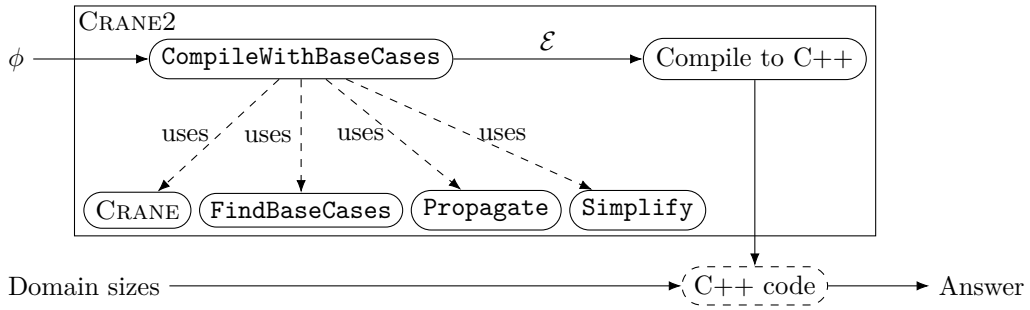


Figure 1 Using CRANE2 to compute the model count of a sentence ϕ. First, CRANE2 compiles ϕ into a set of equations, which form the basis for creating a C++ program. Executing the program with different command line arguments calculates the model count of ϕ for various domain sizes. To accomplish this, the `CompileWithBaseCases` procedure employs CRANE, algebraic simplification techniques (denoted as `Simplify`), and two other auxiliary procedures.

such as i in the expression $\sum_{i=1}^{n}$ `expr`. A *function call* is $f(x_1, \ldots, x_n)$ (or $f(\mathbf{x})$ for short), where f is an n-ary function, and each x_i is an algebraic expression consisting of variables and constants. A (function) *signature* is a function call that contains only variables. Given two function calls, $f(\mathbf{x})$ and $f(\mathbf{y})$, we say that $f(\mathbf{y})$ *matches* $f(\mathbf{x})$ if $x_i = y_i$ whenever $x_i, y_i \in \mathbb{N}_0$. An *equation* is $f(\mathbf{x}) =$ `expr`, where $f(\mathbf{x})$ is a function call.

▶ **Definition 6** (Base case). *Let $f(\mathbf{x})$ be a function call where each x_i is either a constant or a variable. Then the function call $f(\mathbf{y})$ is a* base case *of $f(\mathbf{x})$ if $f(\mathbf{y}) = f(\mathbf{x})\sigma$, where σ is a substitution that replaces one or more* variable *x_i's with a constant while leaving constants unchanged.*

▶ **Example 7.** In the equation $f(m, n) = f(m-1, n) + nf(m-1, n-1)$, the only constant is 1, and the variables are m and n. The equation contains three function calls: one on the left-hand side (LHS) and two on the right-hand side (RHS). The function call on the LHS is a signature. Function calls such as $f(4, n)$, $f(m, 0)$, and $f(8, 1)$ are all considered base cases of $f(m, n)$ (only some of which are useful).

3 Technical Contributions

Figure 1 provides an overview of CRANE2's workflow. We will briefly describe and motivate each procedure before going into more detail in the corresponding subsection.

`CompileWithBaseCases` (see Section 3.1), the core procedure of CRANE2, is responsible for completing the function definitions produced by CRANE with the necessary base cases. To do so, it may recursively call itself (and CRANE) on other sentences. We prove that the number of such recursive calls is upper bound by the number of domains.

Section 3.1 also describes the `Simplify` procedure for algebraic simplification. It is crucial for simplifying, e.g., a sum of n terms, only some of which are non-zero. More generally, the equations returned by CRANE often benefit from easily detectable algebraic simplifications such as $0 \cdot \text{anything} = 0$ and $\text{anything}^0 = 1$. Note that the latter includes $0^0 = 1$, as 1 is the multiplicative identity.

`FindBaseCases` (described in Section 3.2) inspects a set of equations to identify a sufficient set of base cases for a given set of equations. We prove that the returned set of base cases ensures that the evaluation of the resulting function definitions will never get stuck in an infinite loop.

Algorithm 1 `CompileWithBaseCases(`ϕ`)`.

Input: sentence ϕ
Output: set $\mathcal{E}$ of equations
1 $(\mathcal{E}, \mathcal{F}, \mathcal{D}) \leftarrow$ CRANE(ϕ);
2 $\mathcal{E} \leftarrow$ `Simplify(`$\mathcal{E}$`)`;
3 **foreach** *base case* $f(\mathbf{x}) \in$ `FindBaseCases(`$\mathcal{E}$`)` **do**
4 $\quad \psi \leftarrow \mathcal{F}(f)$;
5 $\quad$ **foreach** *index* i *such that* $x_i \in \mathbb{N}_0$ **do** $\psi \leftarrow$ `Propagate(`ψ, $\mathcal{D}(f, i)$, x_i`)`;
6 $\quad \mathcal{E} \leftarrow \mathcal{E} \cup$ `CompileWithBaseCases(`ψ`)`;

Section 3.3 introduces the `Propagate` procedure, which takes a sentence ϕ, a domain Δ, and $n \in \mathbb{N}_0$. It returns ϕ transformed under the assumption that $|\Delta| = n$, with n new constants added and all variables quantified over Δ eliminated. For example, when computing a base case such as $f(0, y)$, `Propagate` will significantly simplify ϕ with the assumption that the domain associated with the first parameter of f (i.e., $\mathcal{D}(f, 1)$) is empty. `CompileWithBaseCases(Propagate(`ϕ, $\mathcal{D}(f, 1)$, 0`))` will then return the equations for the base case $f(0, y)$.

Section 3.4 describes a new *smoothing* procedure that ensures `Propagate` preserves the correct model count. Smoothing is a well-known technique in knowledge compilation algorithms for propositional model counting [5]. Although FOMC algorithms have used smoothing before [31], our setting requires a novel approach.

With the help of other procedures outlined above, `CompileWithBaseCases` returns a set of equations that fully cover the base cases of all recursive functions. While these equations can be intriguing and valuable in their own right, users of FOMC algorithms typically expect a numerical answer. Thus, Section 3.5 describes how CRANE2 compiles these equations into a C++ program that one can execute with different command-line arguments to compute the model count for various combinations of domain sizes.

3.1 Completing the Definitions of Functions

Algorithm 1 presents our overall approach for compiling a sentence into equations that include the necessary base cases. First, we use CRANE to compile the sentence into three components: $\mathcal{E}$, $\mathcal{F}$, and $\mathcal{D}$ (as described in Section 2.4.2). After some algebraic simplifications (described below), the algorithm passes $\mathcal{E}$ to the `FindBaseCases` procedure (see Section 3.2). For each base case $f(\mathbf{x})$, we retrieve the sentence $\mathcal{F}(f)$ associated with the function name f and simplify it using the `Propagate` procedure (explained in detail in Section 3.3). We do this by iterating over all indices of $\mathbf{x}$, where x_i is a constant, and using `Propagate` to simplify ψ by assuming that domain $\mathcal{D}(f, i)$ has size x_i. Finally, on line 6, `CompileWithBaseCases` recurses on these simplified sentences and adds the resulting base case equations to $\mathcal{E}$.

Simplify

The main responsibility of the `Simplify` procedure is to handle the algebraic pattern $\sum_{m=0}^{n} [a \leq m \leq b] f(m)$. Here, n is a variable, a and b are constants, and f is an expression that may depend on m. `Simplify` transforms this pattern into $f(a) + f(a+1) + \cdots + f(\min\{\, n, b \,\})$.

▶ **Example 8.** We return to the bijection-counting problem from Example 2 and its initial solution described in Example 5. `Simplify` transforms

$$g(l,m) = \sum_{k=0}^{m} [0 \leq k \leq 1] \binom{m}{k} g(l-1, m-k)$$

into

$$g(l,m) = g(l-1,m) + mg(l-1,m-1).$$

Then `FindBaseCases` identifies two base cases: $g(0,m)$ and $g(l,0)$. In both cases, `CompileWithBaseCases` recurses on the sentence $\mathcal{F}(g)$ simplified by assuming that one of the domains is empty. In the first case, we recurse on the sentence $\forall x \in \Gamma.\ S(x) \vee \neg S(x)$, where S is a predicate introduced by preprocessing with weights $w^+(S) = 1$ and $w^-(S) = -1$. Hence, we obtain the base case $g(0,m) = 0^m$. In the case of $g(l,0)$, `Propagate(`ψ`, Γ, 0)` returns an empty sentence, resulting in $g(l,0) = 1$. While these base cases overlap when $l = m = 0$, they remain consistent since $0^0 = 1$.

Generally, let ϕ be a sentence with two domains Γ and Δ, and let $n, m \in \mathbb{N}_0$. Then the FOMC of `Propagate(`ϕ`, Δ, n)`, assuming $|\Gamma| = m$, is the same as the FOMC of `Propagate(`ϕ`, Γ, m)`, assuming $|\Delta| = n$.

Although `CompileWithBaseCases` starts with a call to Crane, the proposed algorithm is not merely a post-processing step for FOKC because Algorithm 1 is recursive and can issue additional calls to Crane on various derived sentences. We conclude this section by bounding the recursion depth of the `CompileWithBaseCases` procedure, thereby also proving that the algorithm terminates.

▶ **Theorem 9.** *Let ϕ be a sentence with n domains. The maximum recursion depth of* `CompileWithBaseCases(`ϕ`)` *is then n.*

The proof of this theorem relies on two observations regarding the algorithms presented in Sections 3.2 and 3.3.

▶ **Observation 10.** *Each base case returned by* `FindBaseCases` *contains at least one constant (in line with Definition 6).*

▶ **Observation 11.** *For any sentence ϕ, domain Δ, and $n \in \mathbb{N}_0$,* `Propagate(`ϕ`, Δ, n)` *returns a sentence with no variables quantified over Δ.*

Proof. We proceed by induction on n. If $n = 0$, then ϕ is essentially a propositional formula, and Crane compiles it into an equation of the form $f =$ `expr` with no "function calls". Suppose that – for all sentences with at most n domains – `CompileWithBaseCases` terminates with a recursion depth of at most n. Let ϕ be a sentence with $n+1$ domains. By Observation 10, each base case on line 3 of Algorithm 1 has at least one constant. Therefore, by Observation 11, after line 5, the sentence ψ has at most n domains. Thus, line 6 terminates with a recursion depth of at most n by the inductive hypothesis, completing the inductive proof. ◀

3.2 Identifying a Sufficient Set of Base Cases

Algorithm 2 summarises the implementation of `FindBaseCases`. It considers two types of arguments when a function f calls itself recursively: constants and arguments of the form $x_i - c_i$. When the argument is a constant c_i, line 5 adds a base case with c_i to the set of all base cases $\mathcal{B}$. In the second case, line 7 adds a base case to $\mathcal{B}$ for each constant from 0 to (but not including) c_i.

Algorithm 2 `FindBaseCases`($\mathcal{E}$).

Input: set $\mathcal{E}$ of equations
Output: set $\mathcal{B}$ of base cases

$\mathcal{B} \leftarrow \emptyset$;
foreach *function call* $f(\mathbf{y})$ *on the RHS of an equation in* $\mathcal{E}$ **do**
  $\mathbf{x} \leftarrow$ the parameters of f in its definition;
  foreach $y_i \in \mathbf{y}$ **do**
    if $y_i \in \mathbb{N}_0$ **then** $\mathcal{B} \leftarrow \mathcal{B} \cup \{ f(\mathbf{x})[x_i \mapsto y_i] \}$;
    else if $y_i = x_i - c_i$ *for some* $c_i \in \mathbb{N}_0$ **then**
      for $j \leftarrow 0$ **to** $c_i - 1$ **do** $\mathcal{B} \leftarrow \mathcal{B} \cup \{ f(\mathbf{x})[x_i \mapsto j] \}$;

▶ **Example 12.** Consider the recursive function g from Example 5. `FindBaseCases` iterates over two function calls: $g(l-1, m)$ and $g(l-1, m-1)$. The former produces the base case $g(0, m)$, while the latter produces both $g(0, m)$ and $g(l, 0)$.

In the rest of this section, we will show that the base cases identified by `FindBaseCases` are sufficient for the algorithm to terminate.[1]

▶ **Theorem 13.** *Let $\mathcal{E}$ denote the set of equations returned by* `CompileWithBaseCases`*, and let f be one of the functions defined in $\mathcal{E}$. The evaluation of $f(\mathbf{x})$ terminates for any tuple $\mathbf{x}$ of non-negative integers (of appropriate length).*

We prove Theorem 13 using double induction. First, we apply induction to the number of functions in $\mathcal{E}$. Then, we use induction on the arity of the "last" function in $\mathcal{E}$ according to a topological ordering. Before proving Theorem 13, we make a few observations that are immediate consequences of this and previous work [6, 31].

▶ **Observation 14.** *For each function f, there is precisely one equation $e \in \mathcal{E}$ with $f(\mathbf{x})$ on the LHS where all x_i's are variables (i.e., e is not a base case). We refer to e as the* definition *of f.*

▶ **Observation 15.** *There is a* topological ordering $(f_i)_i$ *of all functions in $\mathcal{E}$ such that equations in $\mathcal{E}$ with f_i on the LHS do not contain function calls to f_j with $j > i$.*

Observation 15 prevents mutual recursion and other cyclic scenarios. It arises from the timeline of FOKC and the fact that, while constructing a graph, the algorithm cannot draw an edge to a vertex that does not yet exist.

▶ **Observation 16.** *For each equation $(f(\mathbf{x}) = \texttt{expr}) \in \mathcal{E}$, the evaluation of* `expr` *terminates when provided with the values of all relevant function calls.*

Indeed, each `expr` is defined by a node type, all of which are finite computations listed in previous work [6, 28, 31].

▶ **Corollary 17.** *If f is a non-recursive function with no function calls on the RHS of its definition, then the evaluation of any function call $f(\mathbf{x})$ terminates.*

[1] Note that characterising the fine-grained complexity of the solutions found by Crane2 or other FOMC algorithms is an emerging area of research. These questions have been partially addressed in previous work [6, 24] and are unrelated to the goals of this section.

▶ **Observation 18.** *For each equation* $(f(\mathbf{x}) = \texttt{expr}) \in \mathcal{E}$*, if* $\mathbf{x}$ *contains only constants, then* `expr` *cannot include any function calls to* f*.*

Additionally, we introduce an assumption about the structure of recursion.

▶ **Assumption 19.** *For each equation* $(f(\mathbf{x}) = \texttt{expr}) \in \mathcal{E}$*, every recursive function call* $f(\mathbf{y}) \in \texttt{expr}$ *satisfies the following:*

- *each* y_i *is either* $x_i - c_i$ *or* c_i *for some constant* c_i *and*
- *there exists* i *such that* $y_i = x_i - c_i$ *for some* $c_i > 0$*.*

Finally, we assume a particular order of evaluation for function calls using the equations in $\mathcal{E}$; specifically, base cases precede the recursive definition.

▶ **Assumption 20.** *When multiple equations in* $\mathcal{E}$ *match a function call* $f(\mathbf{x})$*, preference is given to the equation with the most constants on its LHS.*

With the observations and assumptions mentioned above, we are ready to prove Theorem 13. For readability, we divide the proof into several lemmas of increasing generality.

▶ **Lemma 21.** *Assume that* $\mathcal{E}$ *consists of just* one unary *function called* f*. Then the evaluation of* $f(x)$ *terminates for any* $x \in \mathbb{N}_0$*.*

Proof. If $f(x)$ matches a base case, then its evaluation terminates by Corollary 17 and Observation 18. If f is not recursive, the evaluation of $f(x)$ also terminates by Corollary 17.

Otherwise, let $f(y)$ be an arbitrary function call on the RHS of the definition of $f(x)$. If y is a constant, then there is a base case for $f(y)$. Otherwise, let $y = x - c$ for some $c > 0$. Then there exists $k \in \mathbb{N}_0$ such that $0 \leq x - kc \leq c - 1$. So, after k iterations, the sequence of function calls $f(x), f(x - c), f(x - 2c), \ldots$ will match the base case $f(x \mod c)$. ◀

▶ **Lemma 22.** *Generalising Lemma 21, let* $\mathcal{E}$ *be a set of equations for* one n*-ary function* f *for some* $n \geq 1$*. The evaluation of* $f(\mathbf{x})$ *terminates for any* $\mathbf{x} \in \mathbb{N}_0^n$*.*

Proof. If f is non-recursive, the evaluation of $f(\mathbf{x})$ terminates by previous arguments. We proceed by induction on n, with the base case of $n = 1$ handled by Lemma 21. Assume that $n > 1$. Any base case of f can be seen as a function of arity $n - 1$ since one of the parameters is constant. Thus, the evaluation of any base case terminates by the inductive hypothesis. It remains to show that the evaluation of the recursive equation for f terminates, but this follows from Observation 16. ◀

Now that we have handled the special case when $\mathcal{E}$ defines one function, we are ready to prove the main theorem.

Proof of Theorem 13. We proceed by induction on the number of functions n. The base case of $n = 1$ is handled by Lemma 22. Let $(f_i)_{i=1}^n$ be some topological ordering of these $n > 1$ functions. If $f = f_j$ for $j < n$, then the evaluation of $f(\mathbf{x})$ terminates by the inductive hypothesis since f_j cannot call f_n by Observation 15. Using the inductive hypothesis that all function calls to f_j (with $j < n$) terminate, the proof proceeds similarly to the proof of Lemma 22. ◀

Algorithm 3 `Propagate`(ϕ, Δ, n).

Input: sentence ϕ, domain Δ, $n \in \mathbb{N}_0$
Output: sentence ϕ'

```
1  φ' ← ∅;
2  if n = 0 then
3      foreach clause C ∈ φ do
4          if Δ ∉ Doms(C) then φ' ← φ' ∪ { C };
5          else
6              C' ← { l ∈ C | Δ ∉ Doms(l) };
7              if C' ≠ ∅ then
8                  l ← an arbitrary literal in C';
9                  φ' ← φ' ∪ { C' ∪ { ¬l } };
10 else
11     D ← a set of n new constants in Δ;
12     foreach clause C ∈ φ do
13         (x_i)_{i=1}^m ← the variables in C with domain Δ;
14         if m = 0 then φ' ← φ' ∪ { C };
15         else φ' ← φ' ∪ { C[x_1 ↦ c_1, ..., x_m ↦ c_m] | (c_i)_{i=1}^m ∈ D^m };
```

3.3 Propagating Domain Size Assumptions

Algorithm 3, called `Propagate`, modifies the sentence ϕ based on the assumption that $|\Delta| = n$. When $n = 0$, some clauses become vacuously satisfied and can be removed. When $n > 0$, partial grounding replaces all variables with domain Δ with constants. (None of the sentences examined in this work had $n > 1$.) Algorithm 3 handles these two cases separately. For a literal or clause C, we write the set of corresponding domains as $\text{Doms}(C)$. In the case of $n = 0$, there are three types of clauses to consider:

1. those that do not mention Δ,
2. those in which every literal contains variables quantified over Δ and
3. those with some literals containing such variables and some without.

We transfer clauses of Type 1 to the new sentence ϕ' without any changes. For clauses of Type 2, C' is empty, so these clauses are filtered out. As for clauses of Type 3, lines 7–9 perform a new kind of smoothing, the explanation of which we defer to Section 3.4.

In the case of $n > 0$, n new constants are introduced. Let C be an arbitrary clause in ϕ, and let $m \in \mathbb{N}_0$ be the number of variables in C quantified over Δ. If $m = 0$, C is added directly to ϕ'. Otherwise, a clause is added to ϕ' for every possible combination of replacing the m variables in C with the n new constants.

▶ **Example 23.** Let $C := \forall x \in \Gamma.\ \forall y, z \in \Delta.\ \neg P(x, y) \vee \neg P(x, z) \vee y = z$. Then $\text{Doms}(C) = \text{Doms}(\neg P(x, y)) = \text{Doms}(\neg P(x, z)) = \{\Gamma, \Delta\}$, and $\text{Doms}(y = z) = \{\Delta\}$. A call to `Propagate`$(\{C\}, \Delta, 3)$ would result in the following sentence with nine clauses:

$$
\begin{aligned}
&(\forall x \in \Gamma.\ \neg P(x, c_1) \vee \neg P(x, c_1) \vee c_1 = c_1) \wedge \\
&(\forall x \in \Gamma.\ \neg P(x, c_1) \vee \neg P(x, c_2) \vee c_1 = c_2) \wedge \\
&\qquad\qquad\qquad\qquad \vdots \\
&(\forall x \in \Gamma.\ \neg P(x, c_3) \vee \neg P(x, c_3) \vee c_3 = c_3).
\end{aligned}
$$

Here, c_1, c_2, and c_3 are the new constants.

3.4 Smoothing the Base Cases

Smoothing modifies a circuit to reintroduce eliminated atoms, ensuring the correct model count [5, 31]. This section describes a similar process performed on lines 7–9 of Algorithm 3. Line 7 checks if smoothing is necessary, and lines 8 and 9 execute it. If the condition on line 7 is not satisfied, the clause is not smoothed but omitted.

Suppose `CompileWithBaseCases` calls `Propagate` with arguments $(\phi, \Delta, 0)$, i.e., we are simplifying the sentence ϕ by assuming that the domain Δ is empty. Informally, if there is a predicate P in ϕ unrelated to Δ, smoothing preserves all occurrences of P, even if all clauses with P become vacuously satisfied.

▶ **Example 24.** Let ϕ be

$$(\forall x \in \Delta.\ \forall y, z \in \Gamma.\ Q(x) \vee P(y, z)) \wedge \tag{5}$$
$$(\forall y, z \in \Gamma'.\ P(y, z)), \tag{6}$$

where $\Gamma' \subseteq \Gamma$ is a domain introduced by a compilation rule. Note that P, as a relation, is a subset of $\Gamma \times \Gamma$.

Now, let us reason manually about the model count of ϕ when $\Delta = \emptyset$. Predicate Q can only take one value, $Q = \emptyset$. The value of P is fixed over $\Gamma' \times \Gamma'$ by clause (6), but it can vary freely over $(\Gamma \times \Gamma) \setminus (\Gamma' \times \Gamma')$ since clause (5) is vacuously satisfied by all structures. Therefore, the correct FOMC should be $2^{|\Gamma|^2 - |\Gamma'|^2}$. However, without line 9, `Propagate` would simplify ϕ to $\forall y, z \in \Gamma'.\ P(y, z)$. In this case, P is a subset of $\Gamma' \times \Gamma'$. This simplified sentence has only one model: $\{\, P(y, z) \mid y, z \in \Gamma' \,\}$. By including line 9, `Propagate` transforms ϕ to

$$(\forall y, z \in \Gamma.\ P(y, z) \vee \neg P(y, z)) \wedge (\forall y, z \in \Gamma'.\ P(y, z)),$$

which retains the correct model count.

It is worth mentioning that the choice of l on line 8 of Algorithm 3 is inconsequential because any choice achieves the same goal: constructing a tautological clause that retains the literals in C'.

3.5 Generating C++ Code

In this section, we will describe the final step of Crane2 as outlined in Figure 1, i.e., translating the set of equations $\mathcal{E}$ into C++ code. (Note that the compilation does not rely on any unique features of C++; the equations could easily be compiled into any other programming language.) Recall that this step is crucial for the usability of the algorithm; otherwise, function definitions would remain purely mathematical, with no convenient way to compute the model count for particular domain sizes. Once a C++ program is produced, it can be executed with different command-line arguments to determine the model count of the sentence for various domain sizes. A typical C++ program for one of the benchmarks in Section 4 contains about 80–100 lines of code. Moreover, although we focus on FOMC in this work, support for WFOMC could easily be added by using floating-point numbers instead of integers.

See Algorithm 4 for the typical structure of a generated C++ program. Each equation in $\mathcal{E}$ turns into a C++ function with a separate cache for memoisation. Hence, Algorithm 4 has a function and a cache for $f(\cdot, \cdot)$, $g(\cdot, \cdot)$, $g(\cdot, 0)$, and $g(0, \cdot)$. The implementation of an equation consists of three parts. First (on line 5), we check if the arguments already exist in the corresponding cache. If they do, we return the cached value. Second (on lines 6 and 7),

Algorithm 4 A sketch of the C++ program for the equations in Example 5, particularly highlighting the recursive definition of the function g.

```
initialise Cache_{g(0,m)}, Cache_{g(l,0)}, Cache_g, and Cache_f;
Function g_{0,m}(m): ...
Function g_{l,0}(l): ...
Function g(l, m):
    if (l, m) ∈ Cache_g then return Cache_g(l, m);
    if l = 0 then return g_{0,m}(m);
    if m = 0 then return g_{l,0}(l);
    r ← g(l − 1, m) + mg(l − 1, m − 1);
    Cache_g(l, m) ← r;
    return r;
Function f(m, n): ...
Function Main:
    (m, n) ← ParseCommandLineArguments();
    return f(m, n);
```

we check if the arguments match any of the base cases (as defined in Section 2.5). If so, we redirect the arguments to the C++ function for that base case. Finally, if none of the above cases applies, we evaluate the arguments based on the expression on the RHS of the equation, store the result in the cache, and return it.

4 Experimental Evaluation

Our empirical evaluation sought to compare the runtime performance of Crane2 with the current state of the art, namely FastWFOMC[2] [24, 26] and ForcLift[3]. Our experiments involved two versions of Crane2: Crane2-Greedy and Crane2-BFS. Like its predecessor (see Section 2.4.2), Crane2 has two modes for applying compilation rules to sentences: one that uses a greedy search algorithm similar to ForcLift and another that combines greedy and BFS.

The experiments used an Intel Skylake 2.4 GHz CPU with 188 GiB of memory and CentOS 7. We used the Intel C++ Compiler 2020u4 for C++ programs, Julia 1.10.4 for FastWFOMC, and the Java Virtual Machine 1.8.0_201 for ForcLift and Crane2. Although implemented in different languages, Crane2 and FastWFOMC use the same GNU Multiple Precision Arithmetic Library for arbitrary-precision arithmetic. Additionally, while ForcLift could be extended to support arbitrary-precision arithmetic, its compilation rules are not extensive enough to handle some of the benchmarks.

We ran each algorithm on each benchmark using domains of size $2^1, 2^2, 2^3$, and so on until an algorithm failed to handle a domain size due to a timeout (of one hour) or out-of-memory or out-of-precision errors. We verified the accuracy of the numerical answers using the corresponding integer sequences in the On-Line Encyclopedia of Integer Sequences [17].

[2] https://github.com/jan-toth/FastWFOMC.jl
[3] https://github.com/UCLA-StarAI/Forclift

4.1 Benchmarks

We compare these algorithms using three benchmarks from previous work. The first benchmark is the bijection-counting problem from Example 2. The second benchmark is a variant of the well-known *Friends & Smokers* Markov logic network [21, 29], which takes the form of

$$(\forall x, y \in \Delta.\ S(x) \wedge F(x,y) \Rightarrow S(y)) \wedge (\forall x \in \Delta.\ S(x) \Rightarrow C(x)).$$

In this sentence, we have three predicates, S, F, and C, that denote smoking, friendship, and cancer, respectively. The first clause states that friends of smokers are also smokers, and the second clause asserts that smoking causes cancer. Common additions to this sentence include making the friendship relation symmetric and assigning probabilities to each clause. Finally, we include the function-counting problem [6]

$$(\forall x \in \Gamma.\ \exists y \in \Delta.\ P(x,y)) \wedge (\forall x \in \Gamma.\ \forall y, z \in \Delta.\ P(x,y) \wedge P(x,z) \Rightarrow y = z) \tag{7}$$

as our third benchmark. Here, predicate P represents a function from Γ to Δ. The first clause asserts that each x must have at least one corresponding y, while the second clause ensures the uniqueness of such a y.

▶ Remark. We formulate the *Bijections* and *Functions* benchmarks using two domains, Γ and Δ, as this formulation is known to help FOKC algorithms find efficient solutions [6]. To compare Crane2 and ForcLift with FastWFOMC, which has no support for multiple domains, we set $|\Gamma| = |\Delta|$.

Our Benchmarks in C^2 and $\mathsf{UFO}^2 + \mathsf{CC}$

For completeness and reproducibility, let us translate the benchmark sentences from FO to C^2 and $\mathsf{UFO}^2 + \mathsf{CC}$. Since *Friends & Smokers* is a relatively simple sentence, it remains the same in C^2 and $\mathsf{UFO}^2 + \mathsf{CC}$. For *Functions*, in C^2, one would write

$$\forall x \in \Delta.\ \exists^{=1} y \in \Delta.\ P(x,y).$$

In $\mathsf{UFO}^2 + \mathsf{CC}$, the equivalent formulation is

$$(\forall x, y \in \Delta.\ S(x) \vee \neg P(x,y)) \wedge (|P| = |\Delta|), \tag{8}$$

where $w^-(S) = -1$. Although sentence (8) has more models than its counterpart in C^2, the negative weight $w^-(S) = -1$ causes some of the terms in the definition of WFOMC to cancel out. The translation of *Bijections* is similar to that of *Functions*. In C^2, one could write

$$(\forall x \in \Delta.\ \exists^{=1} y \in \Delta.\ P(x,y)) \wedge (\forall y \in \Delta.\ \exists^{=1} x \in \Delta.\ P(x,y)).$$

Similarly, in $\mathsf{UFO}^2 + \mathsf{CC}$, the equivalent formulation is

$$(\forall x, y \in \Delta.\ R(x) \vee \neg P(x,y)) \wedge (\forall x, y \in \Delta.\ S(x) \vee \neg P(y,x)) \wedge (|P| = |\Delta|),$$

where $w^-(R) = w^-(S) = -1$.

The three benchmarks cover a wide range of possibilities. The *Friends & Smokers* benchmark uses multiple predicates and only two variables without employing any constructs specific to only one of the (W)FOMC algorithms. The *Functions* benchmark, on the other hand, can still be handled by all the algorithms, but its formulation relies on algorithm-specific constructs. Namely, for ForcLift and Crane2, the sentence is written using three variables. Since FastWFOMC supports at most two variables, we rewrite the sentence (7) with two variables and a *cardinality constraint*. Lastly, the *Bijections* benchmark is an example of a sentence that FastWFOMC can handle but ForcLift cannot.

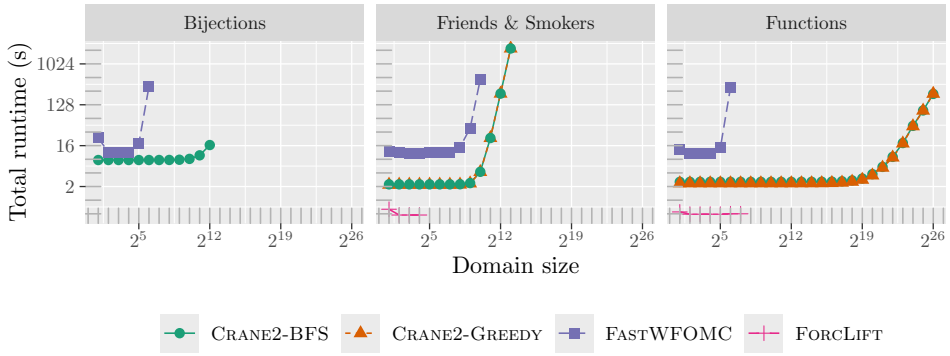


Figure 2 The (total) runtime of the algorithms as a function of domain size. Note that both axes are on a logarithmic scale.

Table 2 The compilation times of FOKC algorithms, sorted in ascending order.

Algorithm	Benchmark	Compilation time (s)
ForcLift	*Functions*	0.44
	Friends & Smokers	0.52
Crane2-Greedy	*Friends & Smokers*	2.20
	Functions	2.36
Crane2-BFS	*Friends & Smokers*	2.23
	Functions	2.53
	Bijections	7.70

4.2 Results

Figure 2 presents a summary of the experimental results. Only FastWFOMC and Crane2-BFS could handle the bijection-counting problem. For this benchmark, the largest domain sizes these algorithms could accommodate were 64 and 4096, respectively. On the other two benchmarks, ForcLift had the lowest runtime. However, since it can only handle model counts smaller than 2^{31}, it only scales up to domain sizes of 16 and 128 for *Friends & Smokers* and *Functions*, respectively. FastWFOMC outperformed ForcLift in the case of *Friends & Smokers*, but not *Functions*, as it could handle domains of size 1024 and 64, respectively. Furthermore, both Crane2-BFS and Crane2-Greedy performed similarly on both benchmarks. Similarly to the *Bijections* benchmark, Crane2 significantly outperformed the other two algorithms, scaling up to domains of size 8192 and 67,108,864, respectively.

One might notice that the runtime of FastWFOMC and ForcLift is slightly higher for the smallest domain size. This peculiarity is the consequence of *just-in-time* (JIT) compilation. As Crane2 is only run once per benchmark, we include the JIT compilation time in its overall runtime across all domain sizes. Additionally, Table 2 summarises the compilation time of each FOKC algorithm on each suitable benchmark. While the compilation time of ForcLift is generally lower compared to Crane2, neither significantly affects the overall runtime.

Based on our experiments, which algorithm should one use in practice? If ForcLift can handle the sentence and the domain sizes are reasonably small, it is likely the fastest algorithm. In other situations, Crane2 will likely be significantly more efficient than FastWFOMC regardless of domain size, provided both algorithms can handle the sentence.

5 Conclusion and Future Work

In this work, we have presented a scalable, automated FOKC-based approach to FOMC. Our algorithm involves completing the definitions of recursive functions and subsequently translating all function definitions into C++ code. Empirical results demonstrate that Crane2 can scale to larger domain sizes than FastWFOMC while supporting a wider range of sentences than ForcLift. The ability to efficiently handle large domain sizes is particularly crucial in the weighted setting, as illustrated by the *Friends & Smokers* example, where the model captures complex social networks with probabilistic relationships. Without this scalability, these models would have limited practical value.

Future directions for research include conducting a comprehensive experimental comparison of FOMC algorithms to better understand their comparative performance across various sentences. The capabilities of Crane2 could also be characterised theoretically, for example, by proving completeness for logic fragments liftable by other algorithms [12, 23, 27]. Additionally, the efficiency of FOMC algorithms can be further analysed using fine-grained complexity, which would provide more detailed insights into the computational demands of different sentences.

References

1 Damiano Azzolini and Fabrizio Riguzzi. Lifted inference for statistical statements in probabilistic answer set programming. *Int. J. Approx. Reason.*, 163:109040, 2023. doi:10.1016/J.IJAR.2023.109040.

2 Jáchym Barvínek, Timothy van Bremen, Yuyi Wang, Filip Zelezný, and Ondřej Kuželka. Automatic conjecturing of P-recursions using lifted inference. In *ILP*, volume 13191 of *Lecture Notes in Computer Science*, pages 17–25. Springer, 2021. doi:10.1007/978-3-030-97454-1_2.

3 Paul Beame, Guy Van den Broeck, Eric Gribkoff, and Dan Suciu. Symmetric weighted first-order model counting. In *PODS*, pages 313–328. ACM, 2015. doi:10.1145/2745754.2745760.

4 Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7):772–799, 2008. doi:10.1016/J.ARTINT.2007.11.002.

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

6 Paulius Dilkas and Vaishak Belle. Synthesising recursive functions for first-order model counting: Challenges, progress, and conjectures. In *KR*, pages 198–207, 2023. doi:10.24963/KR.2023/20.

7 Vibhav Gogate and Pedro M. Domingos. Probabilistic theorem proving. *Commun. ACM*, 59(7):107–115, 2016. doi:10.1145/2936726.

8 Eric Gribkoff, Dan Suciu, and Guy Van den Broeck. Lifted probabilistic inference: A guide for the database researcher. *IEEE Data Eng. Bull.*, 37(3):6–17, 2014. URL: http://sites.computer.org/debull/A14sept/p6.pdf.

9 Manfred Jaeger and Guy Van den Broeck. Liftability of probabilistic inference: Upper and lower bounds. In *StarAI@UAI*, 2012. URL: https://starai.cs.kuleuven.be/2012/accepted/jaeger.pdf.

10 Kristian Kersting. Lifted probabilistic inference. In *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 33–38. IOS Press, 2012. doi:10.3233/978-1-61499-098-7-33.

11 Ananth K. Kidambi, Guramrit Singh, Paulius Dilkas, and Kuldeep S. Meel. Towards practical first-order model counting. *CoRR*, abs/2502.12278, 2025. doi:10.48550/arXiv.2502.12278.

12 Ondřej Kuželka. Weighted first-order model counting in the two-variable fragment with counting quantifiers. *J. Artif. Intell. Res.*, 70:1281–1307, 2021. doi:10.1613/JAIR.1.12320.

13 Sagar Malhotra and Luciano Serafini. Weighted model counting in FO2 with cardinality constraints and counting quantifiers: A closed form formula. In *AAAI*, pages 5817–5824. AAAI Press, 2022. doi:10.1609/AAAI.V36I5.20525.

14 Nils J. Nilsson. Probabilistic logic. *Artif. Intell.*, 28(1):71–87, 1986. doi:10.1016/0004-3702(86)90031-7.

15 Feng Niu, Christopher Ré, AnHai Doan, and Jude W. Shavlik. Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. *Proc. VLDB Endow.*, 4(6):373–384, 2011. doi:10.14778/1978665.1978669.

16 Vilém Novák, Irina Perfilieva, and Jiri Mockor. *Mathematical principles of fuzzy logic*, volume 517. Springer Science & Business Media, 2012.

17 OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences, 2025. Published electronically at http://oeis.org.

18 Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, Giuseppe Cota, and Evelina Lamma. A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics. *Int. J. Approx. Reason.*, 80:313–333, 2017. doi:10.1016/J.IJAR.2016.10.002.

19 Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

20 Dragan Z. Šaletić. Graded logics. *Interdisciplinary Description of Complex Systems: INDECS*, 22(3):276–295, 2024.

21 Parag Singla and Pedro M. Domingos. Lifted first-order belief propagation. In *AAAI*, pages 1094–1099. AAAI Press, 2008. URL: http://www.aaai.org/Library/AAAI/2008/aaai08-173.php.

22 Martin Svatos, Peter Jung, Jan Tóth, Yuyi Wang, and Ondřej Kuželka. On discovering interesting combinatorial integer sequences. In *IJCAI*, pages 3338–3346. ijcai.org, 2023. doi:10.24963/IJCAI.2023/372.

23 Jan Tóth and Ondřej Kuželka. Lifted inference with linear order axiom. In *AAAI*, pages 12295–12304. AAAI Press, 2023. doi:10.1609/AAAI.V37I10.26449.

24 Jan Tóth and Ondřej Kuželka. Complexity of weighted first-order model counting in the two-variable fragment with counting quantifiers: A bound to beat. In *KR*, 2024. doi:10.24963/KR.2024/64.

25 Pietro Totis, Jesse Davis, Luc De Raedt, and Angelika Kimmig. Lifted reasoning for combinatorial counting. *J. Artif. Intell. Res.*, 76:1–58, 2023. doi:10.1613/JAIR.1.14062.

26 Timothy van Bremen and Ondřej Kuželka. Faster lifting for two-variable logic using cell graphs. In *UAI*, volume 161 of *Proceedings of Machine Learning Research*, pages 1393–1402. AUAI Press, 2021. URL: https://proceedings.mlr.press/v161/bremen21a.html.

27 Timothy van Bremen and Ondřej Kuželka. Lifted inference with tree axioms. *Artif. Intell.*, 324:103997, 2023. doi:10.1016/J.ARTINT.2023.103997.

28 Guy Van den Broeck. On the completeness of first-order knowledge compilation for lifted probabilistic inference. In *NIPS*, pages 1386–1394, 2011. URL: https://proceedings.neurips.cc/paper/2011/hash/846c260d715e5b854ffad5f70a516c88-Abstract.html.

29 Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Lifted relax, compensate and then recover: From approximate to exact lifted probabilistic inference. In *UAI*, pages 131–141. AUAI Press, 2012. URL: https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=2349&proceeding_id=28.

30 Guy Van den Broeck, Wannes Meert, and Adnan Darwiche. Skolemization for weighted first-order model counting. In *KR*. AAAI Press, 2014. URL: http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/8012.

31 Guy Van den Broeck, Nima Taghipour, Wannes Meert, Jesse Davis, and Luc De Raedt. Lifted probabilistic inference by first-order knowledge compilation. In *IJCAI*, pages 2178–2185. IJCAI/AAAI, 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-363.

32 Yuanhong Wang, Juhua Pu, Yuyi Wang, and Ondřej Kuželka. Lifted algorithms for symmetric weighted first-order model sampling. *Artif. Intell.*, 331:104114, 2024. doi:10.1016/J.ARTINT.2024.104114.

An Application of SAT Solvers in Integer Programming Games

Pravesh Koirala
Department of Computer Science, Vanderbilt University, Nashville, TN, USA

Aditya Shrey
Department of Computer Science, Vanderbilt University, Nashville, TN, USA

Forrest Laine
Department of Computer Science, Vanderbilt University, Nashville, TN, USA

Abstract

Integer programming games (IPGs) are a popular game-theoretic tool to model an array of games where each player has a discrete strategy set. These games arise in important domains such as economics, transportation, cybersecurity, etc., but solving them is non-trivial as it is known that checking for the existence of pure Nash equilibria in an IPG is Σ_2^p-complete. Recent works have proposed a class of relaxed solution concepts for IPGs called locally optimal integer solutions (LOIS) and shown it to be an efficient alternative for pure Nash equilibria. While LOIS are significantly simpler to compute, they still do not scale when solved using traditional mathematical solvers, especially when high-quality solutions are desired. In this paper, we apply commercially available SAT solvers to find LOIS in IPGs. We investigate efficient encodings for a cybersecurity game and compare solution times when using SAT solvers vs mathematical program solvers. We also investigate the application of SAT solvers in graph games using a graph interdiction example and compare against the obtained LOI solutions against existing heuristics-based solutions. Our results indicate that with appropriate encodings, large-scale IPGs can be solved much more efficiently using SAT solvers. We also show that SAT solvers can be applied to graph games in conjunction with LOIS for obtaining high-quality solutions. Our results emphasize the potential of SAT solvers combined with LOIS to solve significant game theory problems.

2012 ACM Subject Classification Theory of computation → Algorithmic game theory and mechanism design; Security and privacy → Network security

Keywords and phrases Game Theory, Integer Programming Games, SAT Solvers, Local Solutions, Graph Games

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.19

Supplementary Material *Software*: `https://github.com/PraveshKoirala/SATLois`

1 Introduction

Integer programming games (IPGs) are well-known in game-theoretic literature [7, 26]. They arise when every player is restricted to choosing discrete strategies and are useful for modeling a plethora of scenarios from domains such as transportation, communication, cybersecurity, etc [10, 15, 27]. While they provide substantial modeling flexibility, they are known to be difficult to solve. It is well-known that deciding whether or not an IPG admits a pure Nash equilibria, a popular solution concept for IPGs, is Σ_2^p-complete [8]. This poses severe computational challenges and makes solving for large-scale IPGs intractable.

However, recent works have introduced new class of solution concepts for IPGs called *locally optimal integer solutions* (LOIS) that are more relaxed than PNEs and computationally tractable even for large-scale IPGs [25]. In fact, [25] show that deciding whether IPGs admit LOIS is NP-complete, which means that they can be solved by commercially available mathematical solvers that support integer programming and scale well across large IPG

© Pravesh Koirala, Aditya Shrey, and Forrest Laine;
licensed under Creative Commons License CC-BY 4.0

instances. With this relaxed notion of LOIS, many large-scale IPGs that specifically arise in many critical domains, including cybersecurity, become tractable. In particular, in an attacker-defender scenario where there are a large number of cyber infrastructures to be considered with a limited attack/defense budget for both attackers and defenders, the resulting IPG is not readily solvable for large instances using PNE but can indeed be solved using LOIS [25]. Furthermore, a *bilevel* IPG with a leader-follower structure can also be solved by embedding the LOIS optimality conditions of the follower in the leader's program to obtain a single integer program. This allows us to solve an important class of cybersecurity games played on graphs, i.e. *graph interdiction games*, where a defender and an attacker strategically choose nodes to protect and infect while considering the propagation dynamics of the attack on the graph [3]. Despite the scalability of LOIS, they are not without limitations. The most important being that while LOI solutions scale well for lower orders (LOIS-1), higher-order LOI solutions, which are qualitatively better, are time-consuming to calculate due to increased complexity.

SAT solvers that solve the *boolean satisfiability problem* are known to be effective for solving NP-complete problems from various domains including planning, formal verification, cybersecurity, transportation, etc. [2, 20]. Decades of advances in SAT solving techniques, coupled with rigorous optimizations in commercially available SAT solvers make them attractive for solving any large-scale NP-complete problem. Furthermore, the extension of SAT called MaxSAT [28] adds further modeling flexibility to the SAT language, making it possible to perform an *optimization* over the number of true clauses. Despite these powerful modeling and solving capabilities, SAT solvers are underutilized for game-theoretic applications and have not been previously considered for solving IPGs.

In this paper, we show that it is possible to utilize modern SAT solvers to solve for LOI solutions for IPGs using straightforward encodings. In particular, we highlight the fact that modern SAT solvers can scale incredibly well in comparison to traditional mathematical optimizers in certain game-theoretic applications. We apply our methods in two cybersecurity games of interest: a) a critical node game first introduced by [15] where attacker and defender simultaneously choose which infrastructures to attack/defend, and b) a bilevel graph interdiction game with constrained propagation dynamics where defender chooses central nodes to defend followed by the attacker choosing their initial targets to spread infection.

2 Related Works

Local solutions in discrete games. IPGs themself were first investigated by [26] and have found considerable usage since in domains with discrete strategy-space like inventory management, facility location, cybersecurity, etc. [27, 33, 16, 11]. Prevailing methods for solving IPGs focus on finding either pure or mixed Nash equilibrium solutions using a range of algorithms such as branching, cut-and-play, sample generation, zero-regrets etc. [32, 34, 6, 16]. A broad survey of IPGs and related algorithms can be found in [7]. Related to IPGs are also *boolean games* [4, 22], i.e., games where each player's strategy space is composed purely of boolean choices and the players choose assignments with the objective of satisfying their utility (represented by a boolean formula). However, determining the existence of pure Nash equilibria in boolean games is in Σ_2^p as well [4]. The very first work that explored the concept of *local solutions* in discrete strategy games is by [17], where a k-bounded Nash equilibrium is defined for boolean games such that no player can gain utility by unilaterally changing at most k of their strategies. This concept has also been independently explored in a voting game by [29], where they analogously define a *t-local equilibrium* such that no candidate

can gain any utility by unilaterally deviating in at most t of their announced positions. A generalization of this boolean local equilibria concept to IPGs was provided more recently by [25], who define a *locally optimal integer solution* to be one where no player can gain by unilaterally deviating in their *integer neighborhood*. Their formulation allows for deriving local optimality conditions for IPGs with shared constraints and is the focus of this work.

SAT solvers in game theory. Usage of SAT solvers in game-theoretic setting is sparse and has generally been restricted to boolean games with boolean utilities. Notably, [12] use *answer set programming* to solve for pure Nash equilibrium of boolean games. [35] apply SAT and MaxSAT solvers in bounded model checking setting to derive Nash equilibrium in a resource allocation problem. An interesting application of satisfiability theory in bimatrix games is proposed in [9], which uses Horn clauses to identify special substructures of a normal form bimatrix game, potentially speeding the process of obtaining Nash equilibrium solutions. An application of SAT on graphical games is by [23] who use it to solve *parity games*, a special type of two-player game played on graphs with the objective of achieving a desired *parity* of minimum node *priority* in an infinite path of play. Similarly, in *reachability games*, SAT solvers have been used to determine and extract winning strategies [18].

Apart from [9], the abovementioned works do not apply to games with non-boolean objectives or constraints, which IPGs specifically allow. And even [9] is an application of SAT to *identify* potential substructures in discrete games that may simplify solutions (and is not a solution method by itself). Therefore, to our knowledge, our work is the first to investigate the application of SAT solvers on IPGs. In particular, we investigate solutions of both simultaneous IPGs (where players make decisions at once) and sequential (or bilevel) IPGs where a leader makes a decision followed by the follower choosing their best response. In summary, our contributions are as follows [1]:

1. We add to the literature of game theory by investigating SAT solvers as an alternative solution method to a class of games of broad interest (i.e., IPGs), and compare their performance against traditional mathematical solvers.
2. We investigate the application of MaxSAT solvers on bilevel graph games and compare the quality of solutions against prevailing heuristics.

Rest of the paper is organized as follows: In section 3 we start with the overview of IPGs, including its definition and related solution concepts. In sections 4 and 5, we introduce the critical node game and the bilevel graph interdiction game and discuss their encoding approaches. In section 6, we discuss timing comparison for mathematical solvers vs SAT solvers for the critical node game and also analyze solution quality of LOIS over heuristics for the bilevel graph game. We briefly recap our work and provide future directions in section 7.

3 Definitions

Throughout the rest of this paper, we follow the conventions and definitions used by [25].

▶ **Definition 1** (IPG). *IPGs with n players is defined as a simultaneous, complete-information, and non-cooperative game in the form of a n-tuple* $(P_1, P_2, ...P_n)$*, where each* P_i *is a mathematical program of the form:*

[1] The repository for this paper can be found at `https://github.com/PraveshKoirala/SATLois`

$$P_i := \min_{x^i} \; f^i(x^i; x^{-i})$$
$$s.t. \;\; g^i(x^i) \geq 0, \tag{1}$$

where $x^i \in \mathbb{Z}^{n_i}$ is the strategy of player i, $f^i : \mathbb{Z}^N \mapsto \mathbb{R}$ is their objective function with $N = \sum_i n_i$, and $g^i : \mathbb{Z}^{n_i} \mapsto \mathbb{R}^{m_i}$ are the constraint functions. With $x \in \mathbb{Z}^N$ as the joint strategy space, we use x^{-i} to denote all other decision variables sans x^i.

With definition 1 in place, the pure Nash equilibrium (PNE) of an IPG is defined as:

▶ **Definition 2** (PNE for IPGs). *A joint strategy $x^* \in \mathbb{Z}^N$ is a pure Nash equilibrium for an IPG if and only if $\forall i, g^i(x^*) \geq 0$ and for all players i the following holds:*

$$f^i(x^*) \leq f^i(\tilde{x}^i; (x^*)^{-i}) \;\; \forall \tilde{x}^i \in \{x' \in \mathbb{Z}^{n_i} | g^i(x') \geq 0\}$$

Since PNE is computationally prohibitive due to them being in Σ_2^p, [25] introduce the notion of *m-order integer neighborhood* and subsequently *locally optimal integer solution* as follows:

▶ **Definition 3** (m-order integer neighborhood). *For a point $x \in \mathbb{Z}^n$, we define its m-order integer neighborhood as $\mathcal{Z}_m(x) = \{x' \in \mathbb{Z}^n \mid \|x - x'\|_1 \leq m\}$ where the operator $\|\cdot\|_1$ denotes the L1-norm.*

▶ **Definition 4** (Locally optimal integer solution of m-th order). *For a parameterized integer mathematical program $P := \min_{x \in \mathbb{Z}^n} f(x; a)$ s.t. $x \in C$ where the payoff function f is parameterized in $a \in \mathbb{Z}^m$ and C is some feasible set, $\hat{x}^m \in C$ is said to be a locally optimal integer solution of m-th order (LOIS-m) if and only if $f(\hat{x}^m; a) \leq f(x'; a) \;\; \forall x' \in \mathcal{Z}_m(\hat{x}^m) \cap C$.*

By using definition 4, optimality conditions for LOIS can be obtained for a single program as follows:

▶ **Definition 5** (Optimality conditions for LOIS-m). *For a program parameterized in $a \in \mathbb{Z}^p$, where $f : \mathbb{Z}^{n+p} \mapsto \mathbb{R}$ and $\forall j \in 1..J, \; g_j : \mathbb{Z}^n \mapsto \mathbb{R}$, the optimality conditions for $\hat{x}$ to be a LOIS-m can be outlined as the following inequality and implication constraints:*

$$\bigwedge_{j=1}^{J} \left(g_j(\hat{x}) \geq 0\right), \tag{2}$$

$$\forall \left(\hat{x} + \delta \in \mathcal{Z}_m(\hat{x})\right) \quad \left[f(\hat{x}; a) > f(\hat{x} + \delta; a) \rightarrow \bigvee_{j=1}^{J} \left(g_j(\hat{x} + \delta) < 0\right) \right] \tag{3}$$

Finally, using definition 5, the joint optimality condition for a LOIS-m solution for an IPG is defined as:

▶ **Definition 6** (LOIS-m solution for IPGs). *We recall IPGs to be the joint mathematical program $(P_1, ...P_n)$ where each program P_i is parameterized in x^{-i}. For each player i, let $\mathcal{O}_m^i$ be the set of points satisfying their LOIS-m optimality conditions as introduced in definition 5. Then, we define $\hat{x}$ as the LOIS-m solution for the IPG iff:*

$$\hat{x} \in \bigcap_{i=1}^{n} \mathcal{O}_m^i \tag{4}$$

In simpler terms, a *LOIS-m* solution for an IPG is defined such that no single player, by unilaterally deviating in their m-order integer neighborhood, can obtain better payoffs (or lower costs) while remaining feasible. In this way, LOIS-m basically offers a direct adaptation of the concept of *local equilibria* in continuous-strategy games to IPGs. Similarly, when the discrete strategy-set of each player is binary (and the payoff is a logical formula defined over the joint strategy), it is easy to see that LOIS-m for *unconstrained* IPG in such cases translates to the notion of local equilibria for boolean games. In this regard, LOIS-m is actually a generalization of existing local solution concepts from boolean games.

4 Critical Node Game (CNG)

CNG is a cybersecurity game first introduced by [15]. The following description of CNG is derived from the same and we refer readers to the original text for any missing details. The CNG is played over a set of V critical nodes representing sensitive digital infrastructures with the decision variables $x_i \in \{0,1\}, \alpha_i \in \{0,1\}\ \forall i \in V$ indicating respectively the defender and attacker's choice with $x_i = 1$ for defender defending the node i (and zero otherwise), and $\alpha_i = 1$ for the attacker attacking the target (and zero otherwise). Both players have limited *budget* and there are strategic interactions between their choices. The complete 2-player CNG is specified as a simultaneous, non-cooperative, and complete-information game comprised of the programs (P_A, P_D) such that the attacker solves:

$$P_A := \max_{\alpha \in \{0,1\}^{|V|}} f^a(\alpha; x) \quad s.t.\ a^T\alpha \leq A$$

and the defender solves:

$$P_D := \max_{x \in \{0,1\}^{|V|}} f^d(x; \alpha) \quad s.t.\ d^T x \leq D$$

The functions $f^a : \mathbb{Z}^{2|V|} \mapsto \mathbb{R},\ f^d : \mathbb{Z}^{2|V|} \mapsto \mathbb{R}$ are the payoffs of the attacker and the defender respectively. Each attack or defend choice has a cost represented by the vectors $a \in \mathbb{Z}_+^{|V|}, d \in \mathbb{Z}_+^{|V|}$. The total budget for attacker / defender, respectively, is $A \in \mathbb{Z}_+, D \in \mathbb{Z}_+$ and $p_i^d \in \mathbb{Z}_+$, $p_i^a \in \mathbb{Z}_+$ denote the *criticality* of node i for respectively, the defender and the attacker. This formulation varies slightly from the original, as it restricts the costs, criticality, and budget to be integers instead of reals. But as we show later, our encoding scheme can be suitably modified to account for real costs, criticality, and budget as well. The variables $0 \leq \delta, \eta, \epsilon, \gamma \leq 1$ are then chosen as *real-valued* scalar parameters of a CNG with $\delta < \eta < \epsilon$. The full payoff functions for attacker and defender are described in detail in appendix A

4.1 LOIS for CNG

We outline the LOIS-1 optimality conditions for CNG. Optimality conditions for LOIS-m ($m > 1$) are analogously defined. $(\hat{\alpha}, \hat{x})$ are LOIS-1 solutions for CNG iff the following hold.

For the attacker,

$$\begin{aligned} a^T\hat{\alpha} &\leq A \\ 0 \leq \hat{\alpha}_i &\leq 1,\ \forall i \in V \\ \Delta f_i^a > 0 &\rightarrow (\hat{\alpha}_i > 0) \vee (a^T\hat{\alpha} + a_i\hat{\alpha}_i > A),\ \forall i \in V \\ \Delta f_i^a < 0 &\rightarrow (\hat{\alpha}_i < 1) \vee (a^T\hat{\alpha} - a_i\hat{\alpha}_i > A),\ \forall i \in V \\ \text{where, } \Delta f_i^a &= p_i^a(1 + \gamma + \hat{x}_i(\gamma - \eta)) \end{aligned} \tag{5}$$

For the defender,

$$\begin{aligned} d^T\hat{x} &\leq D \\ 0 \leq \hat{x}_i &\leq 1,\ \forall i \in V \\ \Delta f_i^d > 0 &\rightarrow (\hat{x}_i > 0) \vee (d^T\hat{x} + d_i\hat{x}_i > D),\ \forall i \in V \\ \Delta f_i^d < 0 &\rightarrow (\hat{x}_i < 1) \vee (d^T\hat{x} - d_i\hat{x}_i > D),\ \forall i \in V \\ \text{where, } \Delta f_i^d &= p_i^d(\epsilon - 1 + \hat{\alpha}_i(\eta + 1 - \epsilon - \delta)) \end{aligned} \tag{6}$$

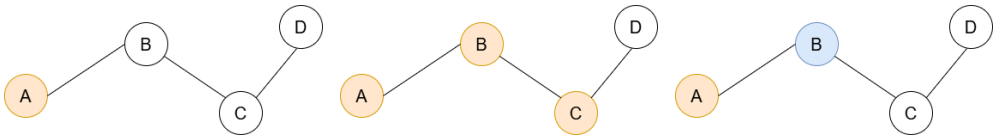
Conditions 5 and 6 when jointly solved, give the LOIS-1 solution $(\hat{\alpha}, \hat{x})$.

4.2 SAT encodings for CNG

The variables for player choices $(\hat{\alpha}, \hat{x})$, attack and defense budgets (A, D), and priority values for each node for attacker and defender (p_i^a, p_i^d) in equations 5 and 6 are integers. For SAT solvers to be able to solve formulas with integer variables, they have to be converted into equivalent fixed-width binary representation, i.e., *bit-vectors* with the process colloquially known as *bit-blasting*. Once equivalent bit-vector representations are obtained, regular arithmetic operations like addition, subtraction, multiplication, and division can be performed using binary circuits. For the parameters $\epsilon, \eta, \delta, \gamma$ that are reals, we encode each of them as ratios a/b where each a, b is a bit-vector of sufficient size. It can be seen that the choice of size needs to be balanced as a larger bit-vector size will result in increased memory and computation, whereas a smaller bit-vector size may result in poor results. Similarly, while we have considered the criticality values, costs, and budgets to be integers, we can substitute them with analogous bit-vector ratios to add support for reals. Once all the variables have been properly encoded, since the operations on them are a combination of simple logical and arithmetic operators (inequality, disjunction, implication, etc.), natively supported by most commercial SAT solvers, these do not require further specialized encoding.

5 Bilevel graph interdiction

We now describe the bilevel graph interdiction game with two players attacker and the defender, with the defender as the leader. The following description of the game is a simplified version of the trilevel game introduced in [31] but with a constrained propagation dynamics. The game is played over an undirected graph $G = (V, E)$ where each node $v \in V$ is a critical digital infrastructure and each edge $(v_1, v_2) \in E$ represents a network connection between v_1, v_2. The objective of the defender (which moves first) is to choose $n_D \in \mathbb{Z}_+$ nodes to *protect*, followed by the attacker that chooses $n_A \in \mathbb{Z}_+$ nodes to *infect* with malware. We assume a constrained propagation model where an attacked node can propagate the malware to *undefended* nodes at a distance of at most r before being detected, making them unsafe, with the distance between a node a and b being defined as the number of edges in the shortest path between a and b. Figure 1 further explains the setup. The full formulation of the problem and its equivalent reduction to a maxSAT form is provided in Appendix B.



(a) Undefended T=0. **(b)** Undefended T=∞. **(c)** Defended T=0/∞.

Figure 1 For a propagation radius $r = 2$. In (a) A is first attacked (orange) and the attack propagates to B and C as shown in (b). D remains safe. In (c), A is first infected but the presence of a defended node B (blue) deters further propagation making B, C, D safe. If there were an edge (A,C) in (c), both C and D would have been eventually infected and B would be the only safe node.

6 Experiments

6.1 CNG

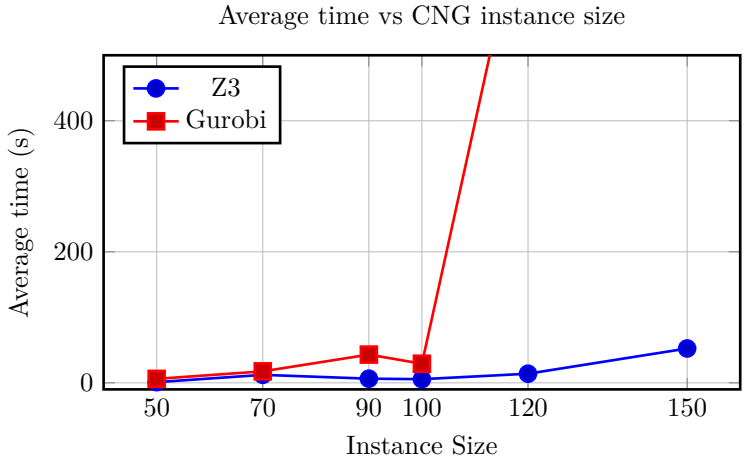


Figure 2 Performance of SAT solver (Z3) and mathematical solver (Gurobi) on CNG instances.

In this experiment, we compare the time taken to obtain higher-order LOIS-2 solutions for increasing CNG instances by popular mathematical program solvers versus a SAT solver. In particular, we choose three mathematical solvers, i.e., HiGHS [24], SCIP [1], and Gurobi [21], to compare the SAT solver of our choice, i.e,. Z3 [13]. We conduct this experiment in two steps. In the preliminary step, we compare the average solution time of HiGHS, SCIP, and Gurobi on 10 random instances of CNG of size 50 generated according to the procedure outlined in [15]. We found the average time for each solver to be HiGHS (991.4s), SCIP(133.2s), and Gurobi(6.05s). As we can see, Gurobi clearly outperforms all other solvers by a large margin. Therefore, in the next step, we exclusively compare Gurobi against Z3 for LOIS-2. We used a bit-vector width of 16 for integer encodings. The results for this step are presented in figure 2. As we can see, Z3 clearly outperforms Gurobi on all instance sizes. Primarily, as the instance size passes a certain threshold, the performance gap becomes starkly visible with Z3 taking an average time of 13.99s for an instance of size 120, whereas Gurobi takes 765.2s for the same.

6.2 Bilevel graph interdiction

For this experiment, we compare the *quality* of results obtained using LOIS-1 solutions against those obtained using prevailing heuristics. In particular, for graphs, metrics like *centrality* measure how important a node is for propagation [14]. Therefore, a heuristic-based method for the bilevel graph interdiction problem is for the defender to defend the most *central* nodes with the highest centrality values, followed by the attacker choosing its best response. We compare the LOIS-1 solution against popular centrality measures in graphs, including degree centrality, eigenvalue centrality, closeness centrality, and betweenness centrality [14]. We also compare with an additional game-theoretic centrality metric known as the *shapley centrality* [30] that is specifically developed for identifying nodes most important for propagation. We consider a modestly sized graph with 30 nodes. We set the defender's budget $n_D = 4$, attacker's budget $n_A = 2$, propagation radius $r = 4$, and evaluate our methods on four kinds of randomly generated graphs i.e. *MST, Line, Edge30*, and *Ring*, where *MST* instances are random minimum spanning tree defined over the nodes, *Edge30* instances contain exactly 30 random edges, *Line* is a path graph [5], and *Ring* is a cycle graph [19]. All solutions are obtained by using the maxSAT solver from Z3 within the time limit of 5 minutes.

Table 1 Number of safe nodes in bilevel graph interdiction game. LOIS-1-based method outperforms all heuristics in every graph variant. Results are averaged over 20 runs for each graph type.

Methods	Line	MST	Ring	Exact30
Betweenness	12	13.8	12	14.75
Closeness	12	13.9	12	13.8
Degree	12	17.35	12	16.75
Eigenvalue	12	13.3	12	13.65
Shapley	12	13.85	12	15.25
LOIS-1	**16.55**	**19.5**	**13.7**	**17.65**

As we can see from the results, when central nodes are obtained using the maxSAT solution of the defender with embedded LOIS-1 optimality constraints of the attacker, it outperforms all popular centrality heuristics. This difference is particularly stark when there are symmetries present in the graph, i.e. *line, ring*. We hypothesize that for symmetric graphs, since all nodes have the same centrality, heuristic-based methods fail to identify key nodes to protect.

7 Conclusion

In this study, we showed that by using local solution concepts like the LOIS, it is possible to solve for IPGs using SAT solvers in both simultaneous and sequential discrete-strategy games. In particular, we note that the encodings required for solving these games are straightforward and are supported out-of-the-box by most commercially available SAT solvers. We also showed that these SAT solvers may be more efficient in terms of solution time when compared against state-of-the-art mathematical solvers like Gurobi, and the game theory community should definitely consider these solvers for large-scale discrete games as an alternative to mathematical program solvers. Finally, we also showed that in graph games, maxSAT solvers in conjunction with LOI solutions can provide better quality solutions with respect to heuristic-based solutions, even for tricky problem instances.

We note the following limitations of our study. First and foremost, only local solution concepts like LOIS remain solvable by SAT solvers. Similarly, games with non-linear objectives or constraints that are not encodable by binary circuits may not be possible to solve. In the bilevel graph interdiction example, although we were able to solve for moderate graph instances, we found scaling it to large graph sizes challenging. Future work seeking to expand this line of research should focus on global solution concepts (or higher-order LOI solutions) for IPGs using SAT and finding novel ways to scale graph games when solved using MaxSAT.

References

1 Tobias Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1:1–41, 2009. doi:10.1007/S12532-008-0001-1.

2 Sahel Alouneh, Sa'ed Abed, Mohammad H Al Shayeji, and Raed Mesleh. A comprehensive study and analysis on sat-solvers: advances, usages and achievements. *Artificial Intelligence Review*, 52:2575–2601, 2019. doi:10.1007/S10462-018-9628-0.

3 Andrea Baggio, Margarida Carvalho, Andrea Lodi, and Andrea Tramontani. Multilevel approaches for the critical node problem. *Operations Research*, 69(2):486–508, 2021. doi:10.1287/OPRE.2020.2014.

4 Elise Bonzon, Marie-Christine Lagasquie-Schiex, Jérôme Lang, and Bruno Zanuttini. Boolean games revisited. In *ECAI*, volume 141, pages 265–269, 2006. URL: http://www.booksonline.iospress.nl/Content/View.aspx?piid=1688.

5 Haitze J Broersma and Cornelis Hoede. Path graphs. *Journal of graph theory*, 13(4):427–444, 1989. doi:10.1002/JGT.3190130406.

6 Margarida Carvalho, Gabriele Dragotto, Andrea Lodi, and Sriram Sankaranarayanan. The cut-and-play algorithm: Computing nash equilibria via outer approximations. *arXiv preprint arXiv:2111.05726*, 2021. arXiv:2111.05726.

7 Margarida Carvalho, Gabriele Dragotto, Andrea Lodi, and Sriram Sankaranarayanan. Integer programming games: a gentle computational overview. In *Tutorials in Operations Research: Advancing the Frontiers of OR/MS: From Methodologies to Applications*, pages 31–51. INFORMS, 2023.

8 Margarida Carvalho, Andrea Lodi, and João P Pedroso. Computing equilibria for integer programming games. *European Journal of Operational Research*, 303(3):1057–1070, 2022. doi:10.1016/J.EJOR.2022.03.048.

9 Vincent Conitzer and Tuomas Sandholm. A technique for reducing normal-form games to compute a nash equilibrium. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 537–544, 2006. doi:10.1145/1160633.1160731.

10 Tobias Crönert and Stefan Minner. Location selection for hydrogen fuel stations under emerging provider competition. *Transportation Research Part C: Emerging Technologies*, 133:103426, 2021.

11 Tobias Crönert and Stefan Minner. Equilibrium identification and selection in finite games. *Operations Research*, 72(2):816–831, 2024. doi:10.1287/OPRE.2022.2413.

12 Sofie De Clercq, Kim Bauters, Steven Schockaert, Mihail Mihaylov, Ann Nowé, and Martine De Cock. Exact and heuristic methods for solving boolean games. *Autonomous agents and multi-agent systems*, 31:66–106, 2017. doi:10.1007/S10458-015-9313-5.

13 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

14 Paramita Dey, Subhayan Bhattacharya, and Sarbani Roy. A survey on the role of centrality as seed nodes for information propagation in large scale network. *ACM/IMS Transactions on Data Science*, 2(3):1–25, 2021. doi:10.1145/3465374.

15 Gabriele Dragotto, Amine Boukhtouta, Andrea Lodi, and Mehdi Taobane. The critical node game. *Journal of Combinatorial Optimization*, 47(5):74, 2024. doi:10.1007/S10878-024-01173-3.

16 Gabriele Dragotto and Rosario Scatamacchia. The zero regrets algorithm: Optimizing over pure nash equilibria via integer programming. *INFORMS Journal on Computing*, 35(5):1143–1160, 2023. doi:10.1287/IJOC.2022.0282.

17 Paul E Dunne and Michael Wooldridge. Towards tractable boolean games. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 939–946, 2012. URL: http://dl.acm.org/citation.cfm?id=2343831.

18 Niklas Eén, Alexander Legg, Nina Narodytska, and Leonid Ryzhyk. Sat-based strategy extraction in reachability games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29(1), 2015.

19 Severino V Gervacio. Cycle graphs. In *Graph Theory Singapore 1983: Proceedings of the First Southeast Asian Graph Theory Colloquium, held in Singapore May 10–28, 1983*, pages 279–293. Springer, 1984.

20 Weiwei Gong and Xu Zhou. A survey of sat solver. In *AIP Conference Proceedings*, volume 1836(1). AIP Publishing, 2017.

21 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024. URL: https://www.gurobi.com.

22 Paul Harrenstein, Wiebe van der Hoek, John-Jules Meyer, and Cees Witteveen. Boolean games. In *Proceedings of the 8th Conference on Theoretical Aspects of Rationality and Knowledge*, pages 287–298, 2001.

23 Keijo Heljanko, Misa Keinänen, Martin Lange, and Ilkka Niemelä. Solving parity games by a reduction to sat. *Journal of Computer and System Sciences*, 78(2):430–440, 2012. doi:10.1016/J.JCSS.2011.05.004.

24 Qi Huangfu and JA Julian Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018. doi:10.1007/S12532-017-0130-5.

25 Pravesh Koirala, Mel Krusniak, and Forrest Laine. Locally optimal solutions for integer programming games. *arXiv preprint arXiv:2503.20918*, 2025. doi:10.48550/arXiv.2503.20918.

26 Matthias Köppe, Christopher Thomas Ryan, and Maurice Queyranne. Rational generating functions and integer programming games. *Operations research*, 59(6):1445–1460, 2011. doi:10.1287/OPRE.1110.0964.

27 Alejandro Lamas and Philippe Chevalier. Joint dynamic pricing and lot-sizing under competition. *European Journal of Operational Research*, 266(3):864–876, 2018. doi:10.1016/J.EJOR.2017.10.026.

28 Chu Min Li and Felip Manya. Maxsat, hard and soft constraints. In *Handbook of satisfiability*, pages 903–927. IOS Press, 2021. doi:10.3233/FAIA201007.

29 Javier Maass, Vincent Mousseau, and Anaëlle Wilczynski. A hotelling-downs game for strategic candidacy with binary issues. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, pages 2076–2084, 2023. doi:10.5555/3545946.3598881.

30 Tomasz P Michalak, Karthik V Aadithya, Piotr L Szczepanski, Balaraman Ravindran, and Nicholas R Jennings. Efficient computation of the shapley value for game-theoretic network centrality. *Journal of Artificial Intelligence Research*, 46:607–650, 2013. doi:10.1613/JAIR.3806.

31 Adel Nabli, Margarida Carvalho, and Pierre Hosteins. Complexity of the multilevel critical node problem. *Journal of Computer and System Sciences*, 127:122–145, 2022. doi:10.1016/J.JCSS.2022.02.004.

32 Simone Sagratella. Computing all solutions of nash equilibrium problems with discrete strategy sets. *SIAM Journal on Optimization*, 26(4):2190–2218, 2016. doi:10.1137/15M1052445.

33 Simone Sagratella, Marcel Schmidt, and Nathan Sudermann-Merx. The noncooperative fixed charge transportation problem. *European Journal of Operational Research*, 284(1):373–382, 2020. doi:10.1016/J.EJOR.2019.12.024.

34 Stefan Schwarze and Oliver Stein. A branch-and-prune algorithm for discrete nash equilibrium problems. *Computational Optimization and Applications*, 86(2):491–519, 2023. doi:10.1007/S10589-023-00500-4.

35 Nils Timm, Josua Botha, and Steven Jordaan. Max-sat-based synthesis of optimal and nash equilibrium strategies for multi-agent systems. *Science of Computer Programming*, 228:102946, 2023. doi:10.1016/J.SCICO.2023.102946.

A Payoff description of CNG

The full payoffs for four distinct scenarios are taken as:

1. **Normal operation ($x_i = 0, \alpha_i = 0$)**: Defender gets full payoff p_i^d, while attacker incurs opportunity cost γp_i^a.
2. **Successful attack ($x_i = 0, \alpha_i = 1$)**: Attacker obtains full payoff p_i^a. Defender obtains reduced δp_i^d.
3. **Successful defense ($x_i = 1, \alpha_i = 0$**: Defender obtains reduced ϵp_i^d. Attacker obtains 0.
4. **Attack and defense ($x_i = 1, \alpha_i = 1$)**: Attacker receives $(1-\eta)p_i^a$. Defender receives ηp_i^d.

The full attacker payoff is then obtained as:

$$f^a(\alpha; x) = \sum_{i \in V} (-\gamma p_i^a (1 - x_i)(1 - \alpha_i) + p_i^a (1 - x_i)\alpha_i + (1 - \eta) p_i^a x_i \alpha_i)$$

Similarly, the full defender payoff is:

$$f^d(x; \alpha) = \sum_{i \in V} \left(p_i^d (1 - x_i)(1 - \alpha_i) + \delta p_i^d (1 - x_i)\alpha_i + \epsilon p_i^d x_i (1 - \alpha_i) + \eta p_i^d x_i \alpha_i\right)$$

B Formulation of bilevel interdiction game

For each node $v \in V$, we use the binary variables $a_v \in \{0,1\}^{|V|}, d_v \in \{0,1\}^{|V|}, s_v \in \{0,1\}^{|V|}$ to represent if the node is attacked, defended, or safe. The bilevel program to be solved for by the leader (defender) can then be outlined as follows[2].

[2] For ease of understanding, we use boolean operations in constraints. These can be equivalently translated to integer constraints for an IPG.

$$\max_{d,a} \sum_{i \in V} s_v \tag{7a}$$

$$s.t. \sum_{i \in V} d_v \leq n_D \tag{7b}$$

$$a \in \min_{\tilde{a}} \sum_{i \in V} s_v \tag{7c}$$

$$s.t. \sum_{i \in v} \tilde{a}_v \leq n_A \tag{7d}$$

$$d_i \rightarrow \neg \tilde{a}_i \quad \forall i \in V \tag{7e}$$

$$s_i^0 = \neg \tilde{a}_i \quad \forall i \in V \tag{7f}$$

$$s_i^{t+1} = d_i \vee \left(\forall i' \in \mathcal{N}_G(i),\ s_{i'}^t\right) \forall t \in \{0...r-1\} \tag{7g}$$

$$s_i = s_i^r \tag{7h}$$

The program on the top level (defender), maximizes the total number of safe nodes by choosing at-most n_D nodes (7b) to defend subject to the fact that the attacker's program chooses at-most n_A nodes to attack (7d). Attacker can not attack a defended node (7e). Initially, only attacked nodes are unsafe (7f) but depending upon the propagation radius r, the final safety status of all nodes are calculated (7g, 7h). The set $\mathcal{N}_G(v)$ in equation 7g denotes all the immediate neighbors of node v in the graph. It's easy to see that the attacker's program is an IPG parameterized in d. Let $\mathcal{A}(d)$ be the set of attacker strategies that satisfy the LOIS-1 optimality of the parameterized program, then the bilevel program can be reformulated as:

$$\max_{d,a} \sum_{i \in V} s_v \tag{8a}$$

$$s.t. \sum_{i \in V} d_v \leq n_D \tag{8b}$$

$$a \in \mathcal{A}(d) \tag{8c}$$

$$\tag{8d}$$

The resulting single-level IPG consists of boolean and integer variables with pseudo-boolean constraints. The objective can also be encoded as a weighted MaxSAT [28] constraint. Therefore, the entire program can be easily solved for using a commercially available MaxSAT solver.

Scalable Precise Computation of Shannon Entropy

Yong Lai
Key Laboratory of Symbolic Computation and Knowledge Engineering Ministry of Education, Jilin University, Changchun, China

Haolong Tong
College of Computer Science and Technology, Jilin University, Changchun, China

Zhenghang Xu
School of Computer Science and Information Technology, Northeast Normal University, Changchun, China

Minghao Yin
School of Computer Science and Information Technology, Northeast Normal University, Changchun, China

Abstract

Quantitative information flow analyses (QIF) are a class of techniques for measuring the amount of confidential information leaked by a program to its public outputs. Shannon entropy is an important method to quantify the amount of leakage in QIF. This paper focuses on the programs modeled in Boolean constraints and optimizes the two stages of the Shannon entropy computation to implement a scalable precise tool PSE. In the first stage, we design a knowledge compilation language called ADD[∧] that combines Algebraic Decision Diagrams and conjunctive decomposition. ADD[∧] avoids enumerating possible outputs of a program and supports tractable entropy computation. In the second stage, we optimize the model counting queries that are used to compute the probabilities of outputs. We compare PSE with the state-of-the-art probabilistic approximately correct tool EntropyEstimation, which was shown to significantly outperform the previous precise tools. The experimental results demonstrate that PSE solved 56 more benchmarks compared to EntropyEstimation in a total of 459. For 98% of the benchmarks that both PSE and EntropyEstimation solved, PSE is at least 10× as efficient as EntropyEstimation.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Knowledge Compilation, Algebraic Decision Diagrams, Quantitative Information Flow, Shannon Entropy

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.20

Supplementary Material *Software (Source Code)*: `https://github.com/laigroup/PSE`
archived at `swh:1:dir:827b579f6c2ef3a9a79de5d7aaf910102c8ac07c`

Funding This work was supported in part by Jilin Provincial Natural Science Foundation [20240101378JC], Jilin Provincial Education Department Research Project [JJKH20241286KJ], and the National Natural Science Foundation of China [U22A2098, 62172185, and 61976050].

Acknowledgements The authors thank the anonymous reviewers for their constructive feedback.

1 Introduction

Quantitative information flow (QIF) is an important approach to measuring the amount of information leaked about a secret by observing the running of a program [11, 16]. In QIF, we often quantify the leakage using entropy-theoretic notions, such as Shannon entropy [2, 5, 30, 33] or min-entropy [2, 29, 30, 33]. Roughly speaking, a program in QIF can be seen as a

Authors are listed alphabetically by last name. Corresponding author: Minghao Yin

© Yong Lai, Haolong Tong, Zhenghang Xu, and Minghao Yin;
licensed under Creative Commons License CC-BY 4.0

function from a set of secret inputs X to outputs Y observable to an attacker who may try to infer X based on the output Y. Boolean formulas are a basic representation to model programs [14, 15]. In this paper, we focus on precisely computing the Shannon entropy of a program expressed in Boolean formulas.

Let $\varphi(X, Y)$ be a (Boolean) formula that models the relationship between the input variable set X and the output variable set Y in a given program, such that for any assignment of X, at most one assignment of Y satisfies the formula $\varphi(X, Y)$. Let p represent a probability distribution defined over the set $\{false, true\}^Y$. For each assignment σ to Y, the probability is defined as $p_\sigma = \frac{|Sol(\varphi(Y \mapsto \sigma))|}{|Sol(\varphi)_{\downarrow X}|}$, where $Sol(\varphi(Y \mapsto \sigma))$ denotes the set of solutions of $\varphi(Y \mapsto \sigma)$ and $Sol(\varphi)_{\downarrow X}$ denotes the set of solutions of φ projected to X. The Shannon entropy of φ is $H(\varphi) = \sum_{\sigma \in 2^Y} -p_\sigma \log p_\sigma$. Then we can immediately obtain a measure of leaked information with the computed entropy and the assumption that X follows a uniform distribution [1] [19].

The workflow of existing precise methods for computing entropy can often be divided into two stages. In the first stage, we enumerate possible outputs, i.e., the satisfying assignments over Y, while in the second stage, we compute the probability of the current output based on the number of inputs mapped to the output [15]. The computation in the second stage often invokes model counting (#SAT), which refers to computing the number of solutions $Sol(\varphi)$ for a given formula φ. Due to the exponential number of possible outputs, the current precise methods are often difficult to scale to programs with a large size of Y. Therefore, researchers have increasingly focused on approximate estimation of Shannon entropy. We remark that Golia et al. [15] proposed the first Shannon entropy estimation tool, EntropyEstimation, which guarantees that the estimate lies within $(1 \pm \epsilon)$-factor of $H(\varphi)$ with confidence at least $1 - \delta$. EntropyEstimation employs uniform sampling to avoid generating all outputs, and indeed scales much better than the precise methods.

As previously discussed, existing methods for precisely computing Shannon entropy struggle to scale when applied to formulas with a large set of outputs. Theoretically, this requires performing up to $2^{|Y|}$ model counting queries. The primary contribution of this paper is to enhance the scalability of precise Shannon entropy computation by improving both stages of the computation process. For the first stage, we design a knowledge compilation language to guide the search and avoid exhaustive enumeration of possible outputs. This language augments Algebraic Decision Diagrams (ADDs), an influential representation, with conjunctive decomposition. For the second stage, instead of performing model counting queries individually, we leverage shared component caching across successive queries. Moreover, we exploit literal equivalence to pre-process the formula corresponding to a given program. By integrating these techniques, we develop a Precise Shannon Entropy tool PSE. We conducted an extensive experimental evaluation over a comprehensive set of benchmarks (459 in total) and compared PSE with the existing precise Shannon entropy computing methods and the current state-of-the-art Shannon entropy estimation tool, EntropyEstimation. Our experiments indicate that EntropyEstimation is able to solve 276 instances, whereas PSE surpasses this by solving an additional 56 instances. Among the benchmarks that were solved by both PSE and EntropyEstimation, PSE is at least 10× as efficient as EntropyEstimation in 98% of these benchmarks.

The remainder of this paper is organized as follows. Section 2 introduces the notation and provides essential background information. Section 3 introduces Algebraic Decision Diagrams with conjunctive decomposition (ADD[∧]). Section 4 discusses the application of ADD[∧] to QIF and introduces our precise entropy tool, PSE. Section 5 details the experimental setup, results, and analysis. Section 6 reviews related work. Finally, Section 7 concludes the paper.

[1] If X does not follow a uniform distribution, techniques exist for reducing the analysis to a uniform case [1].

2 Notations and Background

In this paper, we focus on the programs modeled by (Boolean) formulas. In the formulas discussed, the symbols x and y denote variables, and literal l refers to either the variable x or its negation $\neg x$, where $var(l)$ represents the variable underlying the literal l. A formula φ is constructed from the constants *true*, *false* and variables using negation operator $\neg$, conjunction operator $\wedge$, disjunction operator $\vee$, implication operator $\rightarrow$, and equality operator $\leftrightarrow$, where $Vars(\varphi)$ denotes the set of variables appearing in φ. A clause C (resp. term T) is a set of literals representing their disjunction (resp. conjunction). A formula in conjunctive normal form (CNF) is a set of clauses representing their conjunction. Given a formula φ, a variable x, and a constant b, the substitution $\varphi[x \mapsto b]$ refers to the transformed formula obtained by substituting the occurrence of x with b throughout φ.

An assignment σ over variable set V is a mapping from V to $\{false, true\}$. The set of all assignments over V is denoted by 2^V. Given a subset $V' \subseteq V$, $\sigma_{\downarrow V'} = \{x \mapsto b \in \sigma \mid x \in V'\}$. Given a formula φ, an assignment over $Vars(\varphi)$ satisfies φ ($\sigma \models \varphi$) if the substitution $\varphi[\sigma]$ is equivalent to *true*. Given an assignment σ, if all variables are assigned a value in $\{false, true\}$, then σ is referred to as a complete assignment. Otherwise it is a partial assignment. A satisfying complete assignment is also called solution or model. We use $Sol(\varphi)$ to the set of solutions of φ, and model counting is the problem of computing $|Sol(\varphi)|$. Given two formulas φ and ψ over V, $\varphi \models \psi$ iff $Sol(\varphi \wedge \neg\psi) = \emptyset$.

2.1 Circuit formula and its Shannon entropy

Given a formula $\varphi(X, Y)$ to represent the relationship between input variables X and output variables Y, if $\sigma_{\downarrow X} = \sigma'_{\downarrow X}$ implies $\sigma = \sigma'$ for each $\sigma, \sigma' \in Sol(\varphi)$, then φ is referred to as a circuit formula. It is standard in the security community to employ circuit formulas to model programs in QIF [15].

▶ **Example 1.** The following formula is a circuit formula with input variables $X = \{x_1, \ldots, x_{2n}\}$ and output variables $Y = \{y_1, \ldots, y_{2n}\}$: $\varphi_n^{sep} = \bigwedge_{i=1}^{n}(x_i \wedge x_{n+i} \rightarrow y_i \wedge y_{n+i}) \wedge (\neg x_i \vee \neg x_{n+i} \rightarrow \neg y_i \wedge \neg y_{n+i})$.

In the computation of Shannon entropy, we focus on the probability distribution of outputs. Let p denote a probability distribution defined over the set $\{false, true\}^Y$. For each assignment σ to Y, i.e., $\sigma : Y \mapsto \{false, true\}$, its weight and probability is defined as $\omega_\sigma = |Sol(\varphi(Y \mapsto \sigma))|$ and $p_\sigma = \frac{|Sol(\varphi(Y \mapsto \sigma))|}{|Sol(\varphi)_{\downarrow X}|}$, respectively, where $Sol(\varphi(Y \mapsto \sigma))$ denotes the set of solutions of $\varphi(Y \mapsto \sigma)$ and $Sol(\varphi)_{\downarrow X}$ denotes the set of solutions of φ projected to X. Since φ is a circuit formula, it is easy to prove that $|Sol(\varphi)_{\downarrow X}| = |Sol(\varphi)|$. Then, the entropy of φ is $H(\varphi) = \sum_{\sigma \in 2^Y} -p_\sigma \log p_\sigma$. Following the convention in QIF [33], we use base 2 for log, though the base can be chosen freely.

2.2 Knowledge compilation

Knowledge compilation is the approach of compiling CNF formulas into a form to support tractable reasoning tasks such as satisfiability check, equivalence check, and model counting [10]. Ordered binary decision diagram (OBDD) [4] is one of the most influential knowledge compilation forms, which supports many tractable reasoning tasks. Each OBDD is a rooted directed acyclic graph (DAG) defined over a linear ordering of variables $\prec$. Each internal node v is called decision node and has two outgoing edges, referred to as the low child $lo(u)$ and the high child $hi(u)$, which are typically represented by dashed and solid lines,

respectively. Every node u is labeled with a symbol $sym(u)$. If u is a terminal node, then $sym(u) = \bot$ or $\top$, representing the Boolean constants *false* and *true*, respectively. Otherwise, $sym(u)$ denotes a variable and u represents $(\neg sym(u) \wedge \psi) \vee (sym(u) \wedge \psi')$, where ψ and ψ' are the formulas represented by $lo(u)$ and $hi(u)$, respectively. Each decision node v and its parent u have $sym(u) \prec sym(v)$. OBDD[$\wedge$] [24] is an extended form of OBDD with better space efficiency. It augments OBDD with conjunctive decomposition nodes. Each conjunctive decomposition node u has a set of children $Ch(u)$ representing formulas without shared variables, and u represents a conjunction of the formulas represented by its children. OBDD[$\wedge$] also supports a set of tractable reasoning tasks, including model counting and equivalence check.

Both OBDD and OBDD[$\wedge$] can only represent Boolean functions. An Algebraic Decision Diagram (ADD) [3] is an extension of OBDD to represent algebraic functions. ADD is a compact representation of a real-valued function as a directed acyclic graph. While OBDD has two terminal nodes representing *false* and *true*, ADD includes multiple terminal nodes, each assigned a real value. The order in which decision node labels appear in all paths from the root to the terminal nodes of the ADD also align with a given ordering of variables $\prec$. Given an assignment σ with each variable in $\prec$, we can obtain a path in a top-down way as follows: for a decision node with x, we pick low child if $\sigma(x) = false$, and high child otherwise. ADD maps σ to the value on the terminate node of the path. The original design motivation for ADD was to solve matrix multiplication, shortest path algorithms, and direct methods for numerical linear algebra [3]. In subsequent research, ADD has also been used for stochastic model checking [21], stochastic programming [17], and weighted model counting [12, 26].

3 ADD[$\wedge$]: A New Tractable Representation

In order to compute the Shannon entropy of a circuit formula $\varphi(X, Y)$, we need to use the probability distribution over the outputs. Algebraic Decision Diagrams (ADDs) are an influential compact probability representation that can be exponentially smaller than the explicit representation. Macii and Poncino [28] showed that ADD supports efficient exact computation of entropy. However, we observed in the experiments that the sizes of ADDs often exponentially explode with large circuit formulas. We draw inspiration from a Boolean representation known as the Ordered Binary Decision Diagram with conjunctive decomposition (OBDD[$\wedge$]) [24], which reduces its size through recursive component decomposition and divide-and-conquer strategies. This approach enables the representation to be exponentially smaller than the original OBDD. Accordingly, we propose a probabilistic representation called Algebraic Decision Diagrams with conjunctive decomposition (ADD[$\wedge$]) and demonstrate it supports tractable entropy computation. ADD[$\wedge$] is a general form of ADD and is defined as follows:

▶ **Definition 2.** *An ADD[$\wedge$] is a rooted DAG, where each node u is labeled with a symbol $sym(u)$. If u is a terminal node, $sym(u)$ is a non-negative real weight, also denoted by $\omega(u)$; otherwise, $sym(u)$ is a variable (called* decision *node) or operator $\wedge$ (called* decomposition *node). The children of a decision node u are referred to as the* low *child $lo(u)$ and the* high *child $hi(u)$, and connected by dashed lines and solid lines, respectively, corresponding to the cases where $var(u)$ is assigned the value of false and true. For a decomposition node, its sub-graphs do not share any variables. An ADD[$\wedge$] is imposed with a linear ordering $\prec$ of variables such that given a node u and its non-terminal child v, $var(u) \prec var(v)$.*

Hereafter, we denote the set of variables that appear in the graph rooted at u as $Vars(u)$ and the set of child nodes of u as $Ch(u)$. We now turn to show how an ADD[$\wedge$] defines a probability distribution:

▶ **Definition 3.** *Let u be an ADD[$\wedge$] node over a set of variables Y and let σ be an assignment over Y. The weight of σ is defined as follows:*

$$\omega(\sigma, u) = \begin{cases} \omega(u) & \text{terminal} \\ \prod_{v \in Ch(u)} \omega(\sigma, v) & \text{decomposition} \\ \omega(\sigma, lo(u)) & \text{decision and } \sigma \models \neg var(u) \\ \omega(\sigma, hi(u)) & \text{decision and } \sigma \models var(u) \end{cases}$$

The weight of an non-terminal ADD[$\wedge$] rooted at u is denoted by $\omega(u)$ and defined as $\sum_{\sigma \in 2^{Vars(u)}} \omega(\sigma, u)$. For nodes with a non-zero weight, the probability of σ over u is defined as $p(\sigma, u) = \frac{\omega(\sigma,u)}{\omega(u)}$.

Figure 1 depicts an ADD[$\wedge$] representing the probability distribution of φ_n^{sep} in Example 1 over its outputs with respect to $y_1 \prec y_2 \prec \cdots \prec y_{2n}$. The reader can verify that each equivalent ADD with respect to $\prec$ has an exponential number of nodes. In the field of knowledge compilation [10, 13], the concept of succinctness is often used to describe the space efficiency of a representation. Based on the following observations, we can conclude that ADD[$\wedge$] is strictly more succinct than ADD. First, OBDD and OBDD[$\wedge$] are subsets of ADD and ADD[$\wedge$], respectively. Second, OBDD[$\wedge$] is strictly more succinct than OBDD [24]. Finally, each OBDD[$\wedge$] cannot be transformed into a non-OBDD ADD.

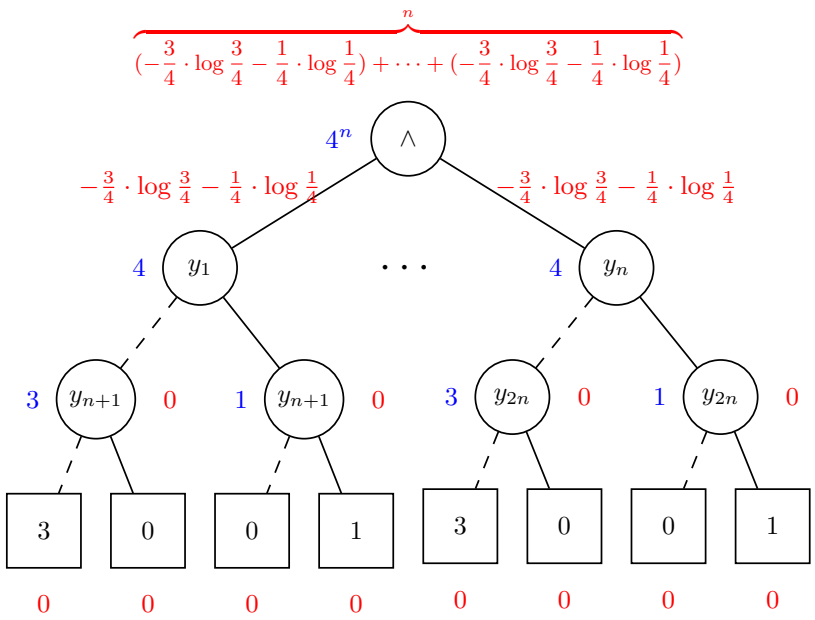


Figure 1 An ADD[$\wedge$] representing the probability distribution of φ_n^{sep} in Example 1 over its outputs. According to Proposition 4, the computed weight for each node is marked in blue font. According to Proposition 5, the computed entropy for each node is marked in red font.

3.1 Tractable Computation of Weight and Entropy

The computation of Shannon entropy for an ADD[$\wedge$] relies on its weight. We first demonstrate that, for an ADD[$\wedge$] node u, its weight $\omega(u)$ can be computed in polynomial time.

▶ **Proposition 4.** *Given a non-terminal node u in ADD[$\wedge$], its weight $\omega(u)$ can be recursively computed as follows in polynomial time:*

$$\omega(u) = \begin{cases} \prod_{v \in Ch(u)} \omega(v) & \textit{decomposition} \\ 2^{n_0} \cdot \omega(lo(u)) + 2^{n_1} \cdot \omega(hi(u)) & \textit{decision} \end{cases}$$

where $n_0 = |Vars(u)| - |Vars(lo(u))| - 1$ *and* $n_1 = |Vars(u)| - |Vars(hi(u))| - 1$.

Proof. The time complexity is immediate by using dynamic programming. We prove the equation can compute the weight correctly by induction on the number of variables of the ADD[$\wedge$] rooted at u. It is obvious that the weight of a terminal node is the real value labeled. For the case of the $\wedge$ node, since the variables of the child nodes are all disjoint, it can be easily seen from Definition 3. Next, we will prove the case of the decision node. Assume that when $|Vars(u)| \leq n$, this proposition holds. For the case where $|Vars(u)| = n+1$, we use Y_0 and Y_1 to denote $Vars(lo(u))$ and $Vars(hi(u))$, and we have $|Y_0| \leq n$ and $|Y_1| \leq n$. Thus, $\omega(lo(u))$ and $\omega(hi(u))$ can be computed correctly. According to Definition 3, $w(u) = \sum_{\sigma \in 2^{Vars(u)}} \omega(\sigma, u)$. The assignments over $Vars(u)$ can be divided into two categories:

- The assignment $\sigma \models \neg var(u)$: It is obvious that $\omega(\sigma, u) = \omega(\sigma_{\downarrow Y_0}, lo(u))$. Each assignment over Y_0 can be extended to exactly 2^{n_0} different assignments over $Vars(u)$ in this category. Thus, we have the following equation:

$$\sum_{\sigma \in 2^{Vars(u)} \wedge \sigma \models \neg var(u)} \omega(\sigma, u) = 2^{n_0} \cdot \omega(lo(u)).$$

- The assignment $\sigma \models var(u)$: This case is similar to the above case.

To sum up, we can obtain that $\omega(u) = 2^{n_0} \cdot \omega(lo(u)) + 2^{n_1} \cdot \omega(hi(u))$. ◀

We now present how ADD[$\wedge$] computes Shannon entropy in polynomial time.

▶ **Proposition 5.** *Given an ADD[$\wedge$] rooted at u, if $\omega(u) = 0$, we define its entropy $H(u)$ as 0, and otherwise its entropy can be recursively computed in polynomial time as follows:*

$$H(u) = \begin{cases} 0 & \textit{terminal} \\ \sum_{v \in Ch(u)} H(v) & \textit{decomposition} \\ p_0 \cdot (H(lo(u)) + n_0 - \log p_0) + p_1 \cdot (H(hi(u)) + n_1 - \log p_1) & \textit{decision} \end{cases}$$

where $n_0 = |Vars(u)| - |Vars(lo(u))| - 1$, $n_1 = |Vars(u)| - |Vars(hi(u))| - 1$, $p_0 = \frac{2^{n_0} \cdot \omega(lo(u))}{\omega(u)}$, *and* $p_1 = \frac{2^{n_1} \cdot \omega(hi(u))}{\omega(u)}$.

Proof. According to Proposition 4, $\omega(u)$ can be computed in polynomial time, and therefore the time complexity in this proposition is obvious. Next we prove the correctness of the computation method. The case of terminal nodes is obviously correct. The case of decomposition follows directly from the additivity property of entropy. Next, we show the correctness of the case of decision.

Let $H_0(u)$ be $-\sum_{\sigma \models \neg sym(u)} p(u, \sigma) \log p(u, \sigma)$ and $H_1(u)$ be $-\sum_{\sigma \models sym(u)} p(u, \sigma) \log p(u, \sigma)$. Similar to proposition 4, we can obtain $H(u) = 2^{n_0} \cdot H_0(u) + 2^{n_1} \cdot H_1(u)$. The assignments over $Vars(u)$ can be divided into two categories:

- The assignment $\sigma \models \neg var(u)$: According to definition 3, the probability $p(\sigma, \omega)$ satisfies $p(\sigma, \omega) = \frac{\omega(\sigma, lo(u))}{\omega(u)}$. Given $p_0 = \frac{2^{n_0} \cdot \omega(lo(u))}{\omega(u)}$, it follows that $\frac{\omega(lo(u))}{\omega(u)} = \frac{p_0}{2^{n_0}}$. Substituting this into the expression for $p(\sigma, u)$, we derive $p(\sigma, u) = \frac{\omega(\sigma, lo(u))}{\omega(lo(u))} \cdot p_0 \cdot 2^{-n_0} = p(\sigma, lo(u)) \cdot p_0 \cdot 2^{-n_0}$. $H_0(u)$ then expands as $H_0(u) = -\sum_{\sigma \models \neg sym(u)} p(\sigma, u) \log p(\sigma, u) =$

$-\sum_{\sigma\models\neg sym(u)} p(\sigma, lo(u)) \cdot p_0 \cdot 2^{-n_0} \cdot (\log p(\sigma, lo(u)) + \log p_0 - n_0) = p_0 \cdot 2^{-n_0} \cdot [-\log p_0 \cdot \sum_{\sigma\models\neg sym(u)} p(\sigma, lo(u)) + n_0 \cdot \sum_{\sigma\models\neg sym(u)} p(\sigma, lo(u)) - \sum_{\sigma\models\neg sym(u)} p(\sigma, lo(u)) \cdot \log p(\sigma, lo(u))]$. Noting that $\sum_{\sigma\models\neg sym(u)} p(\sigma, lo(u)) = 1, -\sum_{\sigma\models\neg sym(u)} p(\sigma, lo(u)) \cdot \log p(\sigma, lo(u)) = H(lo(u))$, we simplify $H_0(u) = p_0 \cdot 2^{-n_0} \cdot (-\log p_0 + n_0 + H(lo(u)))$.

- The assignment $\sigma \models var(u)$: This case is similar to the above case. It is easy to obtain $H_1(u) = p_1 \cdot 2^{-n_1} \cdot (-\log p_1 + n_1 + H(hi(u)))$.

To sum up, we can obtain that $H(u) = p_0 \cdot (H(lo(u)) + n_0 - \log p_0) + p_1 \cdot (H(hi(u)) + n_1 - \log p_1)$ ◀

We conclude this section by explaining why ordering is used in the design of ADD[∧]. In fact, Propositions 4–5 remain valid even when we use only the more general read-once property, where each variable appears at most once along any path from the root of an ADD[∧] to a terminal node. First, our experimental results indicate that the linear ordering determined by the minfill algorithm in our tool PSE outperforms the dynamic orderings employed in the state-of-the-art model counters, where the former imposes the orderedness and the latter imposes the read-once property. Second, ADD[∧] can provide tractable equivalence checking between probability distributions beyond this study.

4 PSE: Scalable Precise Entropy Computation

In this section, we introduce our tool PSE, designed to compute the Shannon entropy of a given circuit CNF formula with respect to its output variables. PSE, as presented in Algorithm 1, takes as input a CNF formula φ, an input set X, and an output set Y, and returns the Shannon entropy $H(\varphi)$ of the formula. Like other tools for computing Shannon entropy, PSE follows a two-stage process: the Y-stage (corresponding to outputs) and the X-stage (corresponding to inputs). In the X-stage (lines 3–4), we perform multiple optimized model counting operations on sub-formulas over variables in X, where the leaves of ADD[∧] are implicitly generated. The optimization technique is discussed in Section 4.1. In the Y-stage (the remaining lines), we conduct a search within the ADD[∧] framework to precisely compute the Shannon entropy, where the internal nodes of ADD[∧] are implicitly generated. The following observation states the input of each recursive call is still a circuit formula and the two input formulas of a call corresponding to a decision node in ADD[∧] have the same output variables.

▶ **Observation 6.** *Given a circuit formula $\varphi(X, Y)$ and a partial assignment σ without any input variables, we have the following properties:*

- *$\varphi[\sigma](X, Y \setminus Vars(\sigma))$ is a circuit formula;*
- *Each $\psi_i(X, Y \cap Vars(\psi_i))$ is a circuit formula if $\varphi = \bigwedge_{i=1}^{m} \psi_i$ and for $1 \le i \ne j \le m$, $Vars(\psi_i) \cap Vars(\psi_j) = \emptyset$;*
- *If $\varphi[\sigma] \equiv true$, σ contains each output variable.*

Proof. The first two properties obviously hold when φ is unsatisfiable. Thereby, we assume φ is satisfiable. For the first property, let φ' be $\varphi[\sigma](X, Y \setminus Vars(\sigma))$. For each $\sigma', \sigma'' \in Sol(\varphi)$, $\sigma'_{\downarrow X} = \sigma''_{\downarrow X}$ implies $\sigma' = \sigma''$. $Sol(\varphi')$ can be seen as a subset of $Sol(\varphi)$. Consequently, for each $\sigma', \sigma'' \in Sol(\varphi')$, $\sigma'_{\downarrow X} = \sigma''_{\downarrow X}$ still implies $\sigma' = \sigma''$, concluding that φ' is also a circuit formula.

For the second property, each solution of ψ_i can be obtain from a solution of φ. Let σ', σ'' be two solutions of φ. We only need to prove that $\sigma'_{\downarrow X\cap Vars(\psi_i)} = \sigma''_{\downarrow X\cap Vars(\psi_i)}$ implies $\sigma'_{\downarrow (X\cup Y)\cap Vars(\psi_i)} = \sigma''_{\downarrow (X\cup Y)\cap Vars(\psi_i)}$. We construct another solution of φ, $\sigma''' = \sigma''_{\downarrow (X\cup Y)\cap Vars(\psi_i)} \cup \sigma'_{\downarrow (X\cup Y)\setminus Vars(\psi_i)}$. Then $\sigma' = \sigma'''$, which implies $\sigma'_{\downarrow (X\cup Y)\cap Vars(\psi_i)} = \sigma'''_{\downarrow (X\cup Y)\cap Vars(\psi_i)} = \sigma''_{\downarrow (X\cup Y)\cap Vars(\psi_i)}$.

We prove the last property by contradiction. Suppose that $\varphi[\sigma] \equiv \mathit{true}$, and σ is a partial assignment with only one free variable $y \in Y$. Then the value of y can take either *false* or *true*. That is, $\sigma \cup \{y = \mathit{false}\}$ and $\sigma \cup \{y = \mathit{true}\}$ are solutions of φ, which contradicts the definition of circuit formula. ◀

Algorithm 1 PSE(φ,X,Y).

Input: A circuit CNF formula φ with input variables X and output variables Y
Output: the entropy of φ

1 **if** $\mathit{Cache}_H(\varphi) \neq \mathit{nil}$ **then return** $\mathit{Cache}_H(\varphi)$
2 **if** $Y = \emptyset$ **then**
3 $\quad \mathit{Cache}_\#(\varphi) \leftarrow$ `CountModels`(φ)
4 $\quad$ **return** $\mathit{Cache}_H(\varphi) \leftarrow 0$
5 $\Psi =$ `Decompose`(φ)
6 **if** $|\Psi| > 1$ **then**
7 $\quad \mathit{Cache}_H(\varphi) \leftarrow \sum_{\psi \in \Psi}$ **PSE**$(\psi, X, Y \cap \mathit{Vars}(\psi_i))$
8 $\quad \mathit{Cache}_\#(\varphi) \leftarrow \prod_{\psi \in \Psi} \mathit{Cache}_\#(\psi)$
9 $\quad$ **return** $\mathit{Cache}_H(\varphi)$
10 $y \leftarrow$ `PickGoodVar`(Y)
11 $\varphi_0 \leftarrow \varphi[y \mapsto \mathit{false}]$; $\varphi_1 \leftarrow \varphi[y \mapsto \mathit{true}]$
12 $H_0 \leftarrow$ **PSE**$(\varphi_0, X, Y \backslash \{y\})$
13 $H_1 \leftarrow$ **PSE**$(\varphi_1, X, Y \backslash \{y\})$
14 $\mathit{Cache}_\#(\varphi) \leftarrow \mathit{Cache}_\#(\varphi_0) + \mathit{Cache}_\#(\varphi_1)$
15 $p_0 = \frac{\mathit{Cache}_\#(\varphi_0)}{\mathit{Cache}_\#(\varphi)}$; $p_1 = \frac{\mathit{Cache}_\#(\varphi_1)}{\mathit{Cache}_\#(\varphi)}$
16 $H \leftarrow p_0 \cdot (H_0 - \log p_0) + p_1 \cdot (H_1 - \log p_1)$
17 **return** $\mathit{Cache}_H(\varphi) \leftarrow H$

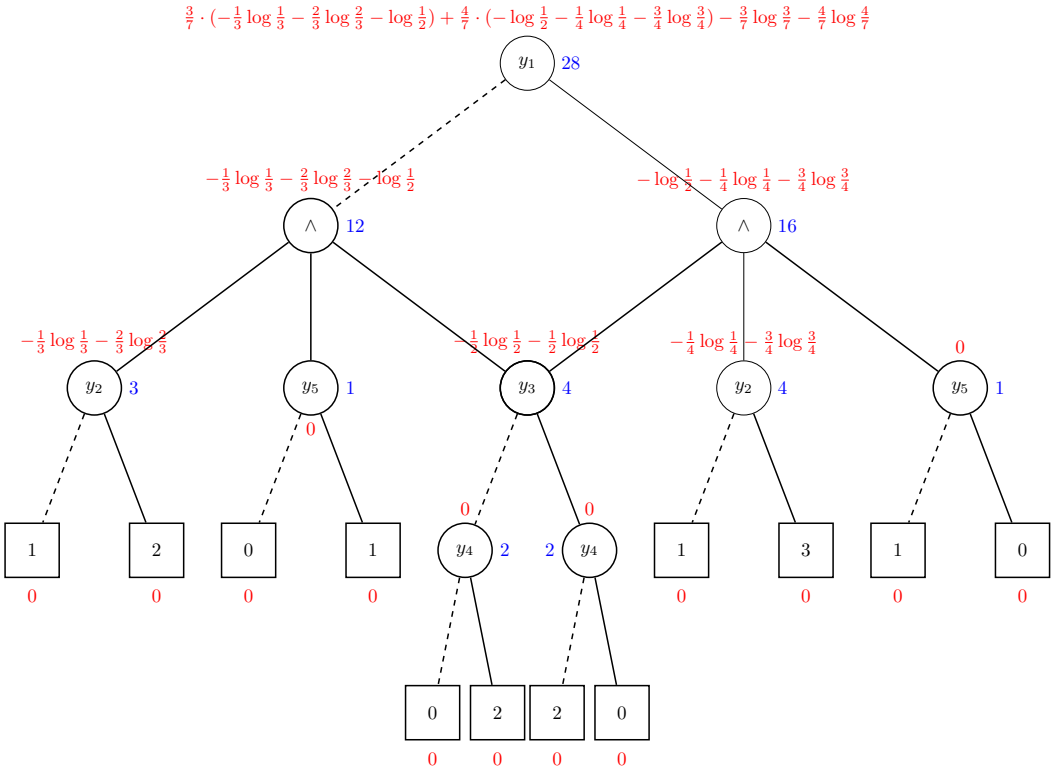
In line 1, if the formula φ is cached, its corresponding entropy is returned. If the current set Y is empty (in line 2), this indicates that a satisfiable assignment has been found under the restriction of the output set Y. We do not explicitly handle the case where φ evaluates to *true*, as this naturally implies that Y is empty, as indicated by Observation 6. Consequently, the scenario in which the set Y is empty inherently encompasses the case where φ evaluates to *true*. Lines 3–4 perform model counting on the residual formula and compute its entropy H, corresponding to the terminal case of Proposition 5. We invoke the `Decompose` function in line 5 to determine whether the formula φ can be decomposed into multiple components. In lines 6–9, if φ can be decomposed into multiple sub-components, we compute the model count and entropy of each component ψ, and subsequently derive the entropy of the formula φ. In this case, computing the model count and computing the entropy correspond respectively to the $\wedge$ cases in Propositions 4 and 5. When there is only one component, we select a variable from Y in line 10. The `PickGoodVar` function operates as a heuristic algorithm designed to select a variable from set Y, with the selection criteria determined by the specific heuristic employed. Moving forward, line 11 generates the residual formulas φ_0 and φ_1, corresponding to assigning the variable y to *false* and *true*, respectively. Subsequently, lines 12 and 13 recursively compute the entropy H for each derived formula. Since φ is a circuit formula, all residual formulas generated in the recursive process after making decisions on variables in Y remain circuit formulas. It follows from Observation 6 that when computing the Shannon entropy of the circuit formula, $n_0 = n_1 = 0$. The model count of φ is cached in line 14,

corresponding to the decision node case in Proposition 4. Finally, in lines 15–16, we compute the entropy of φ (corresponding to the third case in Proposition 5), store it in the cache, and return it as the result in line 17.

▶ **Example 7.** Consider the following circuit CNF formula with input variables $X = \{x_1, x_2, x_3, x_4, x_5\}$ and output variables $Y = \{y_1, y_2, y_3, y_4, y_5\}$:

$$\begin{aligned}\varphi(X,Y) =&(x_2 \vee x_3 \vee y_3) \wedge (\neg y_3 \vee \neg y_4) \wedge (x_2 \vee y_3) \wedge (\neg x_2 \vee y_4) \wedge (\neg x_1 \vee \neg y_1) \wedge (x_1 \vee y_1) \wedge \\ &(\neg x_4 \vee x_5 \vee y_2) \wedge (x_4 \vee \neg x_5 \vee y_2) \wedge (\neg x_4 \vee \neg x_5 \vee \neg y_2) \wedge (x_4 \vee x_5 \vee y_2) \wedge \\ &(\neg y_1 \vee \neg y_5) \wedge (y_1 \vee y_5) \wedge (y_1 \vee x_4 \vee x_5) \wedge (y_1 \vee y_3 \vee y_4)\end{aligned}$$

Figure 2 illustrates the execution trace of PSE taking in φ with the variable ordering $y_1 \prec y_2 \prec y_3 \prec y_4 \prec y_5$, which is an implicit ADD[$\wedge$]. If we do not perform decomposition in line 5, the search trace is depicted in Figure 3, an ADD structure. It is evident that ADD[$\wedge$] and ADD yield consistent results, both in terms of Shannon entropy computation and model counting. After merging identical terminal nodes, the ADD[$\wedge$] contains 14 nodes, which is fewer than the 24 nodes in the ADD. A comparison between Figure 2 and Figure 3 demonstrates the succinctness of the ADD[$\wedge$] structure.


$\frac{3}{7}\cdot(-\frac{1}{3}\log\frac{1}{3}-\frac{2}{3}\log\frac{2}{3}-\log\frac{1}{2})+\frac{4}{7}\cdot(-\log\frac{1}{2}-\frac{1}{4}\log\frac{1}{4}-\frac{3}{4}\log\frac{3}{4})-\frac{3}{7}\log\frac{3}{7}-\frac{4}{7}\log\frac{4}{7}$
y_1
28
$-\frac{1}{3}\log\frac{1}{3}-\frac{2}{3}\log\frac{2}{3}-\log\frac{1}{2}$
$-\log\frac{1}{2}-\frac{1}{4}\log\frac{1}{4}-\frac{3}{4}\log\frac{3}{4}$
$\wedge$
12
$\wedge$
16
$-\frac{1}{3}\log\frac{1}{3}-\frac{2}{3}\log\frac{2}{3}$
$-\frac{1}{2}\log\frac{1}{2}-\frac{1}{2}\log\frac{1}{2}$
$-\frac{1}{4}\log\frac{1}{4}-\frac{3}{4}\log\frac{3}{4}$
0
y_2
3
y_5
1
y_3
4
y_2
4
y_5
1
0
0
0
1
2
0
1
y_4
2
2
y_4
1
3
1
0
0
0

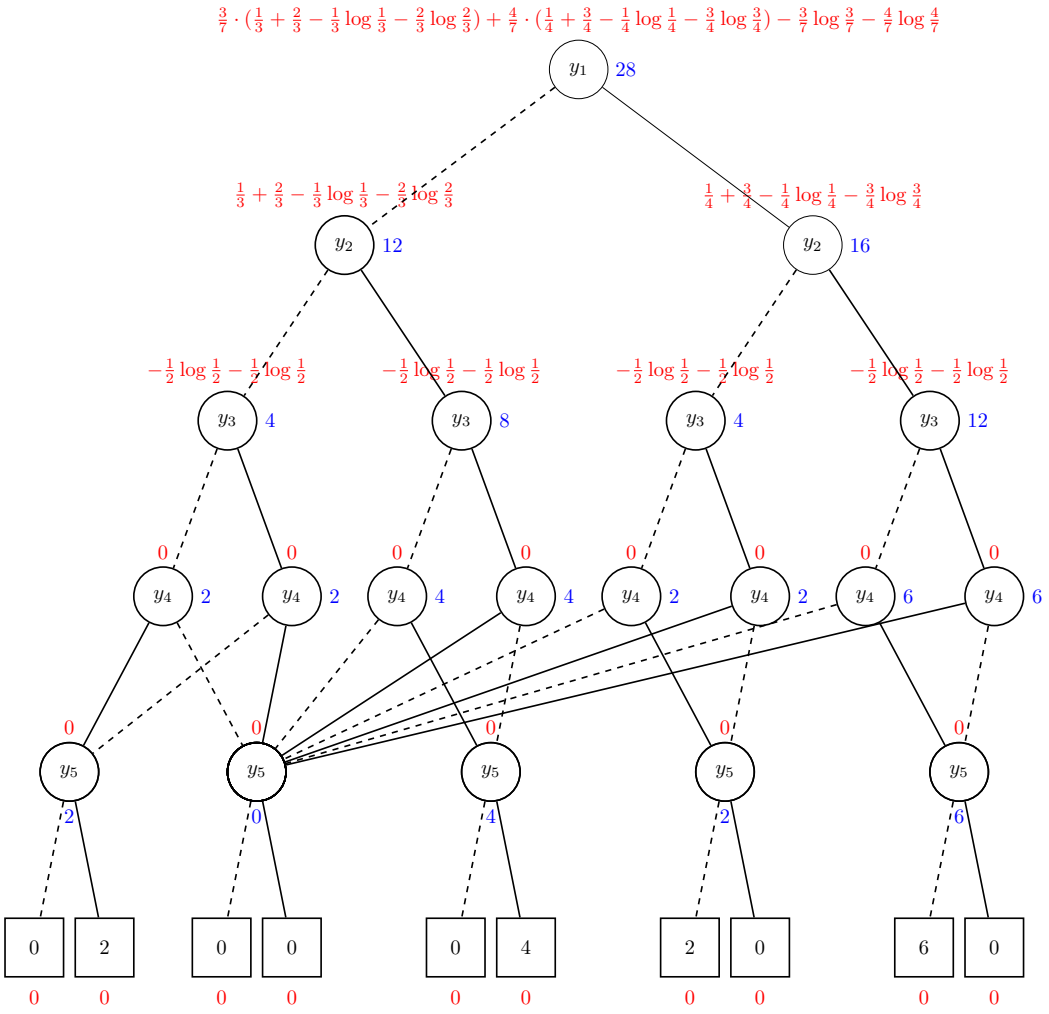


Figure 3 An ADD structure, constructed in Example 7, follows the variable order of $y_1 \prec y_2 \prec y_3 \prec y_4 \prec y_5$. According to Proposition 4, weight is marked in blue font, and according to Proposition 5, entropy is marked in red font.

4.1 Implementation

We now discuss the implementation details that are crucial for the runtime efficiency of PSE. Specifically, leveraging the tight interplay between entropy computation and model counting, our methodology integrates a variety of state-of-the-art techniques in model counting.

In the X-stage of algorithm 1, we have the option to employ various methodologies for the model counting query denoted by `CountModels` in line 3. The first method involves individually employing state-of-the-art model counters, such as SharpSAT-TD [20], Ganak [32], and ExactMC [25]. The second method, known as ConditionedCounting, requires the preliminary construction of a representation for the original formula φ to support linear model counting. The knowledge compilation languages that can be used for this method include d-DNNF [8], OBDD[$\wedge$] [24], and SDD [7]. Upon reaching line 3, the algorithm executes conditioned model counting, utilizing the compiled representation of the formula and incorporating the partial assignment derived from the ancestor calls. The last method, SharedCounting, also relies on exact model counters but, unlike the first method, it shares the component cache across all

model counting queries using a strategy called XCache. To distinguish it from the caching approach used in the X-stage, the caching method in the Y-stage is referred to as YCache. Our experimental observations indicate that the SharedCounting method is the most effective within the PSE framework.

Conjunctive Decomposition. We employed dynamic component decomposition (well-known in model counting and knowledge compilation) to divide a formula into components, thereby enabling the dynamic programming calculation of their corresponding entropy, as stated in Proposition 5.

Variable Decision Heuristic. We implemented the current state-of-the-art model counting heuristics for picking variables from Y in the computation of Shannon entropy, including VSADS [31], minfill [9], the SharpSAT-TD heuristic [20], and DLCP [25]. Our experiments consistently demonstrate that the minfill heuristic exhibits the best performance. Therefore, we adopt the minfill heuristic as the default option for our subsequent experiments.

Pre-processing. We have enhanced our entropy tool, PSE, by incorporating an advanced pre-processing technique that capitalizes on literal equivalence in model counting. This idea is inspired by the work of Lai et al. [25] on capturing literal equivalence in model counting. Initially, we extract equivalent literals to simplify the formula. Subsequently, we restore the literals associated with the variables in set Y to prevent the entropy of the formula from becoming non-equivalent after substitution. This targeted restoration is sufficient to ensure the equivalence of entropy calculations. The new pre-processing method is called Pre in the following. This pre-processing approach is motivated by two primary considerations. Firstly, preprocessing based on literal equivalence can simplify the formula and enhance the efficiency of subsequent model counting. Secondly, and more crucially, it can reduce the treewidth of tree decomposition, which is highly beneficial for the variable heuristic method based on tree decomposition and contributes to improving the solving efficiency.

5 Experiments

We implemented a prototype of PSE in C++ and performed evaluations in order to understand its performance. We experimented on benchmarks from the same domains as the state-of-the-art Shannon entropy tool EntropyEstimation [15], that is, QIF benchmarks, plan recognition, bit-blasted versions of SMTLIB benchmarks, QBFEval competitions, program synthesis, and combinatorial circuits [27] [2]. EntropyEstimation reported results only for 96 successfully solved benchmarks (denoted Suite_1), which we found insufficient for scalability testing. To ensure a rigorous evaluation, we extended Suite_1 as follows:

- Suite_2 (399 benchmarks): Suite_1 is from the benchmarks that were used to test a well-known model counter called Ganak [3]; thereby, we added each circuit formula in the aforementioned domain but not in Suite_1 from the Ganak benchmarks.
- Suite_3 (459 benchmarks): Incorporated 60 additional combinatorial circuits [4] from [27] on the basis of Suite_2.

[2] The paper of EntropyEstimation [15] does not mention the domains of program synthesis and combinatorial circuits but actually presents benchmarks in these two domains.
[3] The benchmarks are available at `https://github.com/meelgroup/ganak`
[4] The additional benchmarks are available at `https://github.com/nianzelee/PhD-Dissertation`

All experiments were run on a computer with Intel(R) Core(TM) i9-10920X CPU @ 3.50GHz and 32GB RAM. Each instance was run on a single core with a timeout of 3000 seconds and 4GB memory, the same setup adopted in the evaluation of EntropyEstimation.

Through our experiments, we sought to answer the following research questions:

RQ1: How does the runtime performance of PSE compare to the state-of-the-art Shannon entropy tools with (probabilistic) accuracy guarantee?

RQ2: How do the utilized methods impact the runtime performance of PSE?

5.1 RQ1: Performance of PSE

Golia et al. [15] have already demonstrated that their probably approximately correct tool EntropyEstimation is significantly more efficient than the state-of-the-art precise Shannon entropy tools. The comparative experiments between PSE and the state-of-the-art precise tools are presented in the appendix. We remark that PSE significantly outperforms the precise baseline (the baseline was able to solve only 18 benchmarks, whereas PSE solved 332 benchmarks). This marked improvement is attributed to the linear entropy computation capability of ADD[$\wedge$] and the effectiveness of various strategies employed in PSE.

Table 1 presents the performance comparison between PSE and EntropyEstimation across the three benchmark suites. For Suite_1, EntropyEstimation solved two more instances than PSE, indicating a slight advantage. However, among the 94 instances that both solved, PSE demonstrated higher efficiency. Moreover, PSE achieved a lower PAR-2 [5] score than EntropyEstimation, suggesting that PSE holds an overall performance advantage. We remark that in the computation of the PAR-2 scores, we *did not perform additional penalization* for each successful run of EntropyEstimation as its output was very close to the true entropy. For Suite_2, PSE solved 44 more instances than EntropyEstimation and achieved a significantly lower PAR-2 score, further demonstrating its superior performance. For Suite_3, PSE solved 56 more instances than EntropyEstimation. Additionally, in terms of overall performance, PSE achieved a significantly lower PAR-2 score than EntropyEstimation, reinforcing its advantage.

Table 1 Detailed performance comparison of PSE and EntropyEstimation. Unique represents the number of instances that can only be solved by a specific tool. Fastest represents the number of instances that a tool solves with the shortest time.

Suite	Tool	Solved Instances			PAR-2 score
		Unique	Fastest	Total	
Suite_1	EntropyEstimation	**2**	0	**96**	149.63
	PSE (ours)	0	**94**	94	**126.04**
Suite_2	EntropyEstimation	2	0	268	2132.52
	PSE (ours)	**46**	**266**	**312**	**1314.59**
Suite_3	EntropyEstimation	2	0	276	2536.53
	PSE (ours)	**58**	**274**	**332**	**1666.11**

[5] The PAR-2 scoring scheme gives a penalized average runtime, assigning a runtime of two times the time limit for each benchmark that the tool fails to solve

Figure 4 demonstrates the detailed performance comparison between PSE and EntropyEstimation on Suite_3. More intuitively, among all the benchmarks that both PSE and EntropyEstimation are capable of solving, in 98% of those benchmarks, the efficiency of PSE surpasses that of EntropyEstimation by a margin of at least ten times. For all the benchmarks where PSE and EntropyEstimation did not timeout and took more than 0.1 seconds, the mean speedup is 506.62, which indicates an improvement of more than two orders of magnitude.

The aforementioned results clearly indicate that PSE outperforms EntropyEstimation in the majority of instances. This validates a positive answer to **RQ1**: PSE outperforms the state-of-the-art Shannon entropy tools with (probabilistic) accuracy guarantee. We remark that EntropyEstimation is an estimation tool for Shannon entropy with probabilistic approximately correct results [15]. PSE consistently performs better than a state-of-the-art entropy estimator across most instances, highlighting that our methods significantly enhance the scalability of precise Shannon entropy computation.

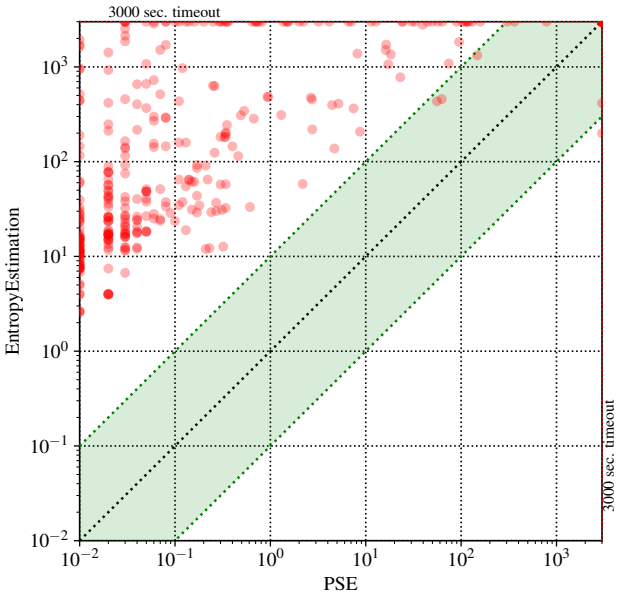


Figure 4 Scatter Plot of the running time Comparison between PSE and EntropyEstimation.

5.2 RQ2: Impact of algorithmic configurations

To better verify the effectiveness of the PSE methods and answer **RQ2**, we conducted a comparative study on all the utilized methods, including methods for the Y-stage: ConjunctiveDecomposition, YCache, Pre, variable decision heuristics (minfill, DLCP, SharpSAT-TD heuristic, VSADS), and methods for the X-stage: XCache and ConditionedCounting. In accordance with the principle of control variables, we conducted ablation experiments to evaluate the effectiveness of each method, ensuring that each experiment differed from the PSE tool by only one method. The cactus plot for the different methods is shown in Figure 5, where PSE represents our tool. PSE-wo-Decomposition indicates that the ConjunctiveDecomposition method is disabled in PSE, which means that its corresponding trace is ADD. PSE-wo-Pre means that Pre is turned off in PSE. PSE-ConditionedCounting indicates that PSE employed the ConditionedCounting method rather than SharedCounting in the X-stage. PSE-wo-XCache

indicates that the caching method is turned off in PSE in the X-stage. PSE-wo-YCache indicates that the caching method is turned off in PSE in the Y-stage. PSE-dynamic-SharpSAT-TD means that PSE replaces the minfill static variable order with the dynamic variable order: the variable decision-making heuristic method of SharpSAT-TD (all other configurations remain identical to PSE, with only the variable heuristic differing). Similarly, PSE-dynamic-DLCP and PSE-dynamic-VSADS respectively indicate the selection of dynamic heuristic DLCP and VSADS.

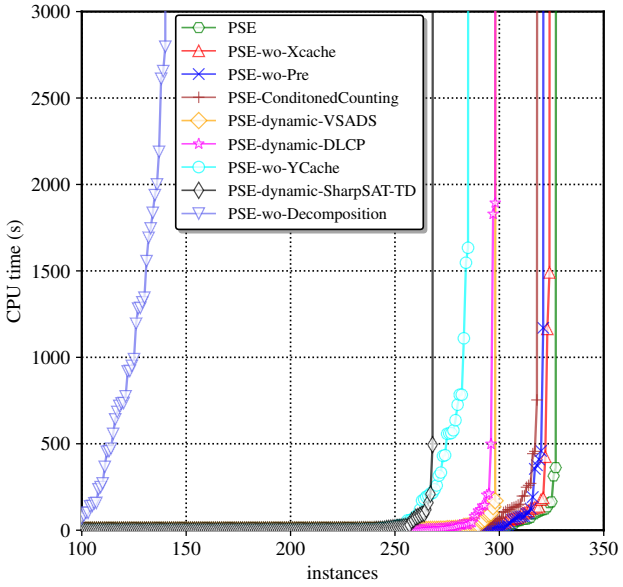


Figure 5 Cactus plot comparing different methods.

The experimental results highlight the significant effects of conjunctive decomposition. Caching also demonstrates significant benefits, consistent with findings from previous studies on knowledge compilation. It can also be clearly observed that Pre improves the efficiency of PSE. Among the heuristic strategies, it is evident that minfill performs the best. In the technique of the X-stage, the ConditionedCounting method performs better than SharedCounting without XCache, but not as well as the SharedCounting method. This comparative experiment indicates that the shared component caching is quite effective. The ConditionedCounting method's major advantage is its linear time complexity [24]. However, a notable drawback is the requirement to construct an OBDD[$\wedge$] (or other knowledge compilation languages such as d-DNNF, SDD, etc.) based on a static variable order, which can introduce considerable time overhead for more complex problems. Although the ConditionedCounting method is not the most effective, we believe it is still a promising and scalable method. In cases where an ADD[$\wedge$] can be efficiently constructed based on a static variable order, the ConditionedCounting method may be more effective than the SharedCounting method, especially when modeling counting in the X-stage is particularly challenging. Finally, PSE utilizes the SharedCounting strategy in the X-stage, and incorporates ConjunctiveDecomposition, YCache, Pre, and the minfill heuristic method in the Y-stage.

Finally, we analyze the effectiveness of algorithmic configurations across benchmarks. In terms of the number of solved instances, PSE either solves the most instances or ties with other configurations across all domains. Regarding the PAR-2 score, on QBF benchmarks, PSE-dynamic-VSADS has the lowest score, while in other domains, PSE has the lowest

scores. Among all the instances, there are a total of two instances [6] which PSE failed to solve within the specified time limit, but were solved by PSE-wo-Pre. In PSE, we use the minfill heuristic to construct a tree decomposition for a given circuit formula. We also observed that the resulting treewidth strongly correlates with compilation size – smaller treewidth in a benchmark typically leads to more efficient PSE execution.

6 Related work

Our work is based on the close relationship between QIF, model counting, and knowledge compilation. We introduce relevant work from three perspectives: (1) quantitative information flow analysis, (2) model counting, and (3) knowledge compilation.

Quantified information flow analysis. At present, the QIF method based on model counting encounters two significant challenges. The first challenge involves constructing the logical postcondition Π_{proc} for a program *proc* [34]. Although symbolic execution can achieve this, existing symbolic execution tools have limitations and are often challenging to extend to more complex programs, such as those involving symbolic pointers. The second challenge concerns model counting, a key focus of our research. For programs modeled by Boolean clause constraints, Shannon entropy can be computed via model counting queries, enabling the quantification of information leakage. Golia et al. [15] have made notable contributions to this field. They proposed the first efficient Shannon entropy estimation method with PAC guarantees, utilizing sampling and model counting. Their approach focuses on reducing the number of model counting queries by employing sampling techniques. Nevertheless, this method yields only an approximate estimation of entropy. Our research is motivated by the work of Golia et al., but diverges in its approach and optimization strategy. We enhance the existing model counting framework for precise Shannon entropy by reducing the number of model counting queries and concurrently improving the efficiency of model counting solutions. Inspired by Golia et al.'s work, our research differs in approach and optimization strategy. We improve the existing model counting framework for precise Shannon entropy by reducing the number of model counting queries and enhancing solution efficiency.

Model counting. Since the computation of entropy relies on model counting, we reviewed advanced techniques in this domain. The most effective methods for exact model counting include component decomposition, caching, variable decision heuristics, pre-processing, and so on. In our research, these methods can all be optimized and improved for application in Shannon entropy computation. The fundamental principle of disjoint component analysis involves partitioning the constraint graph into separate components that do not share variables. The core of ADD[$\wedge$] lies in leveraging component decomposition to enhance the efficiency of construction. We also utilized caching techniques in the process of computing entropy, and our experiments once again demonstrated the power of caching techniques. Extensive research has been conducted on variable decision heuristics for model counting, which are generally classified into static and dynamic heuristics. In static heuristics, the minfill [9] heuristic is notably effective, while in dynamic heuristics, VSADS [31], DLCP [25], and SharpSAT-TD heuristic [20] have emerged as the most significant in recent years. Lagniez et al. [22] offer a comprehensive review of preprocessing techniques in model counting.

[6] The two instances are in bit-blasted versions of SMTLIB benchmarks with names blasted_case_0_ptb_1 and blasted_TR_b12_1_linear.

Knowledge compilation. The motivation for knowledge compilation lies in transforming the original representation into a target language to enable efficient solving of inference tasks. Darwiche et al. first proposed a compiler called c2d [8] to convert the given CNF formula into Decision-DNNF. Lai et al. proposed two extended forms of OBDD: Ordered Binary Decision Diagram with Implied Literals (OBDD-L [23]), which is developed by extracting implied literals recursively; OBDD[∧] [24], which is proposed by integrating conjunctive decomposition. Both forms aim to reduce the size of OBDD. Exploiting literal equivalence, Lai et al. [25] proposed a generalization of Decision-DNNF, called CCDD, to capture literal equivalence. They demonstrate that CCDD supports model counting in linear time and design a model counter called ExactMC based on CCDD. In order to compute the Shannon entropy, the focus of this paper is to design a compiled language that supports the representation of probability distributions. Numerous target representations have been used to concisely model probability distributions. For example, d-DNNF can be used to compile relational Bayesian networks for exact inference [6]; Probabilistic Decision Graph (PDG) is a representation language for probability distributions based on BDD [18]. Macii and Poncino [28] utilized knowledge compilation to calculate entropy, demonstrating that ADD enables efficient and precise computation of entropy. However, the size of ADD often grows exponentially for large scale circuit formulas. To simplify ADD size, we propose an extended form, ADD[∧]. It uses conjunctive decomposition to streamline the graph structure and facilitate cache hits during construction.

7 Conclusion

In this paper, we propose a new compilation language, ADD[∧], which combines ADD and conjunctive decomposition to optimize the search process in the first stage of precise Shannon entropy computation. In the second stage of precise Shannon entropy computation, we optimize model counting queries by utilizing the shared component cache. We integrated preprocessing, heuristics, and other methods into the precise Shannon computation tool PSE, with its trace corresponding to ADD[∧]. Experimental results demonstrate that PSE significantly enhances the scalability of precise Shannon entropy computation, even outperforming the state-of-the-art entropy estimator EntropyEstimation in overall performance. We believe that PSE has opened up new research directions for entropy computing in Boolean formula modeling.

References

1 Michael Backes, Matthias Berg, and Boris Köpf. Non-uniform distributions in quantitative information-flow. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 367–375, 2011. doi:10.1145/1966913.1966960.

2 Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *2009 30th IEEE Symposium on Security and Privacy*, pages 141–153. IEEE, 2009. doi:10.1109/SP.2009.18.

3 R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebric decision diagrams and their applications. *Formal methods in system design*, 10:171–206, 1997. doi:10.1023/A:1008699807402.

4 Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986. doi:10.1109/TC.1986.1676819.

5 Pavol Cernỳ, Krishnendu Chatterjee, and Thomas A Henzinger. The complexity of quantitative information flow problems. In *2011 IEEE 24th Computer Security Foundations Symposium*, pages 205–217. IEEE, 2011.

6 Mark Chavira, Adnan Darwiche, and Manfred Jaeger. Compiling relational Bayesian networks for exact inference. *International Journal of Approximate Reasoning*, 42(1-2):4–20, 2006. doi:10.1016/J.IJAR.2005.10.001.

7 Arthur Choi, Doga Kisa, and Adnan Darwiche. Compiling probabilistic graphical models using sentential decision diagrams. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty: 12th European Conference, ECSQARU 2013, Utrecht, The Netherlands, July 8-10, 2013. Proceedings 12*, pages 121–132. Springer, 2013. doi:10.1007/978-3-642-39091-3_11.

8 Adnan Darwiche. New advances in compiling CNF to decomposable negation normal form. In *Proc. of ECAI*, pages 328–332. Citeseer, 2004.

9 Adnan Darwiche. *Modeling and reasoning with Bayesian networks*. Cambridge university press, 2009.

10 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002. doi:10.1613/JAIR.989.

11 Dorothy Elizabeth Robling Denning. *Cryptography and data security*, volume 112. Addison-Wesley Reading, 1982.

12 Jeffrey Dudek, Vu Phan, and Moshe Vardi. ADDMC: weighted model counting with algebraic decision diagrams. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1468–1476, 2020. doi:10.1609/AAAI.V34I02.5505.

13 Hélène Fargier, Pierre Marquis, Alexandre Niveau, and Nicolas Schmidt. A knowledge compilation map for ordered real-valued decision diagrams. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

14 Daniel Fremont, Markus Rabe, and Sanjit Seshia. Maximum model counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

15 Priyanka Golia, Brendan Juba, and Kuldeep S Meel. A scalable Shannon entropy estimator. In *International Conference on Computer Aided Verification*, pages 363–384. Springer, 2022. doi:10.1007/978-3-031-13185-1_18.

16 James W Gray III. Toward a mathematical foundation for information flow security. *Journal of Computer Security*, 1(3-4):255–294, 1992.

17 Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. *arXiv preprint*, 2013. arXiv:1301.6704.

18 Manfred Jaeger. Probabilistic decision graphs – Combining verification and AI techniques for probabilistic inference. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 12(supp01):19–42, 2004. doi:10.1142/S0218488504002564.

19 Vladimir Klebanov, Norbert Manthey, and Christian Muise. SAT-based analysis and quantification of information flow in programs. In *International Conference on Quantitative Evaluation of Systems*, pages 177–192. Springer, 2013. doi:10.1007/978-3-642-40196-1_16.

20 Tuukka Korhonen and Matti Järvisalo. Integrating tree decompositions into decision heuristics of propositional model counters. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, pages 8:1–8:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.8.

21 Marta Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. *Formal Methods for Performance Evaluation: 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures 7*, pages 220–270, 2007. doi:10.1007/978-3-540-72522-0_6.

22 Jean-Marie Lagniez and Pierre Marquis. On preprocessing techniques and their impact on propositional model counting. *Journal of Automated Reasoning*, 58:413–481, 2017. doi:10.1007/S10817-016-9370-8.

23 Yong Lai, Dayou Liu, and Shengsheng Wang. Reduced ordered binary decision diagram with implied literals: A new knowledge compilation approach. *Knowledge and Information Systems*, 35:665–712, 2013. doi:10.1007/S10115-012-0525-6.

24 Yong Lai, Dayou Liu, and Minghao Yin. New canonical representations by augmenting obdds with conjunctive decomposition. *Journal of Artificial Intelligence Research*, 58:453–521, 2017. doi:10.1613/JAIR.5271.

25 Yong Lai, Kuldeep S Meel, and Roland HC Yap. The power of literal equivalence in model counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3851–3859, 2021.

26 Yong Lai, Zhenghang Xu, and Minghao Yin. Pbcounter: weighted model counting on pseudo-boolean formulas. *Frontiers of Computer Science*, 19(3):193402, 2025. doi:10.1007/S11704-024-3631-1.

27 Nian-Ze Lee, Yen-Shi Wang, and Jie-Hong R Jiang. Solving exist-random quantified stochastic boolean satisfiability via clause selection.c. In *IJCAI*, pages 1339–1345, 2018. doi:10.24963/IJCAI.2018/186.

28 Enrico Macii and Massimo Poncino. Exact computation of the entropy of a logic circuit. In *Proceedings of the Sixth Great Lakes Symposium on VLSI*, pages 162–167. IEEE, 1996. doi:10.1109/GLSV.1996.497613.

29 Ziyuan Meng and Geoffrey Smith. Calculating bounds on information leakage using two-bit patterns. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, pages 1–12, 2011.

30 Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S Păsăreanu. Symbolic quantitative information flow. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012. doi:10.1145/2382756.2382791.

31 Tian Sang, Paul Beame, and Henry Kautz. Heuristics for fast exact model counting. In *Theory and Applications of Satisfiability Testing: 8th International Conference, SAT 2005, St Andrews, UK, June 19-23, 2005. Proceedings 8*, pages 226–240. Springer, 2005. doi:10.1007/11499107_17.

32 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S Meel. GANAK: A Scalable Probabilistic Exact Model Counter. In *IJCAI*, volume 19, pages 1169–1176, 2019. doi:10.24963/IJCAI.2019/163.

33 Geoffrey Smith. On the foundations of quantitative information flow. In *International Conference on Foundations of Software Science and Computational Structures*, pages 288–302. Springer, 2009. doi:10.1007/978-3-642-00596-1_21.

34 Ziqiao Zhou, Zhiyun Qian, Michael K Reiter, and Yinqian Zhang. Static evaluation of noninterference using approximate model counting. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 514–528. IEEE, 2018. doi:10.1109/SP.2018.00052.

A Comparison with precise Shannon entropy computing methods

In the appendix, we compare PSE with the state-of-the-art precise methods of computing Shannon entropy. The existing precise Shannon entropy tools do not use the techniques in the state-of-the-art model counters. Just like [15], we implemented the precise Shannon entropy baseline with state-of-the-art model counting techniques. In the baseline, we enumerate each assignment $\sigma \in Sol(\varphi)_{\downarrow Y}$ and compute $p_\sigma = \frac{|Sol(\varphi(Y \mapsto \sigma))|}{|Sol(\varphi)_{\downarrow X}|}$, where $Sol(\varphi(Y \mapsto \sigma))$ denotes the set of solutions of $\varphi(Y \mapsto \sigma)$ and $Sol(\varphi)_{\downarrow X}$ denotes the set of solutions of φ projected to X. As can be seen from the previous proposition, $|Sol(\varphi)_{\downarrow X}|$ can be replaced by $|Sol(\varphi)|$. Finally, entropy is computed as $H(\varphi) = \sum_{\sigma \in 2^Y} -p_\sigma \log p_\sigma$. For a formula with an output set size of m, 2^m model counting queries are required. For model counting queries, we have adopted two different methods. One is to directly invoke the currently state-of-the-art model counters, and our experiment, SharpSAT-TD, Ganak, and ExactMC are employed. The other method involves utilizing knowledge compilation. Firstly, we construct an offline knowledge compilation language that supports linear model counting, and then perform online conditioning based on each assignment over the Y variables. The knowledge compilation

language in our experiment is (OBDD[∧] via KCBox), and this method corresponds to the baseline-Panini in Table 2. Panini is an efficient compilation tool that supports the compilation of CNF formulas into the form of OBDD[∧] to enable efficient model counting.

Table 2 Entropy computation performance of baselines and PSE. "–" represents that the entropy cannot be computed within the specified time limit.

instance	$\|X\|$	$\|Y\|$	baseline-SharpSAT-TD		baseline-ExactMC		baseline-Ganak		baseline-Panini		PSE	
			Entropy	Time(s)	Entropy	Time(s)	Entropy	Time(s)	Entropy	Time(s)	Entropy	Time(s)
blasted_case102.cnf	11	23	8	81.29	8	0.39	8	35.42	8	1.82	8	0.16
s27_15_7.cnf	7	25	3.3	142.77	3.3	0.15	3.3	0.25	3.3	0.31	3.3	0.14
small-bug1-fixpoint-5.cnf	66	21	12.81	79.51	12.81	2.99	12.81	104.17	12.81	8.79	12.81	0.18
small-bug1-fixpoint-6.cnf	79	25	15.31	1406.47	15.31	51.9	15.31	1846.36	15.31	127.70	15.31	0.21
blasted_case144.cnf	138	627	–	–	–	–	–	–	–	–	77.62	35.42
s1423a_15_7.cnf	91	773	–	–	–	–	–	–	–	–	88.17	232.65
s382_15_7.cnf	24	326	–	–	–	–	–	–	–	–	23.58	0.22
CVE-2007-2875.cnf	752	32	–	–	–	–	–	–	–	–	32	0.72
10.sk_1_46.cnf	47	1447	–	–	–	–	–	–	–	–	13.58	0.18

Our experimental results indicate that all four representative state-of-the-art exact Shannon entropy baselines can only solve 18 benchmarks within the time limit of 3000 seconds, whereas PSE can solve 332 benchmarks. Table 2 shows the comparison between baselines and PSE on some instances. Notably, although some instances have similar sizes of X and Y sets, their computation times vary significantly (e.g., blasted_case144.cnf vs. s1423a_15_7.cnf). To clarify, computation times depend on multiple parameters, such as an exponential relationship with treewidth in addition to problem size. We employ the minfill heuristic to compute tree decompositions, guiding the entropy calculation. Our experimental results show that blasted_case144.cnf has a minfill treewidth of 22, whereas s1423a_15_7.cnf has a minfill treewidth of 27. The results show a significant improvement in the efficiency of PSE for computing the precise Shannon entropy. We remark that the poorer performance of these baselines is due to the exponential size of 2^Y.

Improving Reduction Techniques in Pseudo-Boolean Conflict Analysis

Orestis Lomis
DTAI, KU Leuven, Belgium

Jo Devriendt
Nonfiction Software, Leuven, Belgium

Hendrik Bierlee
DTAI, KU Leuven, Belgium

Tias Guns
DTAI, KU Leuven, Belgium

Abstract

Recent pseudo-Boolean (PB) solvers leverage the cutting planes proof system to perform SAT-style conflict analysis during search. This process learns implied PB constraints, which can prune later parts of the search tree and is crucial to a PB solver's performance. A key step in PB conflict analysis is the *reduction* of a reason constraint, which caused a variable propagation that contributed to the conflict. While necessary, reduction generally makes the reason constraint less strong. Consequently, different approaches to reduction have been proposed, broadly categorised as division- or saturation-based, with the aim of preserving the strength of the reason constraint as much as possible.

This paper proposes two novel techniques in each reduction category. We theoretically show how each technique yields reason constraints which are at least as strong as those obtained from existing reduction methods. We then evaluate the empirical effectiveness of the reduction techniques on hard knapsack instances and the most recent PB'24 competition benchmarks.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Constraint Programming, Pseudo-Boolean Reasoning, Conflict Analysis

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.21

Supplementary Material *Dataset (Experimental Results)*: `https://github.com/ML-KULeuven/SAT25_PB_reductions_experiments` [19]
archived at `swh:1:dir:650c2f9b37926504a59a11a6dbf319620b311dc3`

Funding This research was partly funded by the European Research Council (ERC) under the EU Horizon 2020 research and innovation programme (Grant No 101002802, CHAT-Opt).

1 Problem setting

Although the Boolean satisfiability (SAT) problem is NP-complete [7], in recent decades *conflict-driven clause learning (CDCL)* solvers [20, 1] routinely solve problems with up to millions of variables. However, the resolution proof system on which they are based may require exponential solve-time for some problems, such as the pigeonhole problem [14]. For this reason, higher-level proof systems pose a promising alternative, such as the cutting planes proof system [8]. Cutting planes form the underlying proof system of conflict-driven *pseudo-Boolean (PB)* constraint solvers, where we especially focus on PB constraints that are (weighted) linear inequalities over Boolean variables. The cutting planes proof system generalises the resolution proof system by using such PB constraints instead of clauses, which allow for more powerful reasoning. In theory this means that PB solvers can solve problems like the pigeonhole problem in polynomial time, where CDCL-based solvers would need


© Orestis Lomis, Jo Devriendt, Hendrik Bierlee, and Tias Guns;
licensed under Creative Commons License CC-BY 4.0

exponential time. The effectiveness of good PB reasoning has been shown in the most recent PB competition of 2024 [24], where PB solvers such as *RoundingSat* [12] and its offspring *Exact* [11] show top performance with their native implementation of the cutting planes proof system. *Exact* and *RoundingSat* are not only used as stand-alone solvers, but also as PB oracles in other PB solvers such as Hybrid-CASHWMaxSATDisjCadSP+Exact [22], mixed-bag [17] and IPBHS [16].

However, there still lies a difficulty for PB solvers to use the improved reasoning power of the cutting planes proof system effectively. In CDCL, the resolution of two conflicting clauses can simply be done by taking the union of two conflicting clauses and leaving out the propagated variable. While resolution can be generalised, PB solvers require an additional so-called *reduction step* to ensure the eventual learned constraint prevents the original conflict [5]. Additionally, the stronger the learned constraint the more pruning it will be able to do, hence the need for reduction techniques that keep the reason constraint as strong as possible. Consequently, various types of reduction have been investigated [12, 18, 21]. In this work we theoretically motivate and investigate four new reduction methods that are at least as strong as existing reduction methods.

Our contributions. First, we propose two novel variants of division-based reduction, namely Weaken Superfluous (WS) and Anti-Weakening of non-falsifieds (AW). Both exploit the rounding behaviour of the division operation in division-based reduction. We prove how the reduced constraints using these techniques are at least as strong as the ones obtained from using standard division-based reduction. Furthermore, we show how after applying the WS technique, the division operation is equivalent to the Mixed-Integer Rounding (MIR) operation, even though generally MIR dominates division [21]. Secondly, we propose two novel variants of a saturation-based reduction technique known as Multiply and Weaken (MW) [18], which we call Multiply and Weaken Direct (MWD) and Multiply and Weaken Indirect (MWI). These variants hybridize saturation- and division-based reduction by applying the former when favourable, using the latter as fallback. Then, we show that MWD with MWI produces constraints at least as strong as those from MWD alone. Finally, experiments empirically evaluate how the theoretically stronger reduction techniques impact solver performance on crafted and competition PB problems.

The organisation of the paper is as follows. In Section 2 we review the basics of PB solving, starting from CDCL and working our way to the current state-of-the-art division-based reduction method in *RoundingSat*. In Section 3 and Section 4 we introduce our novel techniques for division- and saturation-based reduction methods respectively. Section 5 contains the results from our experiments. In Section 6 we conclude our finding and discuss future work.

2 Preliminaries

We use notation and terminology from [10]. The term *pseudo-Boolean (PB) constraint* refers to a 0-1 linear inequality. We identify 1 with *true* and 0 with *false*. A *literal* ℓ denotes either a variable x or its negation $\bar{x} = 1 - x$. We assume w.l.o.g. that all constraints $\sum_i c_i \ell_i \geq \delta$ are written in *normalized form*, where literals ℓ_i are over pairwise distinct variables, coefficients c_i are positive integers, and δ is a positive integer called the *degree*. For a constraint C, $\mathit{lits}(C)$ denotes its set of literals and $\mathit{coeff}(\ell_i, C)$ denotes the coefficient of literal ℓ_i. A PB constraint C with degree 1 is a *clause*.

The *(partial) assignment* ρ is an ordered set of literals over pairwise distinct variables. A literal ℓ is *assigned to true* by an assignment ρ if $\ell \in \rho$, *assigned to false* or *falsified* if $\bar{\ell} \in \rho$, and is *unassigned* otherwise. We define the *slack* σ of a constraint $C = \sum_i c_i \ell_i \geq \delta$ under a partial assignment ρ as:

$$slack(C, \rho) = \left(\sum_{\ell_i \in lits(C), \bar{\ell}_i \notin \rho} c_i \right) - \delta \tag{1}$$

In other words, the slack measures how far ρ is from falsifying the constraint. Then, we say that ρ *falsifies* C if $slack(C, \rho) < 0$. A *pseudo-Boolean formula* φ is a set of PB constraints. An assignment ρ is a *solution* to φ if ρ satisfies all constraints in φ. A formula is *satisfiable* if it has a solution.

2.1 Conflict-Driven Pseudo-Boolean Search

Conflict-driven PB solving generalizes the CDCL algorithm for SAT, but uses PB constraints instead of clauses. The state of a PB solver can be represented by a ψ and ρ, where ψ is a set of constraints called the *constraint database*. Initially, ψ is the input formula φ and ρ is the empty set $\{\}$.

Given a solver state, the search loop starts with a *propagation* phase, which checks for any constraint $C \in \psi$ whether it is falsified:

$$slack(C, \rho) < 0, \tag{2}$$

or whether a literal ℓ_i in C with coefficient c_i, where ℓ_i has not yet been assigned by ρ, is implied by C under ρ:

$$slack(C, \rho) - c_i < 0 \text{ with } \ell_i \notin \rho, \bar{\ell}_i \notin \rho. \tag{3}$$

If condition (3) holds, C is falsified by $\rho \cup \bar{\ell}_i$, so ℓ_i is implied by C under ρ. For an assignment ρ we write ℓ_i / C to denote that C is the *reason* for the propagation of ℓ_i, and also use the notation $C = reason(\ell_i, \rho)$. Each propagation can enable new propagations, continuing the propagation phase until condition (3) does not hold for any constraint in the database ψ or until condition (2) holds for at least one. Note that unlike clauses, a single PB constraint can propagate multiple literals, even at different propagation phases.

If condition (2) holds for some constraint, it is considered a *conflict* and the constraint is denoted as the *conflict constraint*. On conflict, the solver enters a *conflict analysis* phase. During this phase, the solver derives a *learned constraint* which is a logical consequence of the current set of reason constraints combined with the conflict constraint. Crucially, the learned constraint must propagate a literal at some earlier search depth, hence preventing the current conflict from occurring again. Then, this learned constraint is added to ψ, after which the solver *backjumps* to a sufficient early search depth.

Alternatively, if no conflict is detected, the solver extends ρ by making a heuristic *decision* to assign some currently unassigned variable. If ℓ_i is a decision then it has no associated reason constraint, which we denote by $\ell_i / \cdot$.

The PB solver reports unsatisfiability whenever it learns a constraint equivalent to the trivial inconsistency $0 \geq 1$. If propagation does not lead to a conflict and all variables have been assigned, the solver reports that the input formula is satisfiable.

2.2 PB Conflict Analysis

In this subsection we will go into more detail about the conflict analysis phase of PB solvers, where the following operations are used [2, 4]:

Addition.

$$\frac{\Sigma c_i \ell_i \geq \delta \quad \Sigma c'_i \ell'_i \geq \delta'}{\Sigma(c_i \ell_i + c'_i \ell'_i) \geq (\delta + \delta')} \text{ Add} \tag{4}$$

Where we implicitly assume that the result is rewritten in normalised form.

▶ **Example 1.** Addition of the constraint $x + \bar{y} \geq 2$ with $y + z \geq 1$ yields $x + y + \bar{y} + z \geq 3$, which normalises to $x + z \geq 2$ by *cancelling* literals y and $\bar{y}$, where $y + \bar{y} = 1$.

Division.

$$\frac{\Sigma c_i \ell_i \geq \delta}{\Sigma \lceil \frac{c_i}{d} \rceil \ell_i \geq \lceil \frac{\delta}{d} \rceil} \text{ Div. by } d \in \mathbb{N}_0 \tag{5}$$

Multiplication.

$$\frac{\Sigma c_i \ell_i \geq \delta}{\Sigma \mu c_i \ell_i \geq \mu \delta} \text{ Mul. by } \mu \in \mathbb{N}_0 \tag{6}$$

Saturation.

$$\frac{\Sigma c_i \ell_i \geq \delta}{\Sigma\ min(c_i, \delta) \ell_i \geq \delta} \text{ Sat.} \tag{7}$$

Weakening.

$$\frac{c\ell + \Sigma c_i \ell_i \geq \delta}{(c - m)\ell + \Sigma c_i \ell_i \geq \delta - m} \text{ Wkn. } \ell \text{ by } m \in \mathbb{N}_0 \tag{8}$$

Weakening is *partial* when $m < c$, and *full* when $m = c$.

Anti-weakening

$$\frac{c\ell + \Sigma c_i \ell_i \geq \delta}{(c + m)\ell + \Sigma c_i \ell_i \geq \delta} \text{ Anti-Wkn. } \ell \text{ by } m \in \mathbb{N}_0 \tag{9}$$

Using these operations, PB solvers often implement different variants of conflict analysis, [6, 12, 3, 11]. We give a general outline in Algorithm 1 [21]. Conflict analysis starts from the conflict constraint C_{co}. We call the last literal of the current assignment, the reason literal ℓ_r. If it was not propagated, or $\bar{\ell_r} \notin \mathit{lits}(C_{co})$, then it did not contribute to the conflict, so it can be removed from the assignment ρ and we continue with the literal propagated just before it. If it is propagated and $\bar{\ell_r} \in \mathit{lits}(C_{co})$, then we should replace $\bar{\ell_r}$ with its reason constraint $C_{rsn} = \mathit{reason}(\ell_r, \rho)$ by addition of the two constraints [15, 4], which requires *reducing* the reason constraint as explained below. When the new C_{co} is propagating we have a learned constraint, and we can exit the conflict analysis phase.

Algorithm 1 analyzeConflict.

Input: Conflict constraint C_{co}, falsifying partial assignment ρ
Output: Learned constraint C_l
while C_{co} *is not propagating* **do**
  $\ell_r \leftarrow$ last literal of assignment ρ
  if ℓ_r *is propagated* $\wedge\ \bar{\ell}_r \in \mathit{lits}(C_{co})$ **then**
    $C_{rsn} \leftarrow \mathit{reason}(\ell_r, \rho)$
    $(C_{red}, C_{co}) \leftarrow \mathit{reduce}(C_{rsn}, C_{co}, \ell_r, \rho)$
    $C_{co} \leftarrow C_{red} + C_{co}$
  $\rho \leftarrow \rho \setminus \{\ell_r\}$
return C_{co}

To preserve the conflict during addition of C_{rsn} and C_{co}, there are two key requirements which make conflict-driven learning non-trivial for PB solving: The first one is that (i) adding C_{rsn} to C_{co} should indeed eliminate $\bar{\ell}_r$. Secondly, the learned constraint must propagate at an earlier stage, therefore (ii) it needs to remain conflicting under the current assignment. Since these requirements generally do not hold [4, Chapter 7], we need a *reduction step*, where $(C_{red}, C_{co}) = \mathit{reduce}(C_{rsn}, C_{co}, \ell_r, \rho)$ [13] such that the requirements are enforced. A sufficient condition for (i) is that the reduced reason has the same coefficient for $\bar{\ell}_r$ as the conflict constraint has for ℓ_r. A sufficient condition for (ii) is that after reduction, the reduced reason has slack 0 or less, since $\mathit{slack}(C_{co}, \rho) < 0$ by definition and slack is *subadditive* [12]: adding two constraints yields a constraint with a slack at most the sum of the slacks of the two original constraints. Hence formally we have the following requirements:

Cancelling Coefficients (Requirement 1) $\mathit{coeff}(\ell_r, C_{red}) = \mathit{coeff}(\bar{\ell}_r, C_{co})$
Negative Slack Condition (Requirement 2) $\mathit{slack}(C_{red}, \rho) \leq 0$

Our work focuses on this reduction step. To compare between two outcomes of a reduction, we say that C is *stronger* than C' when C implies C' (i.e. every satisfying variable assignment to C is also a satisfying assignment to C') but not the other way around. C is *rationally stronger* than C' when the former implies the latter and not the other way around, considering assignments of rational values between the closed interval $[0, 1]$ to the variables. We say reduction method A dominates reduction method B, with reduced constraints C^A_{red} and C^B_{red} respectively, when for any input C^A_{red} rationally implies C^B_{red}.

We already mentioned that slack is subadditive under addition. Additionally, slack remains unchanged when weakening a non-falsified literal or anti-weakening a falsified literal; slack increases when weakening a falsified literal or anti-weakening a non-falsified literal; slack is multiplied by m when multiplying a constraint by m. Hence, to decrease the slack of an individual non-conflicting reason constraint, at least one division or saturation step is necessary. Saturation-based reduction was the first successful implementation of cutting planes for PB solving [3], however most state-of-the-art solvers opt for division-based reduction [12, 17, 11, 22]. We will now look more in depth at division-based reduction.

2.3 *RoundingSat*-style Reduction

We describe the division-based reduction approach proposed by *RoundingSat* [12], which consists of three steps.

Weaken Non-Divisible Non-Falsifieds (Step 1) In the first step, we weaken all non-falsified literals that have a coefficient not divisible by c_{rsn}, the reason literal ℓ_r's coefficient, so that all non-falsified literals become divisible by c_{rsn}. In the original *RoundingSat* paper, these literals were fully weakened, but in the most recent implementation of *RoundingSat*[1], non-falsified literals with coefficient c_i are partially weakened by $c_i \bmod c_{rsn}$, i.e. to the largest multiple of c_{rsn} smaller than c_i. This is a less aggressive version of weakening also discussed in [18].

Divide (Step 2) In the second step, we divide the resulting constraint by c_{rsn} and round up all coefficients as per Equation (5). Since the previous weakening guaranteed that all non-falsified literals are divisible by c_{rsn}, we have $\mathit{coeff}(\ell_r, C_{rsn}) = 1$ and no non-falsified literals are rounded up during division, which would have increased the slack. Furthermore, the authors have proven after this division step the slack is at most 0, because the divisor is larger than the slack [12, Proposition 3.1], hence satisfying Requirement 2. From their work it follows that:

▶ **Corollary 2.** *Given a constraint C_{rsn} with slack σ and a divisor $d \in \mathbb{N}_0$, if the coefficients of all non-falsified literals are divisible by d, then the slack after division is $\lfloor \frac{\sigma}{d} \rfloor$.*

Multiply (Step 3) In the third step, given that ℓ_r's coefficient is now 1, Requirement 1 can be satisfied by multiplying the reason constraint by the coefficient of the negated reason literal in the conflict c_{co}.

▶ **Example 3.** Given the reason constraint $C_{rsn} = x + 3y + 3z + 5w \geq 6$, a conflict constraint $C_{co} = 4y + 4\bar{w} \geq 4$, an assignment $\rho = \{\bar{x}/\cdot, \bar{y}/\cdot, z/C_{rsn}, w/C_{rsn}\}$, reason literal $\ell_r = w$ and divisor $d = \mathit{coeff}(\ell_r, C_{rsn}) = 5$.

Step 1 of this method is to weaken the non-divisible non-falsified literals. So z is weakened by 3. Then in Step 2, the constraint is divided by 5:

$$\dfrac{\dfrac{x + 3y + 3z + 5w \geq 6}{x + 3y + 5w \geq 3}\ \text{Wkn. } z \text{ by } 3}{x + y + w \geq 1}\ \text{Div. by } 5 \tag{10}$$

Note that Requirement 2 is now already satisfied. To satisfy Requirement 1 we need to perform Step 3 by multiplying $1x + 1y + 1w \geq 1$ by $\mathit{coeff}(\bar{\ell}_r, C_{co}) = 4$ to get the reduced reason $C_{red} = 4x + 4y + 4w \geq 4$, which satisfies Requirements 1 and 2. To obtain a new learned constraint we then add the reduced reason to C_{co} and get $C_l = 4x + 4y \geq 4$. This learned constraint will propagate y as soon as $\bar{x}$ is decided, thereby preventing the conflict, and possibly preventing similar conflicts in later search.

3 Division-Based Reduction Variants

In this section we propose two new variants of division-based reduction, based on the above. Notice that since the reduced constraint C_{red} must be implied by the reason constraint C_{rsn}, C_{red} is at best as strong as C_{rsn}. Hence, our goal is to design a reduction method such that C_{red} remains as strong as possible.

In Sections 3.1 and 3.2, we will exploit Corollary 2 and the behaviour of rounding during division-based reduction in two novel techniques. For each technique we show that Requirements 1 and 2 still hold and that each technique dominates division-based reduction without the technique. In Section 3.3, we show how the techniques interact and can be combined.

[1] This recent version also participated in the last PB competition [24].

3.1 Weaken Superfluous (WS)

When dividing a constraint with degree δ by divisor d, the post-division degree is $\delta' = \lceil \frac{\delta}{d} \rceil$. Due to the upward rounding, we can often lower δ by some amount θ without changing δ'. Specifically, $\delta' = \lceil \frac{\delta}{d} \rceil = \lceil \frac{\delta - \theta}{d} \rceil$ for any $\theta \leq (\delta - 1) \bmod d$. After Step 1 (weakening of non-divisible non-falsifieds) of the above division-based reduction, we set θ maximally to get $\theta = (\delta - 1) \bmod d$. We define a *superfluous* literal as:

▶ **Definition 4** (Superfluous Literal). *When dividing a constraint C by d, a literal ℓ_s with coefficient c_s is superfluous when $0 < c_s \bmod d \leq \theta$ with $\theta = (\delta - 1) \bmod d$.*

Hence, we can weaken a superfluous literal ℓ_s by $c_s \bmod d$ without altering the degree δ' obtained after Step 2. But now, ℓ_s is rounded down in Step 2, which makes the reduced constraint stronger. Thus we propose the WS reduction where we iteratively weaken superfluous literals until there are no more left. Since the non-falsifieds have already been weakened to be divisible, the superfluous literals are always falsified. Therefore the reason literal ℓ_r is never superfluous and its coefficient will not change compared to *RoundingSat*-style reduction. Thus Requirement 1 will still be satisfied after Step 3 as when applying this reduction.

▶ **Example 5.** Continuing with Example 3, after Step 1, we can weaken by a total of $\theta = (\delta - 1) \bmod d = (3 - 1) \bmod 5 = 2$ before Step 2. Thus x is a superfluous literal, so we weaken it as well.

$$\dfrac{\dfrac{\dfrac{x + 3y + 3z + 5w \geq 6}{x + 3y + 5w \geq 3}\ \text{Wkn. } z \text{ by } 3}{3y + 5w \geq 2}\ \text{Wkn. } x \text{ by } 1}{y + w \geq 1}\ \text{Div. by } 5 \tag{11}$$

The final constraints from Equations (10) and (11) now both satisfy Requirement 2 and after multiplication both will also satisfy Requirement 1, but the latter is stronger than the former.

We now prove division-based reduction with WS dominates division-based reduction without WS.

▶ **Proposition 6.** *Let $d \in \mathbb{N}_0$ be some divisor. Let $C = c_s\ell_s + \sum c_i\ell_i \geq \delta$ be a constraint with ℓ_s a superfluous literal, so $c_s \bmod d \leq \theta$. Let C' be the constraint after division by d. Let C^{WS} be the constraint after weakening the superfluous literal ℓ_s to the nearest divisible integer (so by $c_s \bmod d$), and then division by d.*

C^{WS} implies C' and the slack of C^{WS} is at most that of C'.

Proof. We know that after division $C' = \lceil \frac{c_s}{d} \rceil \ell_s + \sum \lceil \frac{c_i}{d} \rceil \ell_i \geq \lceil \frac{\delta}{d} \rceil$ and $C^{WS} = \lceil \frac{c_s - c_s \bmod d}{d} \rceil \ell_s + \sum \lceil \frac{c_i}{d} \rceil \ell_i \geq \lceil \frac{\delta - c_s \bmod d}{d} \rceil$. Note that $\lceil \frac{c_s - c_s \bmod d}{d} \rceil = \lceil \frac{c_s}{d} \rceil - 1$ and that $\lceil \frac{\delta - c_s \bmod d}{d} \rceil = \lceil \frac{\delta}{d} \rceil$ (since $c_s \bmod d \leq \theta$). Hence, C^{WS} only differs from C' in that the coefficient of ℓ_s is rounded down. This means $\mathit{antiweaken}(C^{WS}, \ell_s, 1) = C'$ and thus C^{WS} implies C'. ◀

From the proof, $\mathit{antiweaken}(C^{WS}, \ell_s, 1) = C'$, so after division, the slack of the constraint where a superfluous literal is weakened is at most that of the non-weakened variant. So weakening superfluous literals preserves whether Requirement 2 is satisfied after division.

3.1.1 Link to Mixed Integer Rounding (MIR)

It has been proposed to replace the division operation (Step 2) in division-based reduction by the Mixed Integer Rounding (MIR) operation [21]:

Mixed Integer Rounding (MIR).

$$\frac{\Sigma_i c_i \ell_i \geq \delta}{\sum_{\ell \in I_1} \lceil \frac{c_i}{d} \rceil \ell_i + \sum_{\ell_j \in I_2} (\lfloor \frac{c_j}{d} \rfloor + \frac{c_j \bmod d}{(\delta-1) \bmod d+1}) \ell_j \geq \lceil \frac{\delta}{d} \rceil} \text{ MIR by } d \in \mathbb{N}_0 \tag{12}$$

with

$$I_1 = \{\ell_i \mid c_i \bmod d > (\delta - 1) \bmod d \vee c_i \bmod d = 0\},$$

$$I_2 = \{\ell_j \mid 0 < c_j \bmod d \leq (\delta - 1) \bmod d\},$$

To obtain normalised PB constraints with integer coefficients, MIR is followed by multiplication by $((\delta - 1) \bmod d) + 1$.

It was shown that division-based reduction using the MIR operation in Step 2 dominates division-based reduction with the division operation [21]. However, we can show that if there are no superfluous literals, the two reduction variants are equivalent. Consequently, after weakening superfluous literals, using MIR in Step 2 provides no advantage anymore compared to using division.

▶ **Proposition 7.** *Let $C = \sum c_i \ell_i \geq \delta$ be a constraint and $d \in \mathbb{N}_0$ a divisor such that none of the literals ℓ_i are superfluous in C. Let C^{DIV} and C^{MIR} be the constraints obtained by applying the division and the MIR operation with d, respectively. Then $C^{DIV} = C^{MIR}$.*

Proof. As no literals are superfluous, for all literals it holds by Definition 4 that $c_i \bmod d \geq \theta$ or $c_i \bmod d = 0$. Hence, $I_2 = \emptyset$. In that case, Equation (12) simplifies to Equation (5), so $C^{DIV} = C^{MIR}$. ◀

This means that, when there are no superfluous literals (e.g. after removing them with WS) division and MIR are equivalent.[2] This is not the case when there are superfluous literals in the constraint. E.g. WS followed by division on constraint $x + 2y \geq 2$ with divisor 2 is stronger than just MIR on the same constraint. The opposite can also be true, e.g. for constraint $x + y + 2z \geq 2$ with divisor 2. There could be other cases where just the MIR operation, without doing WS first, may be stronger than WS combined with the division operation. Further analysis is left for future work.

3.2 Anti-Weaken Anti-Superfluous (AW)

When dividing a constraint with slack σ by divisor d, the post-division slack is $\lfloor \frac{\sigma}{d} \rfloor$ according to Corollary 2. Due to downward rounding, we can often increase σ by some amount κ without changing $\lfloor \frac{\sigma}{d} \rfloor$. Specifically, $\lfloor \frac{\sigma}{d} \rfloor = \lfloor \frac{\sigma+\kappa}{d} \rfloor$ for any $0 \leq \kappa \leq (d - \sigma - 1) \bmod d$. During Step 1 we set κ maximally to get $\kappa = (d - \sigma - 1) \bmod d$. We define an *anti-superfluous* literal as:

▶ **Definition 8** (Anti-Superfluous Literal). *When dividing a constraint C with slack σ by d, a non-falsified literal ℓ_{aw} with coefficient c_{aw} is anti-superfluous when $0 < d - (c_{aw} \bmod d) \leq \kappa$ with $\kappa = (d - \sigma - 1) \bmod d$.*

[2] Up to multiplication by a constant factor, which is needed for MIR.

During the weakening of non-divisible non-falsifieds, we can then anti-weaken ℓ_{aw} by $d - (c_{aw} \bmod d)$. This makes the literal divisible without it being weakened, while still not increasing the slack as part of division by d. Note that we can repeat this step until there are no more anti-superfluous literals left.

▶ **Example 9.** We can again look at the conflict from Example 3, but now apply AW. Instead of weakening z by 3, we can anti-weaken it by 2 and then divide by 5.

$$\begin{array}{l} \dfrac{x + 3y + 3z + 5w \geq 6}{x + 3y + 5z + 5w \geq 6} \text{ Anti-Wkn. } z \text{ by } 2 \\ \dfrac{}{x + y + z + w \geq 2} \text{ Div. by } 5 \end{array} \tag{13}$$

The final constraints from Equations (10) and (13) now both satisfy Requirement 2 and after multiplication will also satisfy Requirement 1, but the latter is stronger than the former.

We now prove that division-based reduction with anti-weakening non-falsifieds dominates division-based reduction without.

▶ **Proposition 10.** *Let d be some divisor, ρ a partial assignment, $C = c_{aw}\ell_{aw} + \sum_i c_i\ell_i \geq \delta$ a constraint with slack σ, and ℓ_{aw} an anti-superfluous literal, so $0 < d - (c_{aw} \bmod d) \leq \kappa$.*

Let C' be the constraint after Steps 1 and 2. Let C^{AW} be the constraint after Steps 1 and 2 where during Step 1 ℓ_{aw} is anti-weakened by $d - (c_{aw} \bmod d)$ instead of weakened by $c_{aw} \bmod d$.

C^{AW} implies C' and the slack of C^{AW} is equal to that of C'.

Proof. Since the total amount that is anti-weakened is at most κ, we know that the slacks of C' and C^{AW} are still equal after Step 2. Then, since ℓ_{aw} is anti-weakened for C^{AW} it holds that $\mathit{coeff}(\ell_{aw}, C') + 1 = \mathit{coeff}(\ell_{aw}, C^{AW})$. And since the slack of the two constraints is the same, the following equality holds for their respective degrees $\delta' + 1 = \delta^{AW}$. This means $\mathit{weaken}(C^{AW}, \ell_{aw}, 1) = C'$ and thus C^{AW} implies C'. ◀

Hence, we propose the AW reduction where we iteratively anti-weaken anti-superfluous literals until there are no more left. From Proposition 10 it follows that anti-weakening does not increase the slack (as it yields constraints that are at least as strong), preserving Requirement 2. And since in *RoundingSat*-style reduction the divisor is the reason literal's coefficient, the reason literal is never anti-superfluous and will thus be unchanged, satisfying Requirement 1.

3.3 Combining WS and AW reduction

There is an interesting dynamic between the WS and AW reduction methods. Both exploit Corollary 2 in a similar way, by temporarily increasing the slack, while ensuring Requirement 2 is not violated after division. In the context of division-based reduction, WS and AW are two sides of the same coin, where WS applies to falsified literals and AW to non-falsified literals.

▶ **Proposition 11.** *Let d be some divisor and ρ a partial assignment. Let $C = \sum c_i\ell_i \geq \delta$ be some constraint with slack σ, before Step 1. Let $\kappa = (d - \sigma - 1) \bmod d$. Let C' be that same constraint, with degree δ' after Step 1. Let $\theta = (\delta' - 1) \bmod d$. Then $\theta = \kappa$.*

Proof. Before Step 1, $\kappa = (d - \sigma - 1) \bmod d$. Weakening non-falsified literals does not alter this value. So after Step 1, $\theta = (\delta' - 1) \bmod d$. By rearranging Equation (1) and replacing δ' and since all non-falsifieds are divisible by d we get $\theta = (\delta - 1) \bmod d = (\sum_{\bar{\ell}_i \notin \rho} c_i - \sigma - 1) \bmod d = (-\sigma - 1) \bmod d = (d - \sigma - 1) \bmod d$. Therefore, $\theta = \kappa$. ◀

So the amount θ we can (anti-)weaken by is shared between the two techniques, i.e. if we weaken superfluous literals by θ', then we only have $\theta - \theta'$ left to anti-weaken anti-superfluous literals.

We can easily combine AW and WS in one divsion-based reduction approach, which we present in Algorithm 2. We first apply AW in lines 6 to 8, then WS in lines 10 to 14.

Algorithm 2 reduceDivision.

Input: Reason C_{rsn}, conflict C_{co}, reason literal ℓ_r, partial assignment ρ
Output: Tuple of reduced constraint C_{rsn} and conflict constraint C_{co}

```
1  d ← coeff(ℓ_r, C_rsn)
2  θ ← (d − slack(C_rsn, ρ) − 1) mod d
3  for ℓ_i ∈ lits(C_rsn) do
4      α ← coeff(C_rsn, ℓ_i) mod d
5      if α ≠ 0 ∧ ℓ̄_i ∉ ρ then
6          if θ − d + α ≥ 1 then
7              θ ← θ − d + α
8              continue
9          C_rsn ← weaken(C_rsn, ℓ_i, α)
10 for ℓ_i ∈ lits(C_rsn) do
11     α ← coeff(C_rsn, ℓ_i) mod d
12     if 0 < α ≤ θ then
13         θ ← θ − α
14         C_rsn ← weaken(C_rsn, ℓ_i, α)
15 C_rsn ← divide(C_rsn, d)
16 C_red ← coeff(ℓ̄_r, C_co) · C_rsn
17 return (C_red, C_co)
```

4 Saturation-Based Reduction Variants

In the previous section we focused on division-based reduction. In this section we will focus on the other family of reduction techniques: saturation-based reduction. We investigate a saturation-based method called *Multiply and Weaken* (MW) [18]. The main idea we use from MW is to multiply the reason and/or the conflict constraint in order to bring the coefficients of the reason literal in the reason c_{rsn} and conflict c_{co} close to each other, with $c_{rsn} \geq c_{co}$. We develop two reduction variants, both of which use weakening in a different manner to satisfy Requirement 1. As for Requirement 2, the MW reduction variants use the necessary, but less strict, requirement:

Weak Negative Slack Condition (Requirement 3) $slack(C_{red}, \rho) + slack(C_{co}, \rho) < 0$

4.1 Multiply and Weaken Direct (MWD)

The first MW reduction variant, *Multiply and Weaken Direct (MWD)*, applies the following operations. First, multiply C_{rsn} by $\left\lceil \frac{c_{co}}{c_{rsn}} \right\rceil$ and C_{co} by $\mu = \max(1, \left\lfloor \frac{c_{rsn}}{c_{co}} \right\rfloor)$. Then, weaken the reason literal ℓ_r in the multiplied reason constraint by the exact amount needed to

satisfy Requirement 1, i.e. by $\left\lceil \frac{c_{co}}{c_{rsn}} \right\rceil \cdot c_{rsn} - \mu \cdot c_{co}$. However, these operations will only yield a reduced reason and conflict constraint (i.e. meeting Requirement 3) if the following condition holds:

$$\left\lceil \frac{c_{co}}{c_{rsn}} \right\rceil \cdot slack(C_{rsn}, \rho) + \mu \cdot slack(C_{co}, \rho) < 0 \tag{14}$$

If Equation (14) does hold, then Requirement 3 is guaranteed to be satisfied, since the multiplications of the slack are taken into account and weakening of non-falsified literals does not increase the slack. If Equation (14) does not hold, MWD reduction uses a fallback reduction method instead. In some cases, MWD yields stronger constraints than division-based reduction:

▶ **Example 12.** Given a reason constraint $C_{rsn} = x + 2y + 3z + 5w \geq 5$, a conflict constraint $C_{co} = 3u + 4\bar{w} + 5y \geq 7$, an assignment $\rho = \{\bar{x}/\cdot, \bar{y}/\cdot, w/C_{rsn}\}$ and a reason literal $\ell_r = w$.

We show Steps 1 and 2 for division-based reduction:

$$\begin{array}{l} \dfrac{x + 2y + 3z + 5w \geq 5}{x + 2y + 5w \geq 2} \text{ Wkn. } z \text{ by } 3 \\ \dfrac{x + 2y + 5w \geq 2}{x + y + w \geq 1} \text{ Div. by } 5 \end{array} \tag{15}$$

For MWD reduction, we first need to check if it is even possible to satisfy Requirement 3. Since $\left\lceil \frac{c_{co}}{c_{rsn}} \right\rceil \cdot slack(C_{rsn}, \rho) + \mu \cdot slack(C_{co}, \rho) = \lceil \frac{4}{5} \rceil \cdot 3 + \max(1, \lfloor \frac{5}{4} \rfloor) \cdot (-4) = 3 - 4 = -1 < 0$ Requirement 3 will indeed be satisfied. If so, Requirement 3 will indeed be satisfied and we can continue with the MWD approach. Since no multiplication is needed all we need to do is weaken w by 1:

$$\frac{x + 2y + 3z + 5w \geq 5}{x + 2y + 3z + 4w \geq 4} \text{ Wkn. } w \text{ by } 1 \tag{16}$$

We can see that the reduced reason from MWD in Equation (16) is stronger than the one from division-based reduction Equation (15).

4.2 Multiply and Weaken Indirect (MWI)

While MWD always directly weakens the reason literal ℓ_r, another approach is possible when ℓ_r is *saturated*, i.e. if its coefficient is at least as high as the degree of the constraint. Instead of weakening ℓ_r directly, we can lower the degree by weakening a *different* non-falsified literal, and then apply saturation. This lowers ℓ_r's coefficient to the degree of the constraint, effectively giving ℓ_r the same coefficient as if it was weakened. In this case, we weaken two literals "for the price of one". Other than the different weakening approach followed by saturation of the reason constraint, MWI applies the same operations as MWD. We continue Example 12:

▶ **Example 13.** Instead of weakening w directly by 1, we can instead weaken the non-falsified literal z by 1 and then saturate C_{rsn} so that w gets the desired coefficient 4.

$$\begin{array}{l} \dfrac{x + 2y + 3z + 5w \geq 5}{x + 2y + 2z + 5w \geq 4} \text{ Wkn. } z \text{ by } 1 \\ \dfrac{x + 2y + 2z + 5w \geq 4}{x + 2y + 2z + 4w \geq 4} \text{ Sat.} \end{array} \tag{17}$$

Clearly, since the coefficient of z is lower, the reduced reason in Equation (17) is stronger than the one obtained by MWD in Equation (16).

Note that constraints obtained by MWI imply those obtained by MWD, as the only difference in both routines is that some coefficients are lowered for MWI, while the degree in the reduced reason remains exactly the same.

4.3 Combining MWD, MWI, and division-based reduction

We present the novel MW variants in Algorithm 3. On line 3 we check Equation (14) to see if Requirement 3 will hold after MWD. If it does, we multiply the constraint and calculate the amount we directly weaken ℓ_r by on line 17. Otherwise, we use a fallback reduction, in our case this is division-based reduction from Algorithm 2. Note that this allows us to combine the division-based reduction variants, AW and WS, with the saturation-based variants, MWD and MWI. Then from line 9 to 16, if the reason literal is saturated, we apply indirect weakening and saturation. We perform the final step of direct weakening and saturate again. In this fashion, we use MWI in combination with MWD since there is no guarantee that only MWI will always sufficiently reduce the reason coefficient.

Algorithm 3 reduceSaturation.

Input: Reason C_{rsn}, conflict C_{co}, reason literal ℓ_r, partial assignment ρ
Output: Tuple of reduced C_{rsn} and (potentially multiplied) C_{co}

1 $c_{rsn} \leftarrow \mathit{coeff}(C_{rsn}, \ell_r)$
2 $c_{co} \leftarrow \mathit{coeff}(C_{co}, \bar{\ell}_r)$
3 **if** $\left\lceil \frac{c_{co}}{c_{rsn}} \right\rceil \cdot \mathit{slack}(C_{rsn}, \rho) + \mu \cdot \mathit{slack}(C_{co}, \rho) < 0$ **then**
4 $\quad C_{rsn} \leftarrow \left\lceil \frac{c_{co}}{c_{rsn}} \right\rceil \cdot C_{rsn}$
5 $\quad C_{co} \leftarrow \mu \cdot C_{co}$
6 $\quad \alpha \leftarrow \left\lceil \frac{c_{co}}{c_{rsn}} \right\rceil \cdot c_{rsn} - \mu \cdot c_{co}$
7 **else**
8 $\quad$ **return** $\mathit{reduceDivision}(C_{rsn}, C_{co}, \ell_r, \rho)$
9 **if** $c_{rsn} \geq \delta_{C rsn}$ **then**
10 $\quad$ **for** $\ell_i \in C_{rsn}$ *with coefficient* $c_i \wedge \ell_i \notin \rho$ **do**
11 $\quad\quad$ **if** $c_i > \alpha$ **then**
12 $\quad\quad\quad C_{rsn} \leftarrow \mathit{weaken}(C_{rsn}, \ell_i, \alpha)$
13 $\quad\quad\quad \alpha \leftarrow 0$
14 $\quad\quad$ **else**
15 $\quad\quad\quad C_{rsn} \leftarrow \mathit{weaken}(C_{rsn}, \ell_i, c_i)$
16 $\quad\quad\quad \alpha \leftarrow \alpha - c_i$
17 $\quad C_{rsn} \leftarrow \mathit{saturate}(C_{rsn})$
18 $C_{rsn} \leftarrow \mathit{weaken}(C_{rsn}, \ell_r, \alpha)$
19 **return** $(\mathit{saturate}(C_{rsn}), C_{co})$

5 Experimental Evaluation

In this section, we evaluate the impact of the proposed techniques on solver performance. We implemented our techniques into *Exact* 2.1.0 [11][3], which is a fork of *RoundingSat* [12]. We use three benchmark sets:

KNAP crafted knapsack (783 instances) [23, 9]
DEC-LIN the decision linear track of the PB'24 competition (398 instances)
OPT-LIN the optimisation linear track of the PB'24 competition (489 instances)

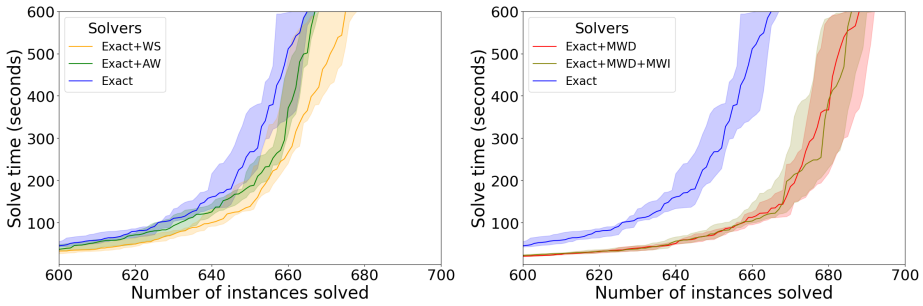
[3] The earlier version of *Exact* submitted to the PB'24 competition already incorporated these techniques

For KNAP the solver is given a memory limit 8GB of RAM and a timeout of 600 seconds. For DEC-LIN and OPT-LIN the solver is given a memory limit of 31GB of RAM and a timeout of 3600 seconds as in the PB'24 competition. The KNAP benchmark set is added because each instance consists of a single constraint, which puts a heavy emphasis on learning strong constraints. Our experiments were run on a cluster with 32 INTEL(R) XEON(R) SILVER 4514Y (2GHz) CPUs with 256GB of shared memory. We run each configuration of the solver with 5 seeds. Our figures are plotted with performance on the x-axis and solve time on the y-axis. Optimisation instances are considered solved if an optimal solution is found *and* it is proven that no better exists. We use the results from the five seeds to plot 95% confidence bands to visualize potential variance in solve-times, with a solid line for the median. The implementation and run logs are included as supplemental material.

5.1 Individual techniques

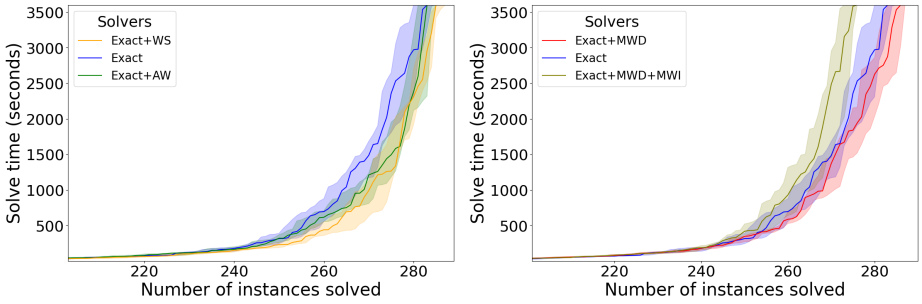
Our first experiment compares each individual technique (i.e. the WS, AW, MWD, and MWD+MWI reduction methods) to *Exact* as a baseline. For the division-based techniques we see that on KNAP in Figure 1a, WS solves more instances and AW solves some instances faster. We also see positive results when it comes to the saturation-based techniques in Figure 1b. MWD and MWD+MWI both perform similarly, while outperforming the baseline. On DEC-LIN we see similar trends for the division-based techniques in Figure 2a. WS still solves more instances than the baseline and AW mostly helps solving some instances faster. For the saturation-based techniques we do see in Figure 2b a different trend compared to KNAP. While MWD still helps improve the performance of the solver, MWD+MWI shows a negative performance. This unexpected result is interesting, since we know MWD+MWI dominates MWD from a theoretical perspective. On OPT-LIN we do not see much of a difference either way for the division-based techniques in Figure 3a, but the performance marginally worsens for the saturation-based techniques in Figure 3b.

We believe the smaller impact of AW is due to how each technique changes the constraint. WS only lowers one coefficient while keeping everything else the same, while AW increases the degree as well as a coefficient. The increase in the degree could however lead to fewer saturation opportunities after addition with the conflict constraint, reducing the impact. On the other hand, lowering a coefficient via WS may lead to fewer variable cancellations and in turn less saturation after addition, though the odds for cancellation on non-propagated literals may be relatively smaller. In the end, both techniques show only moderate impact on the selected benchmarks.



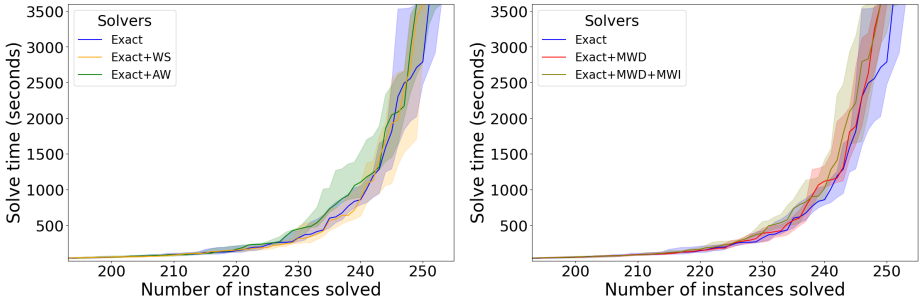
(a) KNAP. **(b)** KNAP.

Figure 1 Comparing individual division- and saturation-based techniques on KNAP.



(a) DEC-LIN. **(b)** DEC-LIN.

Figure 2 Comparing individual division- and saturation-based techniques on DEC-LIN.



(a) OPT-LIN. **(b)** OPT-LIN.

Figure 3 Comparing individual division- and saturation-based techniques on OPT-LIN.

5.2 Combined techniques

In our second experiment we evaluate how combining the different techniques as shown in Section 4.3 impacts the empirical runtime of PB solvers. Additionally, we test an implementation of the combined division-based techniques, WS+AW, on another solver, namely the PB'24 competition version of *RoundingSat*, to see how it behaves in different solvers. Implementing the saturation-based techniques in *RoundingSat* would have required more extensive changes due to overflows after multiplication.

The results are summarised in Table 1. On KNAP, combining all the techniques has a compounding effect for both solvers. On DEC-LIN, the results are less clear. *Exact+WS+AW* does perform very well, which might initially seem surprising since according to Proposition 11 applying AW means less WS is possible. *RoundingSat+WS+AW*, on the other hand has minor impact on *RoundingSat* for DEC-LIN. Still, it seems both techniques can be combined effectively. The same cannot be said when mixing the division- and saturation-based techniques. We hypothesize that since WS+AW improves division-based reduction, a better heuristic than Equation (14) for the saturation-based techniques is necessary. It may be that WS+AW is most effective in many cases where MWD is actually possible, thus the improvements from WS+AW carry over to WS+AW+MWD. On OPT-LIN, the choice of configuration of the techniques still does not seem to have much impact, with similar performance to the configurations using an individual technique.

Table 1 Median number of solved instances for different solver configurations across benchmarks.

Solved	KNAP (783)	DEC-LIN (398)	OPT-LIN (489)
Exact	664	282	250
Exact+WS	674	284	249
Exact+AW	666	282	248
Exact+WS+AW	683	**287**	247
Exact+MWD	687	285	248
Exact+MWD+MWI	685	275	248
Exact+WS+AW+MWD	695	284	248
Exact+WS+AW+MWD+MWI	698	280	249
RoundingSat	691	281	**253**
RoundingSat+WS+AW	**709**	282	251

6 Conclusion and Future Work

We presented novel techniques to generate stronger reduced constraints in both division-based [12] and saturation-based [18] reduction methods. As in established work [21], we can indeed prove dominance relationships between the various reduction methods, which guarantee that reduced constraints obtained from one are at least as strong as those from another. The experiments show that stronger reduced constraints can improve the solver performance for different solvers and benchmarks, but not uniformly across all problems. While there are improvements on crafted knapsack benchmarks, and the competition decision benchmark for *Exact*, we observe little difference on competition optimisation benchmarks. Perhaps more surprisingly, in competition decision benchmarks we see a case of worsening performance for MWD+MWI compared to MWD, despite their dominance relationship.

Hence our theoretical results provide a better understanding of reduction methods and the freedom there is in reducing constraints before addition. Empirically there is a lesser understood relationship between the strength of the reduced reason constraint, the strength of the learned constraint after all iterations of constraint addition in the conflict analysis, and the effect of the learned constraints on solver performance. However, these relationships are complex, because they involve the reduction and resolution of multiple reason constraints with the conflict constraint. With the insights of the paper we also see avenues to strengthen the reduced constraints further. For example, in division-based reduction, relaxing Requirement 2 to Requirement 3 (as in saturation-based reduction) could lead to a smaller divisor or an increase in the amount of superfluous and anti-superfluous literals.

We also saw how some combinations of reduction techniques can be effective, but they use heuristics, e.g. to choose between division- and saturation-based reduction for specific constraints. These heuristics are much less studied and can have a big impact on empirical performance which deserves further study.

References

1 Gilles Audemard and Laurent Simon. On the glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001:1–1840001:25, February 2018. doi:10.1142/S0218213018400018.

2 Danel Le Berre, Pierre Marquis, Stefan Mengel, and Romain Wallon. On Irrelevant Literals in Pseudo-Boolean Constraint Learning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pages 1148–1154, July 2020. doi:10.24963/ijcai.2020/160.

3 Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *J. Satisf. Boolean Model. Comput.*, 7(2-3):59–6, 2010. doi:10.3233/sat190075.

4 Sam Buss and Jakob Nordström. Proof complexity and 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 233–350. IOS Press, 2021. doi:10.3233/FAIA200990.

5 Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *Proceedings of the 40th Annual Design Automation Conference*, pages 830–835, Anaheim CA USA, June 2003. ACM. doi:10.1145/775832.776041.

6 Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24:305–317, 2005. doi:10.1109/TCAD.2004.842808.

7 Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. doi:10.1145/800157.805047.

8 W. Cook, C.R. Coullard, and Gy. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, September 1987. doi:10.1016/0166-218X(87)90039-4.

9 Jo Devriendt. Pisinger's knapsack instances in opb format, July 2020. doi:10.5281/zenodo.3939055.

10 Jo Devriendt. Watched Propagation of 0-1 Integer Linear Constraints. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming*, pages 160–176, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-58475-7_10.

11 Jo Devriendt. Exact: Evaluating cutting-planes learning at the PB'24 competition. *Slides*, 2024.

12 Jan Elffers and Jakob Nordström. Divide and Conquer: Towards Faster Pseudo-Boolean Solving. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 1291–1299, Stockholm, Sweden, July 2018. International Joint Conferences on Artificial Intelligence Organization. doi:10.24963/ijcai.2018/180.

13 Jan Elffers and Jakob Nordström. RoundingSat 2019: Recent work on PB solving. https://slides.com/jod/deck-26-29-32, July 2019.

14 Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985. Third Conference on Foundations of Software Technology and Theoretical Computer Science. doi:10.1016/0304-3975(85)90144-6.

15 John N Hooker. Generalized resolution and cutting planes. *Annals of Operations Research*, 12:217–239, 1988.

16 Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Ipbhs in pb'24 competition. *Slides*, 2024.

17 Christoph Jabs, Jeremias Berg, and Matti Järvisalo. PB-OLL-RS and MIXED-BAG in Pseudo-Boolean Competition 2024. *PB Competition 2024*, 2024.

18 Daniel Le Berre, Pierre Marquis, and Romain Wallon. On Weakening Strategies for PB Solvers. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020*, pages 322–331, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-51825-7_23.

19 Orestis Lomis, Jo Devriendt, Hendrik Bierlee, and Tias Guns. ML-KULeuven/SAT25_PB_reductions_experiments. Dataset, This research was partly funded by the European Research Council (ERC) under the EU Horizon 2020 research and innovation programme (Grant No 101002802, CHAT-Opt)., swhId: swh:1:dir:650c2f9b37926504a59a11a6dbf319620b311dc3 (visited on 2025-07-23). URL: https://github.com/ML-KULeuven/SAT25_PB_reductions_experiments, doi:10.4230/artifacts.24089.

20 J.P. Marques Silva and K.A. Sakallah. Grasp-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, 1996. doi:10.1109/ICCAD.1996.569607.

21 Gioni Mexi, Timo Berthold, Ambros Gleixner, and Jakob Nordström. Improving conflict analysis in mip solvers by pseudo-boolean reasoning. *arXiv preprint arXiv:2307.14166*, 2023. doi:10.48550/arXiv.2307.14166.

22 Shiwei Pan, Yiyuan Wang, Shaowei Cai, Jiangnan Li, Wenbo Zhu, and Minghao Yin. Cashwmaxsat-disjcad: Solver description. *MaxSAT Evaluation 2024*, 2024:25, 2024.

23 David Pisinger. Where are the hard knapsack problems? *Comput. Oper. Res.*, 32(9):2271–2284, 2005. doi:10.1016/j.cor.2004.03.002.

24 Roussel, Olivier. Pseudo-Boolean Competition of 2024. https://www.cril.univ-artois.fr/PB24/.

Enumerating All Boolean Matches

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

Yogev Shalmon
Intel Corporation, Haifa, Israel
Faculty of Data and Decision Sciences, Technion, Haifa, Israel

Abstract

Boolean matching, a fundamental problem in circuit design, determines whether two Boolean circuits are equivalent under input/output permutations and negations. While most works focus on finding a single match or proving its absence, the problem of enumerating all matches remains largely unexplored, with `BooM` being a notable exception. Motivated by timing challenges in Intel's library mapping flow, we introduce `EBat`– an open-source tool for enumerating all matches between single-output circuits. Built from scratch, `EBat` reuses `BooM`'s SAT encoding and introduces novel high-level algorithms and performance-critical subroutines to efficiently identify and block multiple mismatches and matches simultaneously. Experiments demonstrate that `EBat` substantially outperforms `BooM`'s baseline algorithm, solving 3 to 4 times more benchmarks within a given time limit. `EBat` has been productized as part of Intel's library mapping flow, effectively addressing the timing challenges.

2012 ACM Subject Classification Mathematics of computing → Solvers

Keywords and phrases Boolean Matching, All-Boolean-Matching, Enumeration, SAT, Generalization

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.22

Supplementary Material *Software*: `https://github.com/yogevshalmon/ebat`
archived at `swh:1:dir:4b747bd04b637c1e3c938cea66469b2ff0b5d2d9`

Acknowledgements We are grateful to Oded Asulin, Yossi Levani, Aviad Munitz, Pavel Nisanov and Hagay Segal for helpful discussions, which played an important role in shaping our research. We also thank Roland Jiang for providing us with the original code for `BooM`. Additionally, we would like to thank the anonymous reviewers for their valuable comments and insightful suggestions.

1 Introduction

Boolean matching is a pivotal problem of determining whether two Boolean circuits are equivalent under the permutation and negation of inputs and outputs. From the theoretical standpoint, Boolean matching lies between coNP and Σ_2^P [3, 10], which makes it a good candidate for examining the open question about the collapse of the polynomial hierarchy. On the practical front, Boolean matching is widely applied, including in library mapping (also known as library binding or technology mapping) [6, 28], synthesis [16, 43], Engineering Change Order (ECO) [22], equivalence checking [29] and protection against hardware Trojans [40]. Considerable attention has been devoted to solving Boolean matching, highlighted by its inclusion as a CAD contest problem at ICCAD'23 [15]. Existing solving methods can be categorized into canonical-form- [6, 29, 23], signature- [44, 1], and SAT-based [42, 26, 25]. While almost all existing works focus on finding a *single* match or proving its nonexistence, this paper is dedicated to the *all-Boolean-matching* problem of enumerating *all* the matches.

1.1 Motivation

This work was driven by a critical industrial need identified by engineers at Intel.


© Alexander Nadel and Yogev Shalmon;
licensed under Creative Commons License CC-BY 4.0



The modern semiconductor design process relies heavily on the standard-cell methodology, where designers build complex circuits using fundamental building blocks called *standard cells*, organized into *standard cell libraries*. *Library mapping* [6, 28] is the process of transitioning between libraries to re-implement the same logical functionality using a new library. Intel engineers found that standard Boolean matching-based library mapping often failed to meet timing requirements. Specifically, their timing tool revealed significant delay variances in matched pairs, whereas it is crucial to identify matches with minimal delay variance.

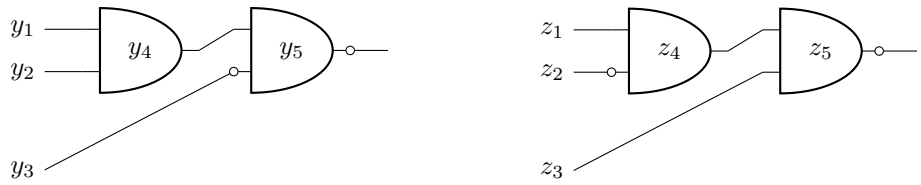
The key issue was that for a given pair of matching cells, multiple input mappings were possible, each resulting in a different delay, yet existing Boolean matching tools returned an arbitrary match. To address this, engineers requested a tool capable of enumerating *all* possible Boolean matches between two cells or proving their absence, so that they could run their timing tool on the results to identify the best match. Notably, the timing tool calculates a complex function that varies according to project specifications, making it infeasible to express the problem using an optimization paradigm like MaxSAT.

This paper presents `EBat`, our novel open-source tool developed to enumerate all possible Boolean matches or prove their absence, fulfilling the engineers' requirement. `EBat` has been productized and is actively used at Intel for library mapping.

We have strong reasons to believe that `EBat` will be valuable to both industrial practitioners and researchers. First, poor timing in library mapping is a common and critical challenge in semiconductor design. Second, a tool that efficiently enumerates all solutions to a widely used decision problem, such as Boolean matching, can open new research directions and practical applications.

1.2 Our Focus

Having clarified our motivation, this paper focuses on algorithmic solutions for all-Boolean-matching. We restrict ourselves to combinational circuits with a *single output*. In line with the literature (see, e.g., [26]), we distinguish between three types of equivalence: Permutation-Equivalence (*P-Equivalence*), where only input permutations are allowed; Negation-Permutation-Equivalence (*NP-Equivalence*), where inputs can also be negated; and Negation-Permutation-Negation-Equivalence (*NPN-Equivalence*), where both inputs and outputs can be negated. In the case of single-output circuits, NPN-equivalence can be reduced to two NP-equivalence checks: the first runs NP-equivalence on Y and Z as-is; the second runs it on Y and Z with Z's output negated. Consequently, this work focuses on *P-equivalence* and *NP-equivalence*. Fig. 1 shows an example of finding all Boolean matches for both P- and NP-equivalence.



(a) Circuit Y implementing $\neg((y_1 \wedge y_2) \wedge \neg y_3)$. **(b)** Circuit Z implementing $\neg((z_1 \wedge \neg z_2) \wedge z_3)$.

Figure 1 For P-equivalence, there are two matches between Y and Z: $[z_1, z_3, z_2]$ (that is, $y_1 \mapsto z_1$; $y_2 \mapsto z_3$; $y_3 \mapsto z_2$) and $[z_3, z_1, z_2]$. For NP-equivalence, there are these two plus four more: $[z_1, \neg z_2, \neg z_3]$, $[\neg z_2, z_1, \neg z_3]$, $[\neg z_2, z_3, \neg z_1]$, and $[z_3, \neg z_2, \neg z_1]$.

1.3 Previous Work: BooM

To our knowledge, BooM [26, 25] is the only Boolean matching approach capable of enumerating all matches. We refer to BooM's all-Boolean-matching algorithm as the *sifter* (BooMS). At a high level, BooMS sifts through a given bucket of mappings using a *mismatch-sifter*, filtering out mismatches and collecting the remaining mappings in a new bucket containing only matches, which are then reported to the user.

Specifically, BooMS constructs two SAT instances: `misms`, initialized to capture only mismatches, and `bucket`, initialized to include all possible mappings. BooMS iteratively visits each unvisited mismatch by querying `misms` for a solution π. It then blocks π and potentially other mismatches from *both* `misms` and `bucket` by adding the same *blocking clause* to both instances. For NP-equivalence, before blocking, the mismatch witness (that is, the solution) is further *extended* through *generalization* [35, 21] using *backward ternary simulation (aka justification)* [37, 39] to cover additional mismatches. Once `misms` returns UNSAT, `bucket` contains only matches as its solutions. These matches can then be enumerated by repeatedly querying `bucket` to obtain and block each match.

1.4 Our Contributions: New Algorithms and EBat

Our solution, EBat, implemented from scratch, reuses BooM's SAT encoding while introducing novel all-Boolean-matching algorithms based on the following insights.

The feasibility of all-Boolean-matching algorithms hinges on efficiently identifying and blocking multiple mismatches and matches simultaneously – a capability essential for scaling to real-world instances. We observed a fundamental limitation in the original BooMS algorithm: it enumerates, extends, and blocks only mismatches. In the following, we describe our contributions, starting with a new algorithm that addresses the above limitation:

- **Our first contribution** is a novel algorithm, named the *picker* (EBatP), which enumerates not only mismatches but also matches. Its core capability, which distinguishes it from BooMS, is explicitly visiting and *strengthening* matches using minimal unsatisfiable core extraction [20, 18] to cover and block multiple matches simultaneously, while still leveraging witness extension for efficient mismatch handling, similarly to BooMS. We observed EBatP breaking a performance bottleneck and solving significantly more instances, but only for NP-equivalence. Our analysis suggested this stemmed from the lack of witness extension for P-equivalence, as we followed BooMS, which omits this procedure since applying generalization for P-equivalence compromises correctness (details in Sect. 5). This led us to our second contribution.
- **Our second contribution** is a new witness extension algorithm for P-equivalence, achieved through a dedicated modification of SAT solver heuristics. With this approach, we successfully broke the performance bottleneck for P-equivalence as well. The following three algorithmic contributions further increase the number of solved instances:
- **Our third contribution** is a novel high-level "sift-and-pick" algorithm, EBatC, which combines BooMS and EBatP.
- **Our fourth contribution** is more efficient witness extension for NP-equivalence using a new generalization algorithm, inspired by our recent results in solution enumeration for circuits (AllSAT-CT) [21], which outperforms BooM's witness extension method (i.e., generalization via backward ternary simulation).
- **Our fifth contribution** is a novel, dedicated mismatch-blocking algorithm for P-equivalence.

- Finally, **our sixth contribution** comprises the implementation of all our algorithms and the baseline `BooM` algorithm in a new open-source tool, `EBat`. Despite the kind assistance of `BooM`'s authors, we could not get the original implementation to work, making `EBat` the only publicly available all-Boolean-matching tool.

Experiments show that our algorithms solve 3 to 4 times more benchmarks than the baseline `BooMS` algorithm within `EBat` for both P- and NP-equivalence across a diverse benchmark set.

In what follows, Sect. 2 provides preliminaries, and Sect. 3 reviews prior work. Sect. 4 introduces the `EBat` algorithms, with mismatch handling for P-equivalence detailed in Sect. 5, and correctness discussed in Sect. 6. Experimental results are presented in Sect. 7, and conclusions in Sect. 8.

2 Preliminaries

We establish the relevant notation, assuming familiarity with Boolean logic fundamentals. Let V be a set of Boolean variables. A *literal* l is either a variable $v \in V$ or its negation $\neg v$. The set of literals corresponding to all variables in V is denoted by $\nu^V := \{v \mid v \in V\} \cup \{\neg v \mid v \in V\}$. For a single variable v, we overload the notation by letting ν^v denote either v or $\neg v$, non-deterministically. Given a function F, $Dom(F)$ denotes its domain.

We introduce the semantics used in this paper, omitting the standard Boolean semantics for brevity. *Ternary logic* [34] extends Boolean logic by introducing a third value, *don't-care* (X). Formally, a *ternary assignment* $\tau : V \mapsto \{0, 1, X\}$ assigns each variable one of the *ternary values* $\{0, 1, X\}$. Hereafter, the term *assignment* will refer to a ternary assignment unless otherwise specified, and every assignment is total (i.e., it is defined for all variables in its domain). We omit variables assigned X when listing assignments. For example, $\tau(\{v_1, v_2\}) \equiv \{v_1 := 1\}$ represents $\tau(\{v_1, v_2\}) \equiv \{v_1 := 1, v_2 := X\}$. The *cardinality* $|\tau|$ is the number of variables in τ assigned either 0 or 1. An assignment is *Boolean* if it has maximal cardinality. To evaluate a formula under an assignment τ, standard Boolean semantics is extended with $(\neg X \equiv X)$, $(X \wedge 1 \equiv X)$, $(X \wedge 0 \equiv 0)$, and $(X \wedge X \equiv X)$. An assignment $\rho(V)$ *subsumes* the assignment $\tau(V)$, denoted by $\rho \subseteq \tau$, if $\tau(v) = \rho(v)$ for every v such that $\rho(v) \in \{0, 1\}$. If $\rho(V)$ subsumes $\tau(V)$, then $\tau(V)$ *extends* $\rho(V)$. For example, $\tau_1 \equiv \{v_1 := 1\}$ subsumes $\tau_2 \equiv \{v_1 := 1, v_2 := 0\}$, whereas τ_2 extends τ_1. We now proceed to define a circuit.

▶ **Definition 1** (Circuit). *A combinational Boolean* circuit *with i inputs I^Y, g gates G^Y, and output o^Y is the Boolean structure:*

$$\overset{g}{\underset{i}{Y}}(I^Y = \{y_1^Y, \ldots, y_i^Y\}) = \langle G^Y = \{y_{i+1}^Y \leftrightarrow g_{i+1}^Y, \ldots, y_{i+g}^Y \leftrightarrow g_{i+g}^Y\}, o^Y \in \nu^{\{y_{i+g}^Y\}} \rangle$$

Each input $y_p^Y \in I^Y$ is a Boolean variable. For each gate, g_k^Y is of the form:

$$g_k^Y = \left(\nu^{y_{k_1}^Y} \bigcirc_k \nu^{y_{k_2}^Y}\right) | 1 \leq k_1, k_2 < k$$

with $\nu^{y_{k_1}^Y}$ and $\nu^{y_{k_2}^Y}$ representing the (possibly negated) inputs to the gate and $\bigcirc_k$ being a Boolean operator (e.g., $\wedge$, $\vee$, $=$, $\oplus$).

For example, Fig. 1a illustrates the circuit $\overset{2}{\underset{3}{Y}}(I = \{y_1, y_2, y_3\}) = \langle G = \{y_4 \leftrightarrow y_1 \wedge y_2, y_5 \leftrightarrow y_4 \wedge \neg y_3\}, o \equiv \neg y_5\}\rangle$. Towards extending the semantics to circuits, we define ternary simulation [37, 24]. Intuitively, ternary simulation propagates an input assignment τ from the circuit inputs through its gates to the output.

▶ **Definition 2** (Ternary Simulation). *Given a circuit* $\overset{g}{\underset{i}{Y}}$ *with inputs* $I^Y = \{y_1^Y, \ldots, y_i^Y\}$, *gates* $G^Y = \{y_{i+1}^Y \leftrightarrow g_{i+1}^Y, \ldots, y_{i+g}^Y \leftrightarrow g_{i+g}^Y\}$, *output* $o^Y \in \nu^{\{y_{i+g}^Y\}}$, *and an assignment* $\tau(I^Y) : I^Y \mapsto \{0, 1, X\}$ *to the circuit's inputs,* ternary simulation *transforms* τ *into the assignment* $\tau^S(\{y_1^Y, \ldots, y_{i+g}^Y\})$, *where:*

1. $\tau^S(y_p^Y) := \tau(y_p^Y)$ *for each input* $y_p^Y \in I^Y$.
2. *For each gate* $y_k^Y \leftrightarrow g_k^Y$, *where* $g_k^Y = \left(\nu^{y_{k_1}^Y} \bigcirc_k \nu^{y_{k_2}^Y}\right)$:

$$\tau^S(y_k^Y) := \tau^S(\nu^{y_{k_1}^Y}) \bigcirc_k \tau^S(\nu^{y_{k_2}^Y})$$

For the example circuit Y in Fig. 1a, ternary propagation would propagate $\tau(I \equiv \{y_1, y_2, y_3\}) \equiv \{y_1 := 0\}$ to $\tau^S(\{y_1, \ldots, y_5\}) \equiv \{y_1 := 0, y_4 := 0, y_5 := 0\}$.

We are now prepared to define what it means for an input assignment to serve as a circuit solution. Our definition relies on *entailment* [38] following the most general of the three formulations of a circuit solution we presented in [21]. Intuitively, $\tau(I)$ is a solution if extending $\tau(I)$ to *any* Boolean assignment and propagating by ternary simulation always results in the circuit outputting 1.

▶ **Definition 3** (Entailment, Solution). *Given a circuit* $\overset{g}{\underset{i}{Y}}$ *with inputs* $I^Y = \{y_1^Y, \ldots, y_i^Y\}$, *gates* $G^Y = \{y_{i+1}^Y \leftrightarrow g_{i+1}^Y, \ldots, y_{i+g}^Y \leftrightarrow g_{i+g}^Y\}$, *output* $o^Y \in \nu^{\{y_{i+g}^Y\}}$, *and an assignment* $\tau(I^Y) : I^Y \mapsto \{0, 1, X\}$ *to the circuit's inputs,* τ entails Y *(denoted by* $\tau \models Y$*), if* $\rho^S(o) = 1$ *for* any ρ *which substitutes* every X *in* τ *by* any *Boolean value. Furthermore,* τ *is a* solution *to* Y *if and only if* $\tau \models Y$.

For example, in the circuit Y shown in Fig. 1a, $\tau_1(I) \equiv \{y_1 := 0\} \models Y$, as any extension followed by propagation of τ_1 yields $o \equiv \neg y_5 := 1$. Conversely, $\tau_2(I) \equiv \{y_1 := 1\} \not\models Y$, since extending τ_2 to $\{y_1 := 1, y_2 := 1, y_3 := 0\}$ and propagating results in $o \equiv \neg y_5 := 0$. We proceed with mappings-related definitions.

▶ **Definition 4** (Mapping, Permutation). *Let* $I^Y = \{y_1, \ldots, y_i\}$ *and* $I^Z = \{z_1, \ldots, z_i\}$ *be ordered sets of* i *Boolean variables each. A* mapping π *is a function* $\pi : I^Y \to \nu^{I^Z}$ *injective w.r.t. variables, i.e., if* $y_p \in Dom(\pi)$ *is mapped to either* z_q *or* $\neg z_q$, *then no other element* $y_r \in Dom(\pi)$, *where* $r \neq p$, *can be mapped to* z_q *or* $\neg z_q$.

A mapping π *is* total *if* $Dom(\pi) = I^Y$, *whereas any mapping, including total mappings, is considered* partial *(i.e., the term mapping refers to a partial mapping unless specified otherwise). A mapping* π *is a* permutation *if its range contains only non-negated variables, i.e.,* $\pi(y) \in I^Z$ *for all* $y \in Dom(\pi)$.

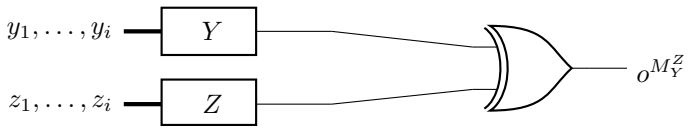
Given $I^Y = \{y_1, y_2, y_3\}$ and $I^Z = \{z_1, z_2, z_3\}$, two example (total) mappings are: $\rho_1 = [\neg z_3; z_2; z_1]$ and $\rho_2 = [z_1; z_3; z_2]$, where we represent the mapping $[y_1 \mapsto \nu^{z_{q_1}}; \ldots; y_n \mapsto \nu^{z_{q_n}}]$ by the shortened notation $[\nu^{z_{q_1}}, \ldots, \nu^{z_{q_n}}]$. Here, ρ_1 is *not* a permutation because its range includes a negated variable, whereas ρ_2 is. When representing partial mappings, we use the bullet sign $\bullet$ for any unmapped elements. For example, $\rho_3 = [y_1 \mapsto \neg z_3; y_3 \mapsto z_1]$ can be written as $[\neg z_3; \bullet; z_1]$.

We next extend the semantics to mappings. A (total ternary) assignment τ satisfies a (partial) mapping π if τ can be extended to an assignment ρ that assigns Boolean values consistently with π to all the variables in Dom(π).

▶ **Definition 5** (Satisfy a Mapping). *Given the ordered sets* $I^Y = \{y_1, \ldots, y_i\}$ *and* $I^Z = \{z_1, \ldots, z_i\}$, *a mapping* $\pi : I^Y \to \nu^{I^Z}$, *and an assignment* $\tau(I^Y \cup I^Z)$, *we say that* τ satisfies π *(denoted by* $\tau \lessapprox \pi$*) if there exists* ρ *such that* $\rho \supseteq \tau$ *and:*

$$\forall y_p \in Dom(\pi) : \rho(y_p) \in \{0, 1\} \text{ and } \rho(y_p) = \rho(\pi(y_p)).$$

Consider, for example, the mapping $\pi \equiv [z_2; \bullet; \bullet]$ from $I^Y = \{y_1, y_2, y_3\}$ to $I^Z = \{z_1, z_2, z_3\}$. Any assignment τ that does not assign Boolean values to both y_1 and z_2, in a way inconsistent with π, satisfies π. For example, $\{y_1 := 0; z_2 := 1\}$ does *not* satisfy π, but $\{y_1 := 0\}$ does (since $\{y_1 := 0\}$ can be extended to $\{y_1 := 0; z_2 := 0\}$). We next define the *miter* [11], a circuit that combines two given circuits by XORing their outputs as follows:



▶ **Definition 6** (Miter). *Given two circuits* $\overset{g}{\underset{i}{Y}}(I^Y)$ *and* $\overset{g'}{\underset{i}{Z}}(I^Y)$*, their miter* M_Y^Z *is the circuit* $\overset{g+g'+1}{\underset{2i}{M_Y^Z}}(\{y_1, \ldots, y_i, z_1, \ldots, z_i\}) = \langle \{y_{i+1}, \cdots y_{i+g}, z_{i+1}, \cdots, z_{i+g'}, y_{i+g} \oplus z_{i+g'}\}, y_{i+g} \oplus z_{i+g'} \} \rangle$.

The miter is used to determine if a mapping π is a match or a mismatch: π is a mismatch iff there is a witness – an assignment to the miter inputs – that entails the miter and satisfies π; otherwise, π is a match. Below, we formalize the related definitions for both P- and NP-equivalence, with the only difference being the restriction of the examined mappings to permutations for P-equivalence.

▶ **Definition 7** (Mismatch, (Mismatch) Witness, Match). *Given circuits* $\overset{g}{\underset{i}{Y}}(I^Y)$ *and* $\overset{g'}{\underset{i}{Z}}(I^Z)$ *and a mapping* $\pi : I^Y \to R$*, where* $R \equiv \nu^{I^Z}$ *for NP- and* $R \equiv I^Z$ *for P-equivalence,* π *is a* mismatch *between* Y *and* Z *iff there exists an assignment* $\sigma(I^Y \cup I^Z)$*, called a* (mismatch) witness *for* π*, such that:*

1. $\sigma \models M_Y^Z$*, and*
2. $\sigma \not\approx \pi$*.*

Furthermore, any mapping $\pi : I^Y \to R$ *that is not a mismatch between* Y *and* Z *is a* match *between them.*

Given the two circuits in Fig. 1, the assignment $\tau(I^Y \cup I^Z) \equiv \{y_1 := 1, y_2 := 1; y_3 := 0; z_3 := 0\}$ is a mismatch witness for the identity mapping $\pi \equiv [z_1, z_2, z_3]$. First, propagating τ causes the outputs of Y and Z to differ, resulting in M_Y^Z outputting 1; thus, τ entails the miter. Second, τ satisfies π, as it can be extended consistently with π by assigning 1 to both z_1 and z_2. Conversely, the mapping $\rho = [z_3, z_1, z_2]$ is a match, as no assignment satisfies ρ and entails the miter.

To enable testing whether a mapping is a match using a SAT solver, we introduce the concepts of a circuit formula and a mapping formula.

▶ **Definition 8** (Circuit Formula). *Given a circuit* $\overset{g}{\underset{i}{Y}}$ *with inputs* $I^Y = \{y_1^Y, \ldots, y_i^Y\}$*, gates* $G^Y = \{y_{i+1}^Y \leftrightarrow g_{i+1}^Y, \ldots, y_{i+g}^Y \leftrightarrow g_{i+g}^Y\}$*, output* $o^Y \in \nu^{\{y_{i+g}^Y\}}$*, its Boolean* circuit formula f^Y *is:*

$$f^Y \equiv o^Y \wedge \bigwedge_{k=i+1}^{i+g} \left(y_k^Y \leftrightarrow g_k^Y\right).$$

▶ **Definition 9** (Mapping Formula). *Given the ordered sets* $I^Y = \{y_1, \ldots, y_i\}$ *and* $I^Z = \{z_1, \ldots, z_i\}$ *and a mapping* $\pi : I^Y \to \nu^{I^Z}$*,* π*'s Boolean* mapping formula f^π *is:*

$$f^\pi \equiv \bigwedge_{p=1}^{i} (y_p \leftrightarrow \pi(y_p)).$$

The following lemma is central to the algorithms in this paper and the underlying works [26, 25]. It can be easily verified.

▶ **Lemma 10.** *Given two circuits Y_i^g and $Z_i^{g'}$, a mapping π between their inputs is a match if and only if $f^{M_Y^Z} \wedge f^{\pi}$ is unsatisfiable.*

We need to define subsumption for mappings. Intuitively, π_1 subsumes π_2 if π_2 agrees with π_1 on the entire domain of π_1.

▶ **Definition 11** (Subsumption for Mappings). *Given the ordered sets I^Y and I^Z of the same cardinality, let $\pi_1, \pi_2 : I^Y \to \nu^{I^Z}$ be two mappings. We say that π_1* subsumes *π_2, denoted $\pi_1 \subseteq \pi_2$, if for every $y_p \in Dom(\pi_1)$, we have $y_p \in Dom(\pi_2)$ and $\pi_1(y_p) = \pi_2(y_p)$.*

For example, $\pi_1 = [\bullet, \neg z_3; \neg z_2]$ subsumes $\pi_2 = [z_1, \neg z_3; \neg z_2]$. We are now ready to define all-Boolean-matching for both NP- and P-equivalence.

▶ **Definition 12** (All-Boolean-Matching). *Given circuits $\underset{i}{\overset{g}{Y}}(I^Y)$ and $\underset{i}{\overset{g'}{Z}}(I^Z)$, and assuming $R \equiv \nu^{I^Z}$ for NP- and $R \equiv I^Z$ for P-equivalence, an* all-Boolean-matching *algorithm reports a set of mappings $S \subseteq (I^Y \to R)$ such that:*

1. *Every $\pi \in S$ is a match between Y and Z, and*
2. *Any $\rho : I^Y \to R$ not subsumed by a $\pi \in S$ is a mismatch between Y and Z.*

Notably, we allow a total match to be subsumed by more than one of the reported matches. In other words, the reported matches need not be disjoint.

We conclude with a brief review of relevant SAT-related concepts and generalization.

A *cardinality constraint* is a Boolean constraint that ensures that at-most, at-least or exactly k literals hold in a literal set. A *clause* is a disjunction (set) of literals. A *cube* is a conjunction (set) of literals. A formula F is in *Conjunctive Normal Form (CNF)* if it is a conjunction (set) of clauses. Given a CNF formula F, a SAT solver decides whether F is satisfiable. Given a satisfiable formula, a SAT solver also returns its *total Boolean solution (solution)*. Many SAT solvers are *incremental* [20, 33]: they can be invoked multiple times, where, for every new query $\text{SAT}(F, A)$, 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, $\text{SAT}(F, A)$ returns an *Unsatisfiable Core (UC)*, that is, a cube $A' \subseteq A$, such that $F \wedge A'$ is still unsatisfiable [20].

Given a circuit T and its Boolean solution $\sigma(I^T)$ (that is, σ entails T), *generalization* [35, 27, 39, 21] transforms (or *generalizes*) σ into a smaller ternary solution σ' such that $\sigma' \subseteq \sigma$ by replacing as many Boolean values as possible with X's, while ensuring that σ' still entails T. One commonly used generalization approach is the ternary-simulation-based *Forward Ternary Simulation (`FTS`)* [37, 19], which iteratively attempts to replace each input's Boolean value with the don't-care value X, using ternary simulation (Def. 2) to check whether the circuit remains satisfied. Another common ternary-simulation-based approach is *Backward Ternary Simulation (`BTS`)* [37, 39], also known as *justification*. This approach checks which internal gates and, ultimately, inputs can be assigned X while still satisfying the circuit, proceeding backward from outputs to inputs. In [21], we demonstrated that *Minimal Unsatisfiable Core (`MUC`)*-based generalization [14], which leverages properties of the so-called *dual circuit* (i.e., the original circuit with its output negated), is theoretically more powerful and empirically more efficient than both `BTS` and `FTS`. In fact, the best results in the context of enumerating circuit solutions were achieved by combining `FTS` and `MUC` (denoted `FTS&MUC` in this work and as `ROC` in [21]).

3 Previous Work: BooM

In this section, we present BooM's all-Boolean-matching flow [26, 25], including its SAT encoding in Sect. 3.1 and the sifter (BooMS) algorithm in Sect. 3.2.

$$\left(\begin{array}{c|cccc} & z_1 & z_2 & \dots & z_i \\ \hline y_1 & \{x^+_{11}, x^-_{11}\} & \{x^+_{12}, x^-_{12}\} & \dots & \{x^+_{1i}, x^-_{1i}\} \\ y_2 & \{x^+_{21}, x^-_{21}\} & \{x^+_{22}, x^-_{22}\} & \dots & \{x^+_{2i}, x^-_{2i}\} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y_i & \{x^+_{i1}, x^-_{i1}\} & \{x^+_{i2}, x^-_{i2}\} & \dots & \{x^+_{ii}, x^-_{ii}\} \end{array}\right)$$

(a) Indicator variables $D = \{x^+_{pq}, x^-_{pq} \mid 1 \le p, q \le i\}$ to represent mappings between $I^Y = \{y_1, \dots, y_i\}$ and $I^Z = \{z_1, \dots, z_i\}$, where x^+_{pq} indicates if $y_p \mapsto z_q$, and x^-_{pq} indicates if $y_p \mapsto \neg z_q$.

$$\left(\forall p \in [1, \dots, i] : \left(\textstyle\sum_{q=1}^{i}(x^+_{pq} + x^-_{pq})\right) = 1\right) \bigwedge \left(\forall q \in [1, \dots, i] : \left(\textstyle\sum_{p=1}^{i}(x^+_{pq} + x^-_{pq})\right) = 1\right)$$

(b) The *map-validity* constraint (comprising a conjunction of two cardinality constraints) ensures that each $y_p \in I^Y$ maps to exactly one z_q or $\neg z_q$, where no other $y_r \in I^Y | r \neq p$ maps to z_q or $\neg z_q$.

$$\bigwedge_{1 \le p,q \le i} (x^+_{pq} \to (y_p \leftrightarrow z_q)) \wedge (x^-_{pq} \to (y_p \leftrightarrow \neg z_q))$$

(c) *Map-to-inputs* constraint: x^+_{pq} implies $y_p \leftrightarrow z_q$, and x^-_{pq} implies $y_p \leftrightarrow \neg z_q$.

Figure 2 BooM's SAT encoding: indicator variables and related constraints.

3.1 SAT Encoding

BooM maintains two SAT instances: `bucket` and `misms`. Both include a Boolean variable for each input of both circuits: $I^Y = \{y_1, \dots, y_i\}$ and $I^Z = \{z_1, \dots, z_i\}$. To reason about mappings between I^Y and I^Z, both instances contain the indicator variables $D = \{x^+_{pq}, x^-_{pq} \mid 1 \le p, q \le i\}$, which represent mappings between $I^Y = \{y_1, \dots, y_i\}$ and $I^Z = \{z_1, \dots, z_i\}$, where x^+_{pq} indicates if $y_p \mapsto z_q$, and x^-_{pq} indicates if $y_p \mapsto \neg z_q$. See Fig. 2a for an illustration. Additionally, we will use the notation x^s_{pq} to denote either x^+_{pq} or x^-_{pq}.

For P-equivalence, all x^-_{pq} variables are fixed to 0. This is the only adjustment needed to adapt the SAT encoding from the default NP-equivalence to P-equivalence.

The first SAT instance, `bucket`, is initialized with the map-validity constraint shown in Fig. 2b, ensuring that it initially represents *all possible mappings*.

The second SAT instance, `misms`, is initialized with *only mismatches*. To achieve this, in addition to the map-validity constraint, it is also initialized with the map-to-inputs constraint shown in Fig. 2c, and the miter formula $f^{M^Z_Y}$ – representing the miter circuit translated to CNF. Each solution to `misms` corresponds to a total mapping π due to the map-validity constraint, but it is restricted to mismatches because every solution must assign input values consistently with the mapping (enforced by the map-to-inputs constraint) while satisfying the miter formula $f^{M^Z_Y}$, ensuring a mismatch.

In the presentation below, we assume that circuits and cardinality constraints are implicitly translated into clauses. Our implementation uses Tseitin encoding [41] for circuit translation and nested encoding [7] for cardinality constraints.

3.2 `BooMS` Algorithm

We present the `BooMS` algorithm in Alg. 2 (please recall Sect. 1.3 for its high-level flow). Differently from the original presentation [26, 25], we isolated the SiftMis subroutine in Alg. 1 to enable SiftMis's integration into our novel `EBatC` algorithm (Sect. 4.2). As shown in Alg. 2, `BooMS` begins by invoking SiftMis.

SiftMis, shown in Alg. 1, takes circuits Y and Z and returns the SAT instance `bucket` containing all total *matches* between them as its solutions. As explained in Sect. 3.1, `bucket` is initialized with all possible mappings (line 1), while SiftMis also maintains another SAT instance `misms`, initialized to capture only mismatches (line 2).

SiftMis proceeds with a while loop at line 3, iterating over all mismatches by querying `misms` until no more are found (i.e., when `misms` returns UNSAT). In each iteration, the algorithm queries `misms` for a new mismatch, extends the witness to cover additional mismatches, and blocks them in *both* `misms` and `bucket`, as detailed in Sect. 3.2.1. Once no more mismatches remain, the algorithm returns `bucket`.

Going back to `BooMS` (Alg. 2), after initializing `matches` with the matches by invoking SiftMis, `BooMS` iteratively reports and blocks the matches.

3.2.1 Mismatch Handling in `BooMS`

We begin by showing the clause added by `BooMS` to block mismatches (when BlkMis is applied at lines 5 and 6 of SiftMis in Alg. 1), distinguishing between NP- and P-equivalence.

For NP-equivalence, the following clause C^N_σ is added to both `bucket` and `misms`:

$$C^N_\sigma := \bigvee_{1\le p,q\le i} \begin{cases} x^+_{pq} & \text{if } \sigma(y_p) \neq \sigma(z_q) \text{ and } \sigma(y_p), \sigma(z_q) \in \{0,1\}, \\ x^-_{pq} & \text{if } \sigma(y_p) = \sigma(z_q) \text{ and } \sigma(y_p), \sigma(z_q) \in \{0,1\}. \end{cases}$$

Adding C^N_σ ensures that every mismatch π satisfied by σ is blocked: for each such π, C^N_σ forces at least one Boolean-assigned y_p to map to $\neg\pi(y_p)$. Furthermore, no mappings unsatisfied by σ are blocked. Indeed, for any such π, there must exist a y_p such that $\sigma(y_p) \neq \sigma(\pi(y_p))$ (otherwise, π would have been satisfied by σ), ensuring that C^N_σ is satisfied by either x^+_{pq} or x^-_{pq}, assuming $\pi(y_p) = z_q$.

For P-equivalence, the blocking clause is as follows:

$$C^P_\sigma := \bigvee x^+_{pq} \quad \text{for all } p, q \text{ such that } \sigma(y_p \in I^Y) = 0 \text{ and } \sigma(z_q \in I^Z) = 1$$

This enforces one of the Y inputs to be mapped to a Z input currently assigned a different value, thereby blocking all satisfied mismatches – and only them, since, by construction, C^P_σ includes a satisfied indicator for any mapping unsatisfied by σ, similar to NP-equivalence.

We introduce another notation: Given an assignment σ to $I^Y \cup I^Z$, we call an indicator $x^s_{pq} \in D$ (recall that $s \in \{-,+\}$) an *X-indicator* if either y_p, z_q, or both are assigned X in σ.

Only for NP-equivalence, before blocking, `BooMS` extends the witness through generalization (via `BTS`): it replaces Boolean values assigned to inputs in σ with X's, while still satisfying the miter. Notably, any X-indicators are then dropped from the blocking clause, thereby blocking more mismatches at once. This is valid because extending the generalized witness to *any* Boolean assignment (by replacing all X's with Boolean values) results in a Boolean

mismatch witness, with our clause blocking all mismatches satisfied by these witnesses at once (as if a blocking clause were added for each such Boolean mismatch). In contrast, for *P-equivalence*, generalizing and dropping X-indicators from the blocking clause is incorrect (see Sect. 5) likely explaining why `BooM` does not extend witnesses for P-equivalence.

Algorithm 1 SiftMis.

Input: Circuits $\overset{g}{Y}_i$ and $\overset{g'}{Z}_i$
Output: The SAT instance `bucket` with only the matches remaining

1: Initialize the `bucket` SAT instance with all mappings ▷ by map-validity (Fig. 2b)
2: Initialize the `misms` SAT instance ▷ by map-validity, map-to-inputs, $f^{M^Z_Y}$
3: **while** $\sigma :=$ SAT(`misms`) is satisfiable **do**
4: $\quad\sigma' :=$ ExtWit(M^Z_Y, σ) ▷ Extend the witness σ to satisfy more mismatches
5: $\quad$BlkMis(`misms`, σ') ▷ Block the satisfied mismatches in `misms`
6: $\quad$BlkMis(`bucket`, σ') ▷ Block the satisfied mismatches in `bucket`
7: **return** `bucket`

Algorithm 2 `BooMS`.

Input: Circuits $\overset{g}{Y}_i$ and $\overset{g'}{Z}_i$
Output: All the matches between Y and Z

1: `matches` := SiftMis(Y, Z)
2: **while** $\pi :=$ SAT(`matches`) is satisfiable **do**
3: $\quad$Report π to the user
4: $\quad$Block π in `matches`, with a clause containing $\neg x^s_{pq}$ for every *satisfied* $x^s_{pq} \in D$

4 All-Boolean-Matching Algorithms in `EBat`

We introduce our high-level algorithms `EBatP` (Sect. 4.1) and `EBatC` (Sect. 4.2).

4.1 The Picker `EBatP`

`EBatP` implements a straightforward *picker* classification algorithm: given a bucket of total mappings, the picker iteratively removes a mapping π, reporting it if it is a match. The key to `EBatP`'s efficiency lies in classifying and removing multiple total mappings simultaneously for *both* mismatches and matches – unlike the previous state-of-the-art algorithm `BooMS` (Sect. 3.2), which did so only for mismatches. Moreover, `EBatP` handles mismatches significantly more efficiently, especially for P-equivalence (more on this later).

`EBatP` is presented in Alg. 3. In addition to the input circuits, `EBatP` can optionally accept a pre-initialized SAT instance, `bucket`, from the user. For the remainder of Sect 4.1, assume that `bucket` is *not* provided.

`EBatP` maintains two SAT instances: `bucket` and `classifier`. `bucket`, initialized at line 2 with map-validity constraint in Fig. 2b, maintains unclassified total mappings as its solutions (similarly to `bucket` in `BooMS`). `classifier`, initialized at line 3 with the miter formula $f^{M^Z_Y}$ and the map-to-inputs constraint shown in Fig. 2c, is used to classify a mapping as either a match or a mismatch.

Algorithm 3 `EBatP`.

Input: Circuits $\overset{g}{Y}_i$ and $\overset{g'}{Z}_i$, and, optionally, the SAT instance `bucket`
Output: All the matches between Y and Z

1: **if** `bucket` is not provided by the user **then**
2: Initialize the `bucket` SAT instance ▷ by map-validity (Fig. 2b)
3: Initialize the `classifier` SAT instance ▷ by $f^{M^Z_Y}$ and map-to-inputs (Fig. 2c)
4: **while** $\pi :=$ SAT(`bucket`) is satisfiable **do**
5: $\sigma :=$ SAT(`classifier`, Q^π) ▷ Is π a match?
6: **if** UNSAT **then** ▷ π is a match
7: $\pi' :=$ StrenMatch(`classifier`, π) ▷ Strengthen π by minimizing the UC
8: Report the match π' to the user ▷ Report the strengthened match π
9: BlkMatch(`bucket`, π') ▷ Block π' by adding $\neg Q^{\pi'}$ to `bucket`
10: **else** ▷ π is a mismatch
11: $\sigma' :=$ ExtWit(M^Z_Y, σ) ▷ Extend the witness σ to satisfy more mismatches
12: BlkMis(`bucket`, σ') ▷ Block the satisfied mismatches

$$Q^\pi \equiv \bigwedge_{p:y_p \in \mathrm{Dom}(\pi)} \begin{cases} x^+_{pq} & \text{if } \pi(y_p) = z_q \\ x^-_{pq} & \text{if } \pi(y_p) = \neg z_q \end{cases}$$

(a) *π-cube* Q^π: assuming Q^π triggers π (i.e, f^π), given the map-to-inputs constraint.

$$\pi^Q(y_p) = \begin{cases} z_q, \text{ if } x^+_{pq} \in Q \\ \neg z_q, \text{ if } x^-_{pq} \in Q \\ \text{unmapped (i.e., } y_p \notin \mathrm{Dom}(\pi^Q)\text{), if } x^+_{pq} \notin Q \text{ and } x^-_{pq} \notin Q \end{cases}$$

(b) Extracting the mapping π^Q from a π-cube Q.

Figure 3 Building the cube Q^π to represent a mapping π and extracting a mapping from a cube.

4.1.1 The Main Loop

After initializing both SAT instances, the algorithm enters the while loop at line 4, iterating over all mappings until no further mappings are found (i.e., when `bucket` returns UNSAT), at which point it terminates.

Each iteration of the algorithm queries `bucket` for an unclassified total mapping π, classifies as either a match or a mismatch via `classifier`, transforms π into a set Γ where every $\rho \in \Gamma$ is a match iff π is, and blocks Γ in `bucket`, reporting matches.

Going back to Alg. 3, assume `bucket` returns a non-classified total mapping π at line 4. By Lemma 10, π is a match iff the conjunction of the miter formula $f^{M^Z_Y}$ (held by `classifier`) and the mapping formula $f^\pi \equiv \bigwedge_{p=1}^{i}(y_p \leftrightarrow \pi(y_p))$ is unsatisfiable. The algorithm tests whether π is a match in `classifier` at line 5 by invoking `classifier` under the assumption cube Q^π in Fig. 3a that enforces f^π for the current `classifier` query.

4.1.2 The Match Case

If `classifier` returns UNSAT, π is classified as a match. The algorithm invokes the function StrenMatch to *strengthen* π to a smaller partial match π' such that $\pi' \subseteq \pi$, based on the unsatisfiable core $U \subseteq Q^{\pi}$, obtained from `classifier`. The UC U induces the partial mapping π^U (see Fig. 3b), which must be a match (otherwise, U would not have been a UC because `classifier` $\wedge\, U$ would have been satisfiable by a mismatch witness for π^U). Since $U \subseteq Q^{\pi}$, it follows that $\pi^U \subseteq \pi$. Instead of simply returning π^U, StrenMatch, shown below, attempts to derive an even smaller match by iteratively minimizing U [18]:

Function StrenMatch(`classifier`, π):
  $U :=$ latest unsatisfiable core from `classifier` ▷ $\pi^U \subseteq \pi$
  forall $a \in U$: **if** SAT(`classifier`, $U \setminus \{a\}$) is UNSAT **then** $U := U \setminus \{a\}$
  return π^U ▷ Recall Fig. 3b

Finally, Alg. 3 invokes the function BlkMatch, shown below, to block the match π', returned by StrenMatch, by adding the clause $\neg Q^{\pi'}$ to `bucket`:

Function BlkMatch(`bucket`, π'):
  AddClause(`bucket`, $\neg Q^{\pi'}$)

4.1.3 The Mismatch Case

If `classifier` returns SAT with a witness σ, then π is a mismatch, satisfied by σ. The algorithm invokes ExtWit, which extends σ to σ', aiming to satisfy additional mismatches while still satisfying π, then calls BlkMis to block all mismatches satisfied by σ'.

For NP-equivalence, our mismatch handling is similar to `BooMS` (Sect. 3.2.1), with the notable exception of the generalization algorithm for extending mismatches. (Recall our brief review of generalization algorithms from the last paragraph of Sect. 2.) While `BooMS` applied `BTS`, and our previous work on enumeration in circuit SAT found that the `FTS&MUC` combination works best in that context [21], we show that the `BTS&MUC` combination outperforms both `BTS` and `FTS&MUC` for all-Boolean-matching (Sect. 7).

Our mismatch handling algorithm for P-equivalence is presented in Sect. 5.

4.2 The Combined `EBatC`

Alg. 4 introduces our combined sift-and-pick `EBatC` algorithm. Our design of `EBatP` and `BooMS` laid the foundation for expressing `EBatC` concisely.

Algorithm 4 `EBatC`.

Input: Circuits $\overset{g}{Y}_i$ and $\overset{g'}{Z}_i$
Output: All the matches between Y and Z

1: `matches` := SiftMis(Y, Z) ▷ See Alg. 1
2: **return** `EBatP`(Y, Z, `matches`) ▷ See Alg. 3

`EBatC` first generates the SAT instance `matches` with all total matches using SiftMis (as in `BooMS`). It then enumerates matches via `EBatP` with `matches` as input. Since `matches` contains only matches, `classifier` queries in `EBatP` always return UNSAT. However, these queries remain essential, as strengthening relies on the UC returned by `classifier`.

The key difference between EBatC and our picker algorithm EBatP is that EBatC processes mismatches first, then matches, while EBatP iterates over all mappings. We expected EBatC to perform better since incremental SAT solvers typically handle similar consecutive queries more efficiently. Indeed, Sect. 7 empirically confirms that querying mismatches first (EBatC) is more effective than mixing match and mismatch queries without prior knowledge (EBatP).

4.2.1 The Trade-off between EBatP and EBatC

While EBatC outperforms EBatP in our experiments (Sect. 7), EBatP has an inherent advantage: it is an **anytime** algorithm. Unlike EBatC, which must process all mismatches before enumerating any matches, EBatP begins generating solutions immediately. This property is valuable for difficult instances, where EBatC may stall in its initial phase and yield no matches. In such cases, EBatP can be used to return as many matches as possible to the user.

5 Mismatch Handling for P-Equivalence in EBat

Contrary to NP-equivalence, for P-equivalence, applying generalization to extend the mismatch witness, and dropping X-indicators from the blocking clause (recall Sect. 3.2.1), is illegal – it might block valid matches, as demonstrated by the following example.

Recall Fig. 1. Please keep in mind that, in both valid matches, y_3 maps to z_2. Consider the mapping $\pi = [z_2, z_3, z_1]$. π is a mismatch with the witness $\sigma \equiv \{y_1 := 1, y_2 := 1, y_3 := 0, z_1 := 0, z_2 := 1, z_3 := 1\}$ (as σ satisfies π and propagating σ in Y and Z results in the outputs of the two circuits being assigned different values, thus satisfying the miter). The corresponding blocking clause in BooMS is $C^P_\sigma \equiv (x^+_{32} \vee x^+_{33})$ (recall Sect. 3.2.1). Generalization could extend σ by substituting z_2's value with X to $\sigma' \equiv \{y_1 := 1, y_2 := 1, y_3 := 0, z_1 := 0, z_2 := X, z_3 := 1\}$, where σ' is still a mismatch witness. If we allowed dropping X-indicators from C^P_σ, the resulting clause would be $C^P_{\sigma'} \equiv (x^+_{33})$. However, that would force y_3 to always be mapped to z_3, erroneously blocking both valid matches where y_3 maps to z_2.

The root cause of this limitation is that, for P-equivalence, extending the generalized witness to a Boolean assignment does *not* necessarily result in a mismatch witness as it might violate the following *0-1 balance* property, unique to P-equivalence: to serve as a mismatch witness, a Boolean assignment σ must assign an equal number of b's to inputs in I^Y and I^Z for both $b \in \{0, 1\}$. This is because, otherwise, σ cannot satisfy any mapping, since, under P-equivalence, every input $y_p \in Y$ assigned $b \in \{0, 1\}$ must be mapped to some input $z_q \in Z$ that is also assigned the very same b.

Intuitively, dropping X-indicators from C^P_σ effectively introduces a blocking clause C^P_ρ for a non-witness assignment ρ that violates 0-1 balance. However, adding such clauses – i.e., BooM's blocking clauses based on non-witnesses – is incorrect.

5.1 Witness-Extension for P-Equivalence

Our novel witness-extension procedure for P-equivalence relies on Boolean logic rather than ternary logic and generalization. It aims to maximize the *merit* of the witness:

Let σ be a Boolean mismatch witness. Let $f_\sigma \in \{0, 1\}$ be the more frequent value assigned by σ to I^Y (or, equivalently, I^Z because of 0-1 balance), and $l_\sigma \neq f_\sigma$ be the less frequent value, assuming $f_\sigma = 0$ in case of a tie. The *merit* M_σ of σ is the count of variables in I^Y assigned f_σ. Observe that the higher the merit, the more total mismatches σ satisfies.

Indeed, for P-equivalence, a Boolean witness σ satisfies $s(\sigma) \equiv M_\sigma! \times (i - M_\sigma)!$ total mismatches. The maximum occurs when $M_\sigma = i$ (i.e., all inputs are assigned the same value, satisfying all $i!$ permutations), while the minimum is reached when M_σ is minimized

at $\lceil \frac{i}{2} \rceil$. As M_σ increases, $s(\sigma)$ – and consequently the number of satisfied total mismatches – also increases, confirming that a higher merit leads to more satisfied total mismatches, as intended.

For clarification of the latest notions, consider a high-level example with two circuits, each having three inputs. The assignment $\rho \equiv \{y_1 := 1, y_2 := 0, y_3 := 0, z_1 := 0, z_2 := 1, z_3 := 0\}$ can serve as a mismatch witness if it satisfies the miter, where $f_\rho = 0$ and $M_\rho = 2$.

Our witness-extension approach for P-equivalence is *not* applied as part of the ExtWit function following the `bucket` SAT query in `EBatP` (Alg. 3, line 5) or the `misms` SAT query in SiftMis (Alg. 1, line 3). Instead, we heuristically bias the SAT solver before the above queries to favor high-merit witnesses. Specifically, we leverage the SAT solver's capability to *bias* variables in a given set V toward given target polarities for a particular SAT query. This involves *boosting* the variable decision heuristic scores for each $v \in V$ before the query, followed by *forcing* the variables in V to their target polarities (i.e., whenever $v \in V$ is selected by the decision heuristic, it is assigned its target polarity) [2, 30, 31].

We employ *alternation*: before each SAT query, we bias all inputs in $I^Y \cup I^Z$ to a polarity a, initially set to 1 and flipped to $\neg a$ after each query. We also tested *fixing* the target polarity to 0 or 1, but alternation performed better. Its superiority likely stems from its adaptability both to instances where 1 maximizes the merit and those where 0 is advantageous. Despite its simplicity, our approach achieves remarkable efficiency, enabling the solution of over three times as many instances (Sect. 7).

5.2 Blocking for P-Equivalence

We introduce a new blocking algorithm for P-equivalence, shown in Alg. 5. It can be configured to either `enf` (based on [26]) or `dyn` (novel).

Algorithm 5 BlockMisP(`bucket`, σ).

1: **Input**: `alg` $\in \{$`enf`, `dyn`$\}$
2: **if** `alg` $=$ `enf` or $M_\sigma < i - 3$ **then**
3: Add the following clause to `bucket`: $C := \bigvee x^+_{pq} | p : y_p \in I^Y_{l_\sigma}$ and $q : z_q \in I^Z_{f_\sigma}$
4: **else** ▷ `alg` $=$ `dyn` and $M_\sigma \geq i - 3$
5: **for** every total permutation π from $I^Y_{l_\sigma}$ to $I^Z_{l_\sigma}$ **do**
6: Clause $C := \{\}$ ▷ blocking clause per π
7: **for** every $p : y_p \in I^Y_{l_\sigma}$ **do**
8: Let q be the index of $z_q = \pi(y_p)$
9: $C := C \cup \{\neg x^+_{pq}\}$ ▷ $\sigma(y_p) = \sigma(z_q) = l_\sigma$
10: Add C to `bucket`

`enf` follows `BooM`'s mismatch-blocking approach to P-equivalence, reviewed in Sect. 3.2.1. It creates a single blocking clause C shown in line 3. Specifically, it adds x^+_{pq} to C for every combination of $I^Y_{l_\sigma} := \{y_p \in I^Y | \sigma(y_p) = l_\sigma\}$ and $I^Z_{f_\sigma} := \{z_q \in I^Z | \sigma(z_q) = f_\sigma\}$ (using l_σ and f_σ instead of 0 and 1 in `BooMS`, respectively, because it yielded slightly better results in preliminary experiments).

We observed that, for high-merit witnesses, `enf` is less efficient than our `dyn` algorithm (lines 5–10). `dyn` adds $(i - M_\sigma)!$ clauses for every one of the $(i - M_\sigma)!$ total permutations π from $I^Y_{l_\sigma}$ to $I^Z_{l_\sigma}$, where every such clause blocks $M_\sigma!$ total mismatches. Every clause blocks a (partial) mismatch satisfied by the current witness, with all the clauses together blocking *all* currently satisfied mismatches, including the original π. Since the number of clauses grows super-exponentially as M_σ decreases, Alg. 5 reverts to `enf` whenever $M_\sigma < i - 3$.

For example, let σ be a witness with merit $i-1$, where $f_\sigma = 1$ and $\sigma(y_1) = \sigma(z_1) := 0$ (with the other variables assigned 1, as expected when f_σ is 1 and M_σ is $i-1$). `enf` would block with $x^+_{12} \vee x^+_{13} \vee \ldots \vee x^+_{1i}$, whereas `dyn` would block with the unit clause $\neg x^+_{11}$. Because of the constraint $\sum_{q=1}^{i} x^+_{1q} = 1$ (recall Fig. 2, with x^-_{pq}'s fixed to 0 for P-equivalence), both clauses achieve the same effect as expected, but `dyn` adds a substantially smaller clause.

6 Correctness Proof Outline

This section outlines the correctness proofs of our new algorithms.

Consider a straightforward *picker* classification algorithm. Given a bucket with total mappings, the picker iteratively removes a total mapping π from the bucket and reports it only if it is a match. Soundness (as per Def. 12) and termination are guaranteed by construction.

Although our picker implementation (`EBatP` in Alg. 3) classifies and removes multiple total mappings at once, correctness is still guaranteed as long as the invariants in Fig. 4 hold for each iteration (note that Alg. 3 reports the matches, and only them, by construction).

1. **Termination**: Every examined total mapping π is removed.
2. **Soundness**: Any removed total mapping is a match iff π is a match.

Figure 4 Loop invariants for `EBatP` correctness.

Furthermore, one can easily verify that meeting the match-handling invariants in Fig. 5 and the mismatch-handling invariants in Fig. 6 ensures the `EBatP` loop invariants of termination and soundness in Fig. 4.

Moreover, our match-handling algorithm, presented in Sect. 4.1 as part of `EBatP`, satisfies the invariants in Fig. 5 by construction. The invariants in Fig. 6 are also satisfied by construction by the mismatch-handling procedures, including the witness-extension by generalization and blocking introduced already in `BooMS` (recall Sect. 3), as well as our novel blocking procedure for P-equivalence (Sect. 5.2). Our novel witness-extension procedure for P-equivalence (Sect. 5.1) clearly satisfies them, as it only adjusts the heuristics.

1. Given a match π, StrenMatch returns a match π' such that $\pi' \subseteq \pi$.
2. Given a match π', BlkMatch removes π' and only π' from `bucket`.

Figure 5 Match-handling invariants.

1. Given a witness σ for π, ExtWit returns a witness $\sigma' \subseteq \sigma$ for π.
2. Given a witness σ', BlkMis blocks all the mismatches witnessed by σ' in the given SAT instance, and no additional mappings.

Figure 6 Mismatch-handling invariants.

Similarly, one can easily formulate loop invariants to prove SiftMis correct, assuming the mismatch-handling invariants in Fig. 6. The correctness of `BooMS` and `EBatC` follows directly from that of SiftMis and `EBatP`.

7 Experimental Results

We implemented our new algorithms and the baseline `BooMS` algorithm within a new open-source all-Boolean-matching tool, `EBat`[1]. Based on preliminary experiments, we used the IntelSAT SAT solver [32] for all queries, except for UNSAT core extraction on the dual circuit, where CaDiCaL [8] proved more effective.

As the original `BooMS` in `BooM` [26, 25] does not function correctly, we use our re-implementation of `BooMS` as the baseline for comparison.

We adapted benchmarks from the following relevant works: [21], [26], and [15]. All circuits were converted to the AIGER format [9], with multi-output circuits transformed into single-output ones as described below. To manage the complexity of all-Boolean-matching, and because circuits with more than 50 inputs are rarely encountered in our industrial practice, we restricted our selection to circuits with up to 50 inputs. Below, we provide a detailed description of our benchmark selection process, resulting in a total of 388 instances, ranging from 1 to 32,153 gates:

From `BooM` [26]: We reused the ITC'99 suite [17], also used in `BooM` experiments [26]. Each benchmark consists of a combinatorial circuit and its optimized counterpart. To focus on a single output, we used *aigsplit* [9] to split the outputs and compared each instance to its optimized version. In total, we considered 9 circuits, resulting in 268 benchmarks.

From ICCAD'23 Boolean matching contest [15]: We selected 3 out of 5 circuits with up to 50 inputs (the other 2 are currently infeasible for all-Boolean matching) and transformed them into single-output circuits by creating a benchmark for every combination of output versus output, resulting in 48 benchmarks.

From [21]: We selected benchmarks originating from the EPFL [4] and ISCAS'85 [13], already transformed by [21] to be single-output by either applying `or`, `xor` or `last` (output) operator. We created six all-Boolean-matching instances from each circuit by comparing: `or` vs. `or`, `or` vs. `xor`, `or` vs. `last`, `xor` vs. `xor`, `xor` vs. `last`, and `last` vs. `last`. In total, we used 12 circuits from [21] (6 from EPFL and 6 from ISCAS'85), resulting in 72 benchmarks.

We conducted experiments on Intel®Xeon® machines (32GB memory, 3GHz CPU) with a 1-minute timeout, reflecting our industrial requirements. We measured the number of *solved* instances (all matches found or absence proven within timeout) and the *PAR-2 score* (solved benchmarks contribute runtime; unsolved ones contribute twice the timeout).

We studied the impact of:

1. *High-level algorithms*: the baseline `BooMS` vs. our novel `EBatP` and `EBatC`.
2. *Blocking schemes for P-equivalence*: the baseline `enf` vs. the new `dyn`.
3. *Witness-extension*:
   a. *NP-equivalence*: `BTS` as in `BooMS` vs. `FTS&MUC` [21] vs. our novel `BTS&MUC`.
   b. *P-equivalence*: none in the baseline `BooMS` vs. our novel `Alternation` algorithm and its two simpler versions, `Fixed to 1` and `Fixed to 0`.
4. *Strengthening*: none as in the baseline `BooMS` vs. our novel strengthening approach.

7.1 The Results

Table 1 summarizes our experimental results. To complement the data in the table, for NP-equivalence, at least one configuration solved 328 benchmarks, with 65 not matching and 2 having only one match. For P-equivalence, the corresponding numbers are 333 solved, with 75 not matching and 6 with one match.

[1] `EBat` and all the benchmarks are available at `https://github.com/yogevshalmon/ebat`.

Table 1 Summary of results for NP-equivalence (top) and P-equivalence (bottom) for a 1-minute timeout. Each non-header row corresponds to a configuration of our EBat tool. The default configuration in EBat is highlighted in light gray, while the baseline configuration from BooM is highlighted in dark gray. Other configurations correspond to the EBat default with one (and only one) algorithmic feature changed, and with unchanged features appearing faded. The first four columns represent the algorithmic features of each configuration, as indicated by the column titles. The last two columns display the results: the number of solved instances and the PAR-2 score.

NP-Equivalence					
Algorithm	**Blocking**	**Witness-Extension**	**Strengthening**	**Solved**	**PAR-2**
EBatC	default	BTS&MUC	✓	327	686
EBatC	default	FTS&MUC	✓	325	939
EBatP	default	BTS&MUC	✓	322	1252
EBatC	default	BTS	✓	308	2932
EBatC	default	BTS&MUC	–	99	27826
BooMS	default	BTS	–	86	29466
P-Equivalence					
Algorithm	**Blocking**	**Witness-Extension**	**Strengthening**	**Solved**	**PAR-2**
EBatC	dyn	Alternation	✓	329	871
EBatC	enf	Alternation	✓	326	1324
EBatP	dyn	Alternation	✓	308	3467
EBatC	dyn	Fixed to 0	✓	303	4325
EBatC	dyn	Fixed to 1	✓	301	4465
EBatC	dyn	Alternation	–	115	26232
EBatC	dyn	–	✓	106	27460
BooMS	enf	–	–	90	29348

Overall, the default configuration of our new all-Boolean-matching tool, EBat, significantly outperforms the baseline BooMS algorithm for both NP- and P-equivalence, solving 3 to 4 times more benchmarks within our 1-minute timeout. Furthermore, on instances completed by both BooMS and our default configuration of EBat, BooMS issues, on average, 13,000 times more SAT queries for NP-equivalence and 1,500 times more SAT queries for P-equivalence.

In an additional experiment, we confirmed that this advantage persists with a longer 10-minute timeout: for P-equivalence, BooMS solves 95 instances, while EBat solves 339; for NP-equivalence, BooMS solves 87, compared to 335 by EBat.

Table 1 shows that, for NP-equivalence, strengthening is the key factor in improving performance. Also, our default generalization approach, BTS&MUC, outperforms both BTS, applied by BooM, and FTS&MUC from [21]. Finally, comparing the high-level algorithms, EBatC outperforms EBatP, while switching to BooMS in the default configuration is impossible, as BooMS cannot apply strengthening by construction.

For P-equivalence, our novel witness-extension procedure contributes the most, closely followed by strengthening; together, they enable solving 329 instances, compared to only 106–115 without either. Additionally, in witness-extension, alternation outperforms fixing to 0 or 1. Also, EBatC surpasses EBatP, and our dyn blocking scheme outperforms BooM's enf. Additionally, even though EBatC outperforms EBatP on the same default configuration of EBat, for P-Equivalence, one instance is solved uniquely by EBatP (for NP-Equivalence, all instances solved by EBatP are also solved by EBatC).

8 Conclusion

Motivated by the industrial need for automated, timing-aware library mapping, we presented the first dedicated study on enumerating all matches between two Boolean circuits (all-Boolean-matching). We introduced novel algorithms and implemented them from scratch in `EBat`, the only open-source tool for this problem. `EBat` solves 3 to 4 times more benchmarks than the state-of-the-art algorithm `BooMS`, within both the application-driven 1-minute timeout and a 10-minute timeout for NP- and P-equivalence.

In future work, we plan to extend `EBat` to handle multiple outputs and sequential circuits, and implement circuit preprocessing techniques [12, 36, 5] to further improve its performance.

References

1 Afshin Abdollahi. Signature based boolean matching in the presence of don't cares. In Limor Fix, editor, *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*, pages 642–647. ACM, 2008. doi:10.1145/1391469.1391635.

2 Sabih Agbaria, Dan Carmi, Orly Cohen, Dmitry Korchemny, Michael Lifshits, and Alexander Nadel. Sat-based semiformal verification of hardware. 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 25–32. IEEE, 2010. URL: https://ieeexplore.ieee.org/document/5770929/.

3 Manindra Agrawal and Thomas Thierauf. The boolean isomorphism problem. In *37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996*, pages 422–430. IEEE Computer Society, 1996. doi:10.1109/SFCS.1996.548501.

4 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.

5 Daniil Averkov, Tatiana Belova, Gregory Emdin, Mikhail Goncharov, Viktoriia Krivogornitsyna, Alexander S. Kulikov, Fedor Kurmazov, Daniil Levtsov, Georgie Levtsov, Vsevolod Vaskin, and Aleksey Vorobiev. Cirbo: A new tool for boolean circuit analysis and synthesis. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(11):11105–11112, April 2025. doi:10.1609/aaai.v39i11.33207.

6 Luca Benini and Giovanni De Micheli. A survey of boolean matching techniques for library binding. *ACM Transactions on Design Automation of Electronic Systems*, 2(3), 1997. doi:10.1145/264995.264996.

7 Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. Detecting cardinality constraints in CNF. 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 285–301. Springer, 2014. doi:10.1007/978-3-319-09284-3_22.

8 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

9 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.

10 Bernd Borchert, Desh Ranjan, and Frank Stephan. On the computational complexity of some classical equivalence relations on boolean functions. *Theory Comput. Syst.*, 31(6):679–693, 1998. doi:10.1007/S002240000109.

11 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.

12 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.

13 Franc Brglez and Hideo Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator. In *Fortran. ISCAS'85*, 1985.

14 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.

15 Chung-Han Chou, Chih-Jen Jacky Hsu, Chi-An Rocky Wu, Kuan-Hua Tu, and Kei-Yong Khoo. Invited paper: 2023 iccad cad contest problem a: Multi-bit large-scale boolean matching. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–4, 2023. doi:10.1109/ICCAD57390.2023.10323797.

16 Jason Cong and Yean-Yow Hwang. Boolean matching for lut-based logic blocks with applications toarchitecture evaluation and technology mapping. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 20(9):1077–1090, 2001. doi:10.1109/43.945303.

17 F. Corno, M.S. Reorda, and G. Squillero. Rt-level itc'99 benchmarks and first atpg results. *IEEE Design & Test of Computers*, 17(3):44–53, 2000. doi:10.1109/54.867894.

18 Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 36–41. Springer, 2006. doi:10.1007/11814948_5.

19 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. URL: http://dl.acm.org/citation.cfm?id=2157675.

20 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT, Proceedings*, 2003. doi:10.1007/978-3-540-24605-3_37.

21 Dror Fried, Alexander Nadel, Roberto Sebastiani, and Yogev Shalmon. Entailing generalization boosts enumeration. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*, volume 305 of *LIPIcs*, pages 13:1–13:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.SAT.2024.13.

22 Shao-Lun Huang, Wei-Hsun Lin, Po-Kai Huang, and Chung-Yang Huang. Match and replace: A functional ECO engine for multierror circuit rectification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 32(3):467–478, 2013. doi:10.1109/TCAD.2012.2226456.

23 Zheng Huang, Lingli Wang, Yakov Nasikovskiy, and Alan Mishchenko. Fast boolean matching based on NPN classification. In *2013 International Conference on Field-Programmable Technology, FPT 2013, Kyoto, Japan, December 9-11, 2013*, pages 310–313. IEEE, 2013. doi:10.1109/FPT.2013.6718374.

24 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. doi:10.1147/SJ.83.0178.

25 Chih Fan Lai, Jie Hong R. Jiang, and Kuo Hua Wang. Boolean matching of function vectors with strengthened learning. In *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, 2010. doi:10.1109/ICCAD.2010.5654215.

26 Chih Fan Lai, Jie Hong R. Jiang, and Kuo Hua Wang. BooM: A decision procedure for Boolean matching with abstraction and dynamic learning. In *Proceedings - Design Automation Conference*, 2010. doi:10.1145/1837274.1837398.

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 Alan Mishchenko, Supratik Chatterjee, Robert Brayton, Xiangjian Wang, and Tony Kam. Technology mapping with boolean matching, supergates and choices. Technical report, EECS Department, University of California, Berkeley, March 2005.

29 Janett Mohnke, Paul Molitor, and Sharad Malik. Application of bdds in boolean matching techniques for formal logic combinational verification. *Int. J. Softw. Tools Technol. Transf.*, 3(2):207–216, 2001. doi:10.1007/S100090100039.

30 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. doi:10.23919/FMCAD.2019.8894273.

31 Alexander Nadel. Polarity and variable selection heuristics for sat-based anytime maxsat. *J. Satisf. Boolean Model. Comput.*, 12(1):17–22, 2020. doi:10.3233/SAT-200126.

32 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.

33 Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In *Theory and Applications of Satisfiability Testing - SAT, Proceedings*, 2012. doi:10.1007/978-3-642-31612-8_19.

34 Emil L. Post. Introduction to a general theory of elementary propositions. *American Journal of Mathematics*, 43, 1921.

35 W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952. doi:10.1080/00029890.1952.11988183.

36 Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. eslim: Circuit minimization with SAT based local improvement. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*, volume 305 of *LIPIcs*, pages 23:1–23:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.SAT.2024.23.

37 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.

38 Roberto Sebastiani. Are you satisfied by this partial assignment? *CoRR*, abs/2003.04225, 2020. arXiv:2003.04225.

39 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.

40 Pawel Swierczynski, Marc Fyrbiak, Christof Paar, Christophe Huriaux, and Russell Tessier. Protecting against cryptographic trojans in fpgas. In *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015*, pages 151–154. IEEE Computer Society, 2015. doi:10.1109/FCCM.2015.55.

41 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.

42 Kuo-Hua Wang, Chung-Ming Chan, and Jung-Chang Liu. Simulation and sat-based boolean matching for large boolean networks. In *Proceedings of the 46th Design Automation Conference,*

DAC 2009, San Francisco, CA, USA, July 26-31, 2009, pages 396–401. ACM, 2009. doi:10.1145/1629911.1630016.

43 Chaofan Yu, Lingli Wang, Chun Zhang, Yu Hu, and Lei He. Fast filter-based boolean matchers. *IEEE Embed. Syst. Lett.*, 5(4):65–68, 2013. doi:10.1109/LES.2013.2280582.

44 Juling Zhang, Guowu Yang, William N. N. Hung, and Jinzhao Wu. A canonical-based NPN boolean matching algorithm utilizing boolean difference and cofactor signature. *IEEE Access*, 5:27777–27785, 2017. doi:10.1109/ACCESS.2017.2778338.

Symbolic Conflict Analysis in Pseudo-Boolean Optimization

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

In the the last two decades, a lot of effort has been devoted to the development of satisfiability-checking tools for a variety of SAT-related problems. However, most of these tools lack optimization capabilities. That is, instead of finding any solution, one is sometimes interested in a solution that is best according to some criterion.

Pseudo-Boolean solvers can be used to deal with optimization by successively solving a series of problems that contain an additional pseudo-Boolean constraint expressing that a better solution is required. A key point for the success of this simple approach is that lemmas that are learned for one problem can be reused for subsequent ones.

In this paper we go one step further and show how, by using a simple symbolic conflict analysis procedure, not only can lemmas be reused between problems but also strengthened, thus further pruning the search space traversal. In addition, we show how this technique automatically allows one to infer upper bounds in maximization problems, thus giving an estimation of how far the solver is from finding an optimal solution. Experimental results with our PB solver reveal that (i) this technique is indeed effective in practice, providing important speedups in problems where several solutions are found and (ii) on problems with very few solutions, where the impact of our technique is limited, its overhead is negligible.

2012 ACM Subject Classification Theory of computation

Keywords and phrases SAT, Pseudo-Boolean Optimization, Conflict Analysis

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.23

Supplementary Material
Software (Source Code): `https://github.com/dearzhaorui/symbolic-conflict-analysis`
archived at `swh:1:dir:f5894275ca84f010f2626a78762667102cefb9f4`

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

SAT solvers are nowadays routinely used in an increasing number of applications areas. Despite success stories initially emerged in the area of verification [4], they are now common in security [10, 35], cryptography [34] and even mathematics [18]. In parallel to these practical developments, theoretical works have shown that CDCL [23], the most successful procedure for SAT, cannot solve certain type of problems (e.g. the pigeon-hole principle [17]) in polynomial time [2, 28].


© Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Rui Zhao;
licensed under Creative Commons License CC-BY 4.0



Pseudo-Boolean (PB) solving, also known as 0-1 Integer Linear Programming, has established itself as a promising alternative to SAT. First of all, 0-1 linear constraints are more general than clauses, and allow one to encode problems in a more compact way. Secondly, cutting planes [7], the underlying proof system of CDCL-based PB solvers [31] is exponentially more powerful than resolution [30], the proof system which CDCL-based SAT solvers rely on. Thus, PB solvers are, at least from the theoretical point of view, exponentially more powerful than SAT solvers.

Last, but not least, sometimes satisfiability checking is not enough. In several applications [16, 1, 26], one wants to find an optimal solution according to some criterion. Unlike what happens with SAT solvers, it is very easy to turn a PB solver into a PB optimizer, which finds a solution to a set of 0-1 linear constraints that maximizes a given objective 0-1 linear function. The solver is initially run on the set of constraints, ignoring the objective function, in order to obtain a first solution. If the objective function value for this solution is C, a constraint expressing that only solutions with objective function value strictly larger than C is added and the solver is executed again. The process is repeated until the solver is not able to find further solutions, concluding that the last solution found is optimal.

A noteworthy aspect of the previous approach, known as *Linear Search*, is that any lemma learned by the solver when looking for a solution larger than C can still be used at any future point when a solution larger than $C' > C$ is sought. This well-known fact [25] is crucial for the performance of this method.

Contributions. In this paper, we go one step further and show how constraints learned when looking for a solution larger than C can not only be reused, but also automatically *strengthened* so that a stronger version of them is available during the search for solutions larger than $C' > C$. This is accomplished by considering C as a symbolic variable and adapting conflict analysis to take care of this feature. As a by-product of this symbolic way of treating constraints, one can automatically extract upper bounds from them, thus giving an estimation of how far the solver is from reaching an optimal solution. Experimental results show that the overhead of this symbolic procedure is negligible, and that its ability to further prune the search space results in important runtime improvements.

Related Work. Pseudo-Boolean optimization is a well-studied problem for which three main methods exist: Linear Search, Core-guided Search [13, 24, 9], and Implicit Hitting Set approaches [33, 32]. They are considered to be complementary in the type of instances for which they work best and solvers might even use combinations of them (e.g. RoundingSat combines Linear with Core-guided Search). Linear Search, which is the topic of this paper, has also been applied to the MaxSAT problem. Two prominent contributions in this direction are Pacose [27] and MaxCDCL solver [21, 22], which combines a branch and bound search with a lower-bounding procedure to prune the search space. As far as we know, no previous work uses a symbolic treatment of the objective function and no other linear-search method can compute upper bounds with no additional cost. Hence, we believe the community will benefit from the introduction of this novel technique and will develop additional ways to further exploit it.

This paper is organized as follows. After some preliminaries in Section 2, we revisit the conflict analysis procedure of PB solvers in Section 3. Section 4 presents the main contributions of this paper: the symbolic conflict analysis procedure and its additional ability to compute upper bounds. Experimental evidence of their positive impact on a solver and a careful analysis of the data collected is reported in Section 5. We conclude in Section 6, where we also outline future research directions.

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 and Logical Consequence. 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*. A formula F is a *logical consequence* of G if any model of G is also a model of F.

Inference Rules. In order to determine the satisfiability of a set of constraints, one can use the *cutting planes* proof system, which consists of axioms $l \geq 0$ for all literals l, and the following two inference rules:

$$\text{Linear combination:} \quad \frac{\sum_i a_i l_i \geq A \quad \sum_i b_i l_i \geq B}{\sum_i (\alpha a_i + \beta b_i) l_i \geq \alpha A + \beta B} \qquad \alpha, \beta \in \mathbb{N}^+$$

$$\text{Division:} \quad \frac{\sum_i a_i l_i \geq A}{\sum_i \lceil a_i/\alpha \rceil l_i \geq \lceil A/\alpha \rceil} \quad \alpha \in \mathbb{N}^+$$

It is well known [19] that a set of constraints is unsatisfiable if and only if $0 \geq 1$ can be derived using these rules. We note that in the application of linear combination we implicitly assume that any constraint is simplified by using that fact that $l + \overline{l} = 1$. For example, the constraint $2x + 3\overline{x} + 5y \geq 7$ can be rewritten as $2(x + \overline{x}) + \overline{x} + 5y \geq 7$ and is hence equivalent to $\overline{x} + 5y \geq 5$. Another well-known rule, which is very useful in PB solving, limits coefficients to be at most equal to the degree:

$$\text{Saturation:} \quad \frac{\sum_i a_i l_i \geq A}{\sum_i min(a_i, A)\, l_i \geq A}$$

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 [23] algorithm for SAT can be applied to the pseudo-Boolean case [31]. 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 uses the aforementioned inference rules to derive 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. If we see ρ as a sequence, any literal appearing in the sequence between the k-th decision literal (included) and before the $(k+1)$-th one is said to belong to *decision level* k.

Pseudo-Boolean Optimization. Given a set S of PB-constraint and an *objective* function $\sum_i c_i l_i$, the *Pseudo-Boolean Optimization* problem consists of finding, among all assignments that satisfy S, one that maximizes the value of the objective function. A very simple approach for this problem is to first run a PB solver to determine the satisfiability of S. If a solution is found, for which the objective function has value C, a constraint $\sum_i c_i l_i \geq C + 1$ is added to S and the same solver is launched again. Eventually, the solver will report the unsatisfiability of the augmented set of constraints, and the last found solution is an optimal one. This is sometimes known as the *Linear Search* approach to PB optimization.

3 Conflict Analysis in CDCL-Based Pseudo-Boolean Solvers

In CDCL-based PB solvers, when unit propagation finds a conflicting constraint, a procedure is launched whose ultimate goal is to derive a new constraint that (i) is a logical consequence of the current formula and hence can safely be added to it, and (ii) allows the solver to backjump to some previous decision level and propagate a literal at that point. This mimics conflict analysis in SAT solvers, where the 1-UIP scheme [36] has established itself as the dominant approach and, after twenty years, only small variants have been added to it (e.g. [12]).

Conflict analysis in PB solvers is a more complex task than in SAT and one can find different variants in state-of-the-art CDCL-based PB solvers. However, we believe that Algorithm 1 is an adequate abstraction of most of them. For the purpose of this paper, this simplified presentation is detailed enough so that we can introduce our procedure in Section 4. The overall idea of Algorithm 1 is that a series of linear combination steps are applied between the conflicting constraint and the reasons for certain literals in the trail ρ until we obtain a constraint that propagates some literal at some previous decision level. This is essentially what conflict analysis does in CDCL-based SAT solvers, where resolution can be seen as a concrete case of linear combination.

There are three differences we would like to remark. The first difference can be found in the loop in lines 4-6. In a SAT solver, this loop looks for the topmost literal $l \in \rho$ whose negation appears in $confCtr$. But the actual property we want about l is that the addition of l to ρ is the one that caused $confCtr$ to be false. In a PB constraint, the literal l with this property need not be the topmost in ρ whose negation appears in $confCtr$. For example, if $\rho = (\overline{x}, \overline{y}, \overline{z})$, the conflicting constraint $2x + y + z \geq 3$ already became false when $\overline{x}$ was added to ρ, although the topmost literal whose negation appears in the constraint is $\overline{z}$.

Algorithm 1 CDCL-Based PB solver Conflict Analysis.

```
1  Function Conflict-Analysis(Assignment ρ, Constraint confCtr):
2      while true do
3          int slck := slack(confCtr, ρ)  // slck < 0 because confCtr is a conflict.
4          while slck < 0 do
5              Literal l := ρ.pop() // ρ is a stack
6              if l̄ ∈ confCtr then slck := slck + coef(l̄, confCtr)
7          Constraint reasonCtr := reason(l, ρ)  // l ∈ reasonCtr and l̄ ∈ confCtr
8          weakenConstraints(l, confCtr, reasonCtr)
9          α := coef(l̄, confCtr)
10         β := coef(l, reasonCtr)
11         confCtr := linearComb(β, confCtr, α, reasonCtr)  // confCtr is false in ρ
12         if confCtr propagates at previous decision level then return confCtr
```

The second, and probably the most important difference, is that we want the application of a linear combination to preserve the fact that the resulting constraint is false in ρ. This property, which trivially holds in SAT solvers, is not true when using linear combinations in general.

▶ **Example 1.** Let us consider the two constraints $5v+2\overline{x}+3y+3z \geq 6$ and $5\overline{v}+5x+3y+4z \geq 6$. Assume that we decide on $\overline{x}$ and later on $\overline{y}$. The second constraint propagates z and $\overline{v}$ and hence we have the assignment $\rho = (\overline{x}, \overline{y}, z, \overline{v})$. But now the first constraint is conflicting. After the loop in lines 4-6, l is $\overline{v}$ and, if line 8 were not present, $\alpha = \beta = 5$ and hence the linear combination[1] would be $15x + 30y + 35z \geq 25$ (note that the right hand side is $30 + 30 - 25 - 10$, where the negative numbers come from canceling 25 units of variable v and 10 units of variable x). One can now check that the derived constraint is no longer false in the new assignment $(\overline{x}, \overline{y}, z)$.

To avoid this problem, which is well-known in the PB [6, 11, 3] and the ILP communities [20, 29], function *weakenConstraints* replaces the constraints by weaker versions, that is, logical consequences. Probably the easiest solution is to replace the reason of a literal by the clause that expresses the implication. In the previous example we could convert the reason for v, which was $5\overline{v} + 5x + 3y + 4z \geq 6$, to $\overline{v} + x + y \geq 1$, expressing that if both x and y are false, then $\overline{v}$ needs to be true. Now $\alpha = 5$ and $\beta = 1$ and the linear combination that consists of adding 5 times the new reason clause with the conflicting constraint is $3x + 8y + 3z \geq 4$, which is now false in the assignment $(\overline{x}, \overline{y}, z)$. Other possibilities for weakening the reason constraint exist [6, 11, 3] but they are out of the scope of this paper.

The last difference in the procedure refers to the termination of the procedure. In SAT solving, conflict analysis terminates when the 1-UIP is found, which guarantees that the current $confCtr$ propagates at some previous decision level. In PB solving, that constraint might be propagating even before the 1-UIP is found and hence it makes sense to check, after every linear combination step, whether $confCtr$ propagates at some previous level. In order to alleviate the computational effort of this check, we use the fact that if $confCtr$ and $reasonCtr$ do not have any contradictory literal other than l, then if $confCtr$ does not propagate at some previous decision level, neither does the constraint resulting of the linear combination between $confCtr$ and $reasonCtr$ computed in line 11 of Algorithm 1. We prove this in the following.

[1] Note that in Algorithm 1 one can use the least common multiple to compute smaller α, β but we omit this detail in the presentation.

▶ **Lemma 2.** *Let $\mathcal{C} := al + C \geq k_c$ and $\mathcal{D} := b\bar{l} + D \geq k_d$ be two PB constraints such that their parts C and D have no contradictory literal and let $\mathcal{R}$ be the linear combination $b\mathcal{C} + a\mathcal{D}$. For any assignment ρ that assigns some value to l, if $\mathcal{R}$ is false in ρ then either $\mathcal{C}$ or $\mathcal{D}$ are also false in ρ.*

Proof. Given a constraint $X \geq d$, we denote by $SNF(X, \rho)$ the (S)um of the coefficients of all literals in X that are (N)on-(F)alse in ρ. It holds that $SNF(X, \rho) = slack(X \geq d, \rho) + d$.

Let us assume that neither $\mathcal{C}$ nor $\mathcal{D}$ are false in ρ and reach a contradiction. Since $\mathcal{R}$ is false in ρ, we have that $slack(\mathcal{R}, \rho) < 0$. Since we know that $\mathcal{R}$ is of the form $bC + aD \geq bk_c + ak_d - ab$, with no contradictory literals between C and D, we have that $slack(\mathcal{R}, \rho) = SNF(bC+aD, \rho) - bk_c - ak_d + ab = bSNF(C, \rho) + aSNF(D, \rho) - bk_c - ak_d + ab$. Note that the last equality holds because C and D have no contradictory literals, otherwise cancellation between those literals could make it invalid. All in all, we can conclude that $bSNF(C, \rho) + aSNF(D, \rho) < bk_c + ak_d - ab$.

Now, since $\mathcal{C}$ is not false, we have that $slack(\mathcal{C}, \rho) \geq 0$ and hence $SNF(al + C, \rho) - k_c \geq 0$ and also $aSNF(l, \rho) + SNF(C, \rho) - k_c \geq 0$ and $abSNF(l, \rho) + bSNF(C, \rho) - bk_c \geq 0$. Similarly $abSNF(\bar{l}, \rho) + aSNF(D, \rho) - ak_d \geq 0$. If we add these last two inequalities, and considering that $SNF(l, \rho) + SNF(\bar{l}, \rho) = 1$ because l is assigned in ρ, we have that $ab + bSNF(C, \rho) + aSNF(D, \rho) - bk_c - ak_d \geq 0$, which contradicts the last inequality of the previous paragraph. ◀

▶ **Corollary 3.** *Let $\mathcal{C} := al + C \geq k_c$ and $\mathcal{D} := b\bar{l} + D \geq k_d$ be two PB constraints such that their parts C and D have no contradictory literal and let $\mathcal{R}$ be the linear combination $b\mathcal{C} + a\mathcal{D}$. For any assignment ρ that assigns some value to l, if unit propagation of $\mathcal{R}$ under ρ propagates a literal l_p, then either $\mathcal{C}$ or $\mathcal{D}$ also propagate l_p under ρ.*

Proof. Apply Lemma 2 with assignment $\rho \cup \{\bar{l}_p\}$ and use the fact that a constraint $\mathcal{C}$ is false in $\rho \cup \{\bar{l}_p\}$ if and only if $\mathcal{C}$ propagates a literal l_p under ρ. ◀

We conclude this section with an example showing the execution of a CDCL-based PB solver that uses a conflict analysis method based on Algorithm 1. This allow us to show how our procedure of Section 4 is able to improve it. For simplicity, we have chosen an example for which the application of *weakenConstraints* is never necessary.

▶ **Example 4.**

$$
\begin{array}{lllllllllll}
Max & & & x & +2y & +3z & +4u & +5v & +6w & & \\
 & \overline{a} & & +\overline{x} & +\overline{y} & & & & & \geq 2 & (C_1) \\
 & & \overline{b} & & & +\overline{z} & +\overline{u} & & & \geq 2 & (C_2) \\
 & & & \overline{c} & & & & +\overline{v} & +\overline{w} & \geq 2 & (C_3)
\end{array}
$$

First solution. Since no constraint is propagating, the solver decides on a literal, say a, which propagates $\overline{x}$ and $\overline{y}$ due to C_1. Next decision is b, which propagates $\overline{z}$ and $\overline{u}$ due to C_2. Next decision is c, which propagates $\overline{v}$ and $\overline{w}$ due to C_3. All constraints are satisfied and we have found a solution for which the objective function has value 0.

Second solution. Since we have to look for a better solution, constraint $C_4 := x + 2y + 3z + 4u + 5v + 6w \geq 1$ is added. The solver makes the same decisions and propagations as before, but the assignment $\rho = (a^d, \overline{x}_1, \overline{y}_1, b^d, \overline{z}_2, \overline{u}_2, c^d, \overline{v}_3, \overline{w}_3)$, where a superscript d indicates a decision and a subscript i represents that constraint C_i is the reason for that propagation, now falsifies constraint C_4. Conflict analysis is started, popping literal $\overline{w}$ from ρ, making *slck*

be 5 and the loop in lines $4-6$ terminate. Since C_3 is the reason for $\overline{w}$, and $\alpha = 6, \beta = 1$, the linear combination is $1 \cdot C_4 + 6 \cdot C_3$, which is $C_5 := 6\overline{c} + x + 2y + 3z + 4u + \overline{v} \geq 2$. This constraint propagates $\overline{c}$ at decision level 2, and hence the solver backjumps to produce assignment $\rho = (a^d, \overline{x}_1, \overline{y}_1, b^d, \overline{z}_2, \overline{u}_2, \overline{c}_5)$. No other constraint propagates and the solver now decides on v, which propagates $\overline{w}$ due to C_3 and ρ is a solution with value 5.

Third solution. Since a better solution is needed we replace C_4 by $x + 2y + 3z + 4u + 5v + 6w \geq 6$. The solver restarts and makes the same decisions and propagations to produce $\rho = (a^d, \overline{x}_1, \overline{y}_1, b^d, \overline{z}_2, \overline{u}_2)$. However, now C_4 propagates w and, after that, C_3 propagates $\overline{v}$ and $\overline{c}$, which produces a solution with value 6.

Fourth solution. C_4 now becomes $x + 2y + 3z + 4u + 5v + 6w \geq 7$. The solver restarts and proceeds as before to reach $\rho = (a^d, \overline{x}_1, \overline{y}_1, b^d, \overline{z}_2, \overline{u}_2)$. Now C_4 propagates v and w and the resulting assignment $\rho = (a^d, \overline{x}_1, \overline{y}_1, b^d, \overline{z}_2, \overline{u}_2, v_4, w_4)$ falsifies constraint C_3 The loop at lines $4-6$ pops w from ρ, making *slck* to be 0. Since the coefficient of w in the reason of w is $\beta = 6$ and the coefficient of $\overline{w}$ in C_3 is $\alpha = 1$ the linear combination computed is $6 \cdot C_3 + 1 \cdot C_4$, which is $C_{tmp} := 6\overline{c} + x + 2y + 3z + 4u + \overline{v} \geq 8$. Since this constraint does not propagate at any previous decision level, one more iteration of conflict analysis is required. The loop at lines 4-6 pops literals v and $\overline{u}$, after which the *slck* is 3 and the loop terminates. The procedure will now try to eliminate $\overline{u}$ from C_{tmp} by performing an appropriate linear combination with the reason for $\overline{u}$, which is C_2. This linear combination is $1 \cdot C_{tmp} + 4 \cdot C_2$, which results in $C_6 := 4\overline{b} + 6\overline{c} + x + 2y + \overline{z} + \overline{v} \geq 9$. This constraint is learned since it allows the solver to backjump to decision level 1 and propagate $\overline{c}$ and $\overline{b}$. The assignment is now $\rho = (a^d, \overline{x}_1, \overline{y}_1, \overline{c}_6, \overline{b}_6)$. No other constraint propagates and the solver decides on v, which propagates $\overline{w}$ due to C_3. Next decision is on u, which propagates $\overline{z}$ due to C_2. The current assignment ρ is a solution with value 9.

Next step. C_4 now becomes $x + 2y + 3z + 4u + 5v + 6w \geq 10$. Let us now remark that the solver, apart from the original constraints, only has the lemmas:

$$
\begin{array}{ll}
x + 2y + 3z + 4u + 5v + 6w \geq 10 & (C_4) \\
6\overline{c} + x + 2y + 3z + 4u + \overline{v} \geq 2 & (C_5) \\
4\overline{b} + 6\overline{c} + x + 2y + \overline{z} + \overline{v} \geq 9 & (C_6)
\end{array}
$$

The solver restarts and, since no constraint propagates at decision level zero, it will decide on a. However, as we will show in the next section, this can be done much better.

4 Symbolic Conflict Analysis in CDCL-Based Pseudo-Boolean Solvers

This section starts with the presentation of our procedure. Then, in Section 4.2 we explain how to obtain upper bounds and how to use it inside a binary-search solving procedure. Next, we explain in Section 4.3 how the procedure interacts with other inference rules commonly used in PB solvers.

4.1 The Symbolic Conflict Analysis Procedure

Let us consider again the maximization problem in Example 4. After every solution found, constraint C_4, which states that only better solutions are accepted, is strengthened. However, we know that lemma C_5 was obtained via the linear combination $1 \cdot C_4 + 6 \cdot C_3$, and lemma

C_6 was $1 \cdot C_4 + 6 \cdot C_3 + 4 \cdot C_2$. What we show in this section is how, after every solution is found, not only can we strengthen C_4 but also strengthen every lemma that was derived by using C_4. Moreover, we can do it in a simple and efficient way that only requires minor modifications to the solver. Let us introduce our *symbolic conflict analysis* procedure with the following example.

▶ **Example 5.** Let us consider the same maximization problem as in Example 4.

First solution. We proceed as before in order to find the first solution, with value 0.

Second solution. Now, instead of adding constraint $C_4 := x + 2y + 3z + 4u + 5v + 6w \geq 1$, we modify its degree so that it contains a symbolic representation of it. We consider a symbolic variable δ and instead add constraint $C_4 := x + 2y + 3z + 4u + 5v + 6w \geq 1 \; [\![1 + \delta]\!]$. The symbolic part $[\![1 + \delta]\!]$ intuitively expresses that the objective function has to be at least 1 unit larger than the value of the best solution found so far, which is symbolically represented by a variable δ. In terms of propagation, this symbolic expression will be ignored in all constraints.

Hence, we have the same constraints as in the previous example and the solver constructs $\rho = (a^d, \overline{x}_1, \overline{y}_1, b^d, \overline{z}_2, \overline{u}_2, c^d, \overline{v}_3, \overline{w}_3)$. Our symbolic conflict analysis is started, proceeding as usual except when performing linear combinations, where the symbolic part will be taken into account. As before, we have to compute $1 \cdot C_4 + 6 \cdot C_3$. Note that input constraints like C_3 also have a symbolic representation of the degree. However, in those cases it will be equal to the degree. That is, the linear combination is:

$1 \cdot C_4$:		x	$+2y$	$+3z$	$+4u$	$+5v$		$+6w$	$\geq$	1	$[\![1 + \delta]\!]$
$6 \cdot C_3$:	$6\overline{c}$					$+6\overline{v}$		$+6\overline{w}$	$\geq$	12	$[\![12]\!]$
C_5 :	$6\overline{c}$	$+x$	$+2y$	$+3z$	$+4u$	$+(\overline{v} + 5)$		$+6$	$\geq$	13	$[\![13 + \delta]\!]$

A simple arithmetic manipulation, also on the symbolic part, results in the final learned lemma $C_5 : 6\overline{c} + x + 2y + 3z + 4u + \overline{v} \geq 2 \; [\![2 + \delta]\!]$. Since the symbolic representation is ignored when propagating, the solver proceeds as in the previous example to obtain solution $\rho = (a^d, \overline{x}_1, \overline{y}_1, b^d, \overline{z}_2, \overline{u}_2, \overline{c}_5, v^d, \overline{w}_3)$ with cost 5.

Third solution. This is where our procedure makes the solver behave differently. Since we are looking for a solution of cost larger than 5, we strengthen C_4 to become $x + 2y + 3z + 4u + 5v + 6w \geq 6 \; [\![1 + \delta]\!]$ but we can also automatically strengthen C_5 to become $6\overline{c} + x + 2y + 3z + 4u + \overline{v} \geq 7 \; [\![2 + \delta]\!]$. More explicitly, the new degree of a constraint is equal to symbolic part where variable δ is replaced by the value of the best solution found so far. We can observe that C_5 allows for more propagations than its initial non-strengthened version. However none of these propagations apply here and the solver finds the same solution as before, with value 6.

Fourth solution. Constraints C_4 and C_5 are strengthened again to become now C_4 : $x + 2y + 3z + 4u + 5v + 6w \geq 7 \; [\![1 + \delta]\!]$ and $C_5 : 6\overline{c} + x + 2y + 3z + 4u + \overline{v} \geq 8 \; [\![2 + \delta]\!]$. By making the same decisions, the solver finds again the same conflict and the first linear combination to be computed is again $6 \cdot C_3 + 1 \cdot C_4$, which gives $C_{tmp} : 6\overline{c} + x + 2y + 3z + 4u + \overline{v} \geq 8 \; [\![2 + \delta]\!]$.

This constraint does not propagate at any previous decision level and one additional iteration is needed. The following linear combination is $1 \cdot C_{tmp} + 4 \cdot C_2$, which results in $C_6 : 4\overline{b} + 6\overline{c} + x + 2y + \overline{z} + \overline{v} \geq 9 \; [\![3 + \delta]\!]$. It propagates $\overline{c}$ and $\overline{b}$ at decision level 1 and the solver proceeds identically to obtain a solution of value 9.

Fifth solution. Constraints C_4, C_5 and C_6 are strengthened again and become

$$\begin{array}{ll} C_4 : x + 2y + 3z + 4u + 5v + 6w \geq 10 & [\![1 + \delta]\!] \\ C_5 : 6\overline{c} + x + 2y + 3z + 4u + \overline{v} \geq 11 & [\![2 + \delta]\!] \\ C_6 : 4\overline{b} + 6\overline{c} + x + 2y + \overline{z} + \overline{v} \geq 12 & [\![3 + \delta]\!] \end{array}$$

These constraints are now much stronger than in the previous example due to symbolic conflict analysis. This clearly pays off now: C_6 propagates $\overline{b}$ and $\overline{c}$ at decision level zero, something that was not possible in Example 4, where our symbolic conflict analysis procedure was not used. The solver now decides on a, which propagates $\overline{x}$ and $\overline{y}$ due to C_1. Since we now have the strengthened version of C_5, we can propagate u. Unit propagation ends up finding a solution $\rho = (\overline{b}_6, \overline{c}_6, a^d, \overline{x}_1, \overline{y}_1, u_5, \overline{z}_2, w_4, \overline{v}_3)$ with value 10.

Sixth solution. Constraints C_4, C_5 and C_6 are strengthened again and become now:

$$\begin{array}{ll} C_4 : x + 2y + 3z + 4u + 5v + 6w \geq 11 & [\![1 + \delta]\!] \\ C_5 : 6\overline{c} + x + 2y + 3z + 4u + \overline{v} \geq 12 & [\![2 + \delta]\!] \\ C_6 : 4\overline{b} + 6\overline{c} + x + 2y + \overline{z} + \overline{v} \geq 13 & [\![3 + \delta]\!] \end{array}$$

After the propagation of $\overline{b}, \overline{c}$ at decision level zero, the solver decides on a and unit propagation produces the assignment $\rho = (\overline{b}_6, \overline{c}_6, a^d, \overline{x}_1, \overline{y}_1, u_5, z_5)$ with value 10. This now falsifies C_2 and conflict analysis is triggered. So z is popped from ρ, and the linear combination $3 \cdot C_2 + 1 \cdot C_5$ is performed, giving $C_{tmp} : 3\overline{b} + 6\overline{c} + x + 2y + u + \overline{v} \geq 12\ [\![2 + \delta]\!]$. Since it does not propagate at decision level zero, conflict analysis continues. Then u and $\overline{y}$ are popped and the linear combination $1 \cdot C_{tmp} + 2 \cdot C_1$ is performed, resulting in $C_7 : 2\overline{a} + 3\overline{b} + 6\overline{c} + \overline{x} + u + \overline{v} \geq 13\ [\![3 + \delta]\!]$, which propagates $\overline{a}$ at decision level zero. The solver backjumps to $\rho = (\overline{b}_6, \overline{c}_6, \overline{a}_7)$. Since nothing else propagates, the solver decides on v and unit propagation results in solution $\rho = (\overline{b}_6, \overline{c}_6, \overline{a}_7, v^d, \overline{w}_3, y_6, \overline{x}_1, u_4, \overline{z}_2)$ with value 11.

Seventh solution. Constraints C_4, C_5, C_6 and C_7 are strengthened again and become now:

$$\begin{array}{ll} C_4 : x + 2y + 3z + 4u + 5v + 6w \geq 12 & [\![1 + \delta]\!] \\ C_5 : 6\overline{c} + x + 2y + 3z + 4u + \overline{v} \geq 13 & [\![2 + \delta]\!] \\ C_6 : 4\overline{b} + 6\overline{c} + x + 2y + \overline{z} + \overline{v} \geq 14 & [\![3 + \delta]\!] \\ C_7 : 2\overline{a} + 3\overline{b} + 6\overline{c} + \overline{x} + u + \overline{v} \geq 14 & [\![3 + \delta]\!] \end{array}$$

These strengthenings allow the solver to compute the optimal solution only by unit propagation $\rho = (\overline{b}_6, \overline{c}_6, \overline{a}_7, \overline{x}_7, u_7, \overline{v}_7, \overline{z}_2, y_4, w_4)$, which has value 12.

Summing up, there are only three differences with respect to the underlying algorithm of a CDCL-based PB solver. First of all, PB constraints now have a symbolic representation of their degree. Note that this representation will always be of the form $[\![p \cdot \delta + q]\!]$, where $p, q \in \mathbb{Q}$. We cannot guarantee that they are natural numbers because of the use of the division rule that we will explain later. This symbolic representation is updated accordingly when the linear combination rule is applied. The second difference is that, when the first solution is found (with value V) a new constraint with symbolic part $[\![\delta + 1]\!]$ and degree $V + 1$ is added. The third difference is that when a new solution is found with value V, the degree of all constraints is updated. More concretely, if a constraint has as symbolic part $[\![p \cdot \delta + q]\!]$ its degree will become $p \cdot V + q$. The following theorem guarantees the correctness of our procedure.

▶ **Theorem 6.** *Let us consider S a set of PB constraints and $\sum_i o_i l'_i$ a PB objective function to maximize. If the symbolic constraint $\sum_i c_i l_i \geq [\![p \cdot \delta + q]\!]$ can be derived by a series of symbolic linear combination steps from S and the symbolic constraint $\sum_i o_i l'_i \geq [\![\delta + 1]\!]$, then $\sum_i c_i l_i \geq p \cdot V + q$ can be derived by a series of linear combination steps from S and $\sum_i o_i l'_i \geq V + 1$.*

4.2 Additional Benefits of Symbolic Conflict Analysis

In the previous section we showed the main benefit of symbolic conflict analysis: the strengthened constraints have stronger unit propagation capabilities and hence allow for a more efficient traversal of the search space. We now report on additional benefits that we obtain by using our symbolic conflict analysis procedure.

Computation of Upper Bounds. Somewhat surprisingly, a by-product of the symbolic procedure we have introduced is that we can easily compute upper bounds on the value of the objective function we want to maximize.

Again, the idea is best illustrated by revisiting Example 5. Let us consider constraint C_5 in its original form $6\overline{c} + x + 2y + 3z + 4u + \overline{v} \geq 2\ [\![2 + \delta]\!]$. Assume that a solution with value V is found. Our procedure would automatically update the degree of C_5 and it would become, removing its symbolic part for the sake of this reasoning, $6\overline{c} + x + 2y + 3z + 4u + \overline{v} \geq 2 + V$. The left-hand side of this inequality can at most be 17 (the sum of its coefficients). This means that if $2 + V \geq 18$ (i.e. $V \geq 16$) this constraint cannot be satisfied and hence the solver would conclude that there is no solution strictly larger than V. Hence the best objective can at most be 16. Thus, after second solution (with value 5) is found in Example 5, we can guarantee that the optimal solution is in the interval $[5, 16]$.

▶ **Theorem 7.** *Let us consider $\mathcal{P}$ a PB maximization problem consisting of a set S of PB constraints and an objective function $\sum_i o_i l'_i$. If we can derive by a series of symbolic linear combination steps a symbolic constraint $\sum_i c_i l_i \geq [\![p \cdot \delta + q]\!]$ from S and $\sum_i o_i l'_i \geq [\![\delta + 1]\!]$, then the value of the optimal solution to $\mathcal{P}$ is at most $\left\lfloor \frac{(\sum_i c_i) - q}{p} + 1 \right\rfloor$.*

Proof. Let O be the value of the optimal solution to $\mathcal{P}$. Let us consider the set of constraints $S' = S \cup \{\sum_i o_i l'_i \geq O\}$, which we know is satisfiable. Theorem 6 and the premises of this theorem guarantee that a series of linear combination steps exists that allow us to derive from S' the constraint $\sum_i c_i l_i \geq p \cdot (O - 1) + q$. Since linear combination produces logical consequences, the latter constraint must be satisfiable. Hence O must be such that $\sum_i c_i \geq p \cdot (O - 1) + q$, which is the same as $O \leq \frac{(\sum_i c_i) - q}{p} + 1$. Since O must be an integer value, we can take the floor of the right-hand side to obtain the desired bound. ◀

In Example 5, the previous theorem allows the symbolic conflict analysis procedure to produce an upper bound of 13 due constraint C_6, and a better upper bound of 12 due to C_7.

An interesting question is whether the derivation of upper bounds can trigger an early termination of the procedure because the best solution found matches the computed upper bound. We now prove that, in this situation, the constraint from which the upper bound is computed becomes trivially unsatisfiable after the corresponding strengthening step.

▶ **Theorem 8.** *Let us consider the constraint $C := \sum_i c_i l_i \geq [\![p \cdot \delta + q]\!]$ from which the upper bound θ is derived. If a solution with value θ is obtained, the corresponding strenghtened version of C is unsatisfiable.*

Proof. Due to theorem 7, we know that $\theta = \left\lfloor \frac{(\sum_i c_i)-q}{p} + 1 \right\rfloor$. If we find a solution with value θ we can strengthen C to obtain $\sum_i c_i l_i \geq p \cdot \left\lfloor \frac{(\sum_i c_i)-q}{p} + 1 \right\rfloor + q$. We can manipulate the degree of the constraint as follows:

$$p\left\lfloor \frac{(\sum_i c_i) - q}{p} + 1 \right\rfloor + q = p\left\lfloor \frac{(\sum_i c_i) - q}{p} \right\rfloor + p + q = p\left(\frac{(\sum_i c_i) - q}{p} - \varepsilon \right) + p + q = \sum_i c_i + p(1-\varepsilon)$$

for some $0 \leq \varepsilon < 1$. Hence, the strengthened constraint is $\sum_i c_i l_i \geq \sum_i c_i + p(1-\varepsilon)$ which is clearly unsatisfiable. ◀

Use in a Binary-Search Approach. The standard CDCL-based PB procedure we revisited in Example 4 strengthens the degree of the objective function every time a solution is found in order to find a better one. As mentioned before, this is sometimes known as Linear Search. However, we can use the well-known binary-search approach in which we always have an interval $[LB, UB]$ that includes the value of the optimal solution. If $LB = UB$ the optimal has been found. Otherwise we pick $M = (LB + UB)/2$ and recursively solve two problems, the first one in which a solution is sought in $[LB, M]$, and a second problem in which a solution is sought in $[M + 1, UB]$.

One of the main drawbacks of this approach is that constraints learned when solving the $[M + 1, UB]$ problem cannot be reused for the $[LB, M]$ problem, because they might have been derived by using the constraint $obj \geq M + 1$ that expresses that a solution of value at least $M + 1$ is needed, which is a not a valid constraint in the $[LB, M]$ problem.

However, this problem can easily be overcome in our symbolic conflict analysis procedure. First of all, we can easily identify the constraints that depend on $obj \geq M + 1$: the ones with a non-zero coefficient on δ in the symbolic part. But, even more importantly, we can even reuse those ones for solving the $[LB, M]$ problem by substituting δ by LB in the symbolic part and changing the degree of the constraint to that value.

4.3 Interaction with Other Inference Rules

As we have mentioned, our presentation in Algorithm 1 of the conflict analysis procedure used by CDCL-based PB solvers omitted some details in order to (i) ease its understanding and (ii) do not bias it towards any concrete solver. One of the ingredients that our presentation abstracts away is the use of other rules apart from linear combination. In particular, as explained in [15], the use of division and saturation allows solvers to process some problems exponentially faster. Apart from this important effect, they also allow the solver to deal with smaller coefficients in the constraints, which may become very large after a few linear combinations are applied. The recurrent use of infinite-precision arithmetic in a solver might have remarkable negative effects on performance.

The Division Rule. This rule can be smoothly adapted to be used in our symbolic conflict analysis procedure. Whenever a constraint $\sum_i a_i l_i \geq A$ is divided by a positive natural number α to obtain $\sum_i \lceil a_i/\alpha \rceil l_i \geq \lceil A/\alpha \rceil$ its symbolic part $[\![p \cdot \delta + q]\!]$ is also divided by the same constant to become $[\![\frac{p}{\alpha} \cdot \delta + \frac{q}{\alpha}]\!]$. This is the reason why coefficients in the symbolic part might be rational numbers. When a new solution with value V is found and the constraint needs to be strengthened, its degree will become the ceiling $\lceil \frac{p}{\alpha} \cdot V + \frac{q}{\alpha} \rceil$.

The Saturation Rule. Another common rule in PB solvers is saturation, which forces all coefficients of a constraint to be at most equal to its degree. In general, the use of this rule is incompatible with our symbolic procedure. This is easy to see if we consider again Example 5. The first lemma that is learned is $C_5 : 6\overline{c} + x + 2y + 3z + 4u + \overline{v} \geq 2\ [\![2 + \delta]\!]$. At this point, we could apply saturation to obtain $C_5 : 2\overline{c} + x + 2y + 2z + 2u + \overline{v} \geq 2\ [\![2 + \delta]\!]$. However, after the fourth solution (with value 9) is found, the procedure would update the degree of this constraint to be $2 + 9 = 11$, which would make the constraint unsatisfiable. Hence, the solver would wrongly conclude that the value of the optimal solution is 9.

We want to remark that saturation is still applicable to all constraints where δ has zero coefficient on the symbolic part, i.e., the ones that do not depend on the objective function. Regarding the rest of the constraints, there are a few possibilities: (i) apply saturation but remove the symbolic part or (ii) do not apply saturation, or (iii) apply saturation but store the original coefficients of the constraints in order to be used later when a strengthening step is performed.

5 Experimental Evaluation

In order to empirically evaluate the impact of our symbolic conflict analysis procedure, we have run experiments on the set of benchmarks that were used in the OPT-LIN (optimization problems with linear constraints) track of the 2024 Pseudo-Boolean Competition[2]. Since our solver does not accept big integers on the constraints, we limit our analysis to the 385 benchmarks that only use 32-bit integers. All experiments[3] were done on 3.3Ghz 16GB Intel Xeon E-2124 machines, setting a time limit of one hour per benchmark. We report our findings in what follows.

Potential and Overhead of Symbolic Conflict Analysis. After every new solution is found, except for the first one, our symbolic conflict analysis procedure performs a strengthening step: it changes the degree of all symbolic constraints. Hence, having a systematic low number of strengthening steps on all benchmarks would indicate that our procedure is rarely applicable. In order to confirm that this pessimistic scenario is not common we computed, for all 385 benchmarks, how many such steps are done within one hour. This information is summarized in the leftmost cumulative plot of Figure 1.

A point (x, y) in the plot means that there exist $y\%$ of the benchmarks for which at least x strengthening steps are applied. For example, around 50% of the benchmarks have at least 8 steps, one third of them has at least 15 steps, and 20% of the benchmarks have at least 24 steps. Hence, the important message one has to infer from the plot is that it is very common that a non-negligible number of strengthening steps are used. There are some instances, left out of the plot for readability, with an unusually large number of steps: 4 benchmarks with between 100 and 150 steps, 2 benchmarks between 151 and 200, and 2 benchmarks with more than 201, being 255 the maximum one. There are also extreme cases on the opposite direction: for about 18% of the benchmarks no strengthening step was applied. This is not a huge number but we believe it is large enough so that it is not negligible: doing expensive computations to later realize that our technique is not applicable at all would definitely slow down the solver on around 20% of the instances.

[2] `https://www.cril.univ-artois.fr/PB24/`
[3] Additional material can be found in `https://github.com/dearzhaorui/symbolic-conflict-analysis`.

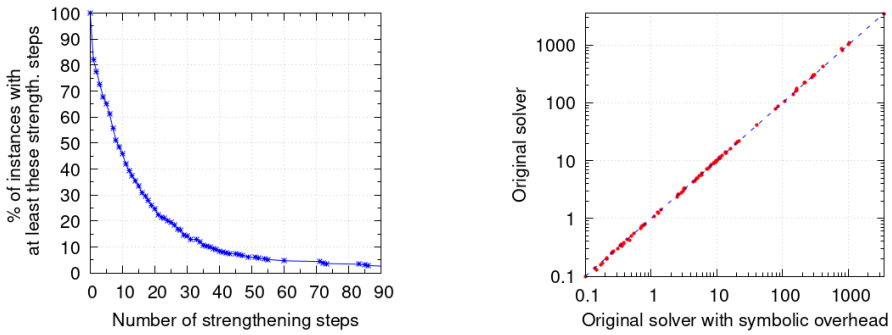


Figure 1 Left plot displays the percentage of benchmarks with at least a certain number of strengthening steps. The plot on the right shows the negligible overhead of the symbolic conflict analysis procedure.

Fortunately, the rightmost scatter plot of Figure 1 reveals how surprisingly small the overhead of symbolic conflict analysis is. For precisely assessing this overhead, we implemented our symbolic procedure on top of our solver: all constraints have a symbolic part; linear combination, saturation and division are applied taking into account this symbolic part but, after a new solution is found, the strengthened degree of all symbolic constraints is computed but not updated. That is, we pay all the overhead of our procedure but, since the constraints are unchanged, the solver has the same search behavior as its baseline version. The scatter plot compares the runtime in seconds of this modified system and our original solver. It is entirely obvious that the overhead caused by the symbolic procedure is negligible. We want to remark that this is the case independently of the package we use for infinite-precision rational numbers on the symbolic part: we tried GMP [14], Boost [5] and a custom package developed in our research group and no differences were observed.

Runtime Improvement. Let us now show that implementing symbolic conflict analysis on top of our solver produces important performance gains. The CDF plot on the left of Figure 2 shows how the addition of the symbolic conflict analysis procedure makes the solver able to solve more problems within any given time limit. For a time limit of 1 hour the solver is able to solve 6 more benchmarks up to optimality. The scatter plot on the right contains two parallel green lines delimiting speedups of 2x. We can see that despite there are concrete instances for which the symbolic solver is slower, they are mostly easy instances solved within 30 seconds, and the slowdown is never larger than 2x. On the other hand, there are several benchmarks for which the symbolic solver outperforms the baseline by very large speedup factors.

Finally, we want to compare our system with the state of the art. For this purpose we chose the latest version of RoundingSat [11, 9, 8]. Note that, in the optimization linear constraints category of the 2024 Pseudo-Boolean Competition, 8 of the 10 best performing solvers used RoundingSat, either as an oracle or combined with additional techniques. We ran RoundingSat in linear-search mode. Despite its full version, which combines linear and core-guided search is more powerful, the purpose of this comparison is limited to linear search. The CDF plot on Figure 3 shows how our original solver is slower than RoundingSat. This might be due to maturity of the implementation, different search strategies and, more importantly, concrete details in conflict analysis related to different weakening procedures, different uses of saturation and division, and termination criteria of the main loop as we explained in Section 3. However, the addition of symbolic conflict analysis bridges this gap

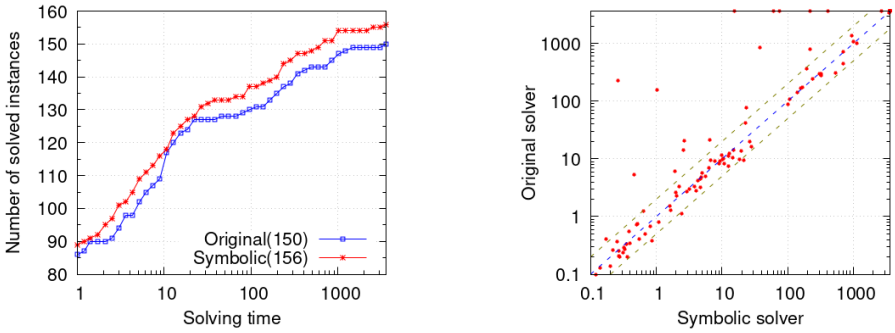


Figure 2 CDF and scatter plots comparing runtimes of our original solver with a version of it implementing symbolic conflict analysis.

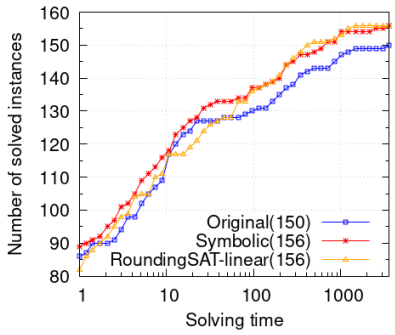


Figure 3 CDF plot comparing RoundingSat, our original solver and our implementation of symbolic conflict analysis on top of it.

and makes them behave more similarly. As we will mention, it is part of our future work to examine how symbolic conflict analysis can improve the linear-search component of other solvers, and RoundingSat is definitely a candidate.

In the optimization linear constraints category, 8 of the 10 best performing solvers did use RoundingSAT. That is, it was either used as an oracle or combined with additional techniques. Hence, it is at the core of almost every solver, except for SCIP.

Percentage of Symbolic Lemmas and Impact of Saturation. The next question we want to address is to determine how many lemmas have a non-constant symbolic part, that is, $[\![p \cdot \delta + q]\!]$ with $p \neq 0$. The left histogram in Figure 4 summarizes this information. For each of the 312 out of the overall 385 benchmarks that apply some strengthening step, whenever this step is performed we compute the percentage of PB constraints that have a non-constant symbolic part. Note that in our solver clauses have a particular treatment and they are not considered as PB constraints in this computation. When the execution finishes, we compute the average over all strengthening steps of these percentages. The histogram essentially represents this average. More concretely, the first bar indicates that for 34 benchmarks this average is zero. The second bar shows that for 65 benchmarks this average is in the interval $(0, 5]$; the third bar over the 10 label means that the average is in (5,10] for 13 benchmarks, and so on. As expected from the previous results on performance, the percentage of constraints with non-constant symbolic part is remarkable, making the strengthening step a powerful one.

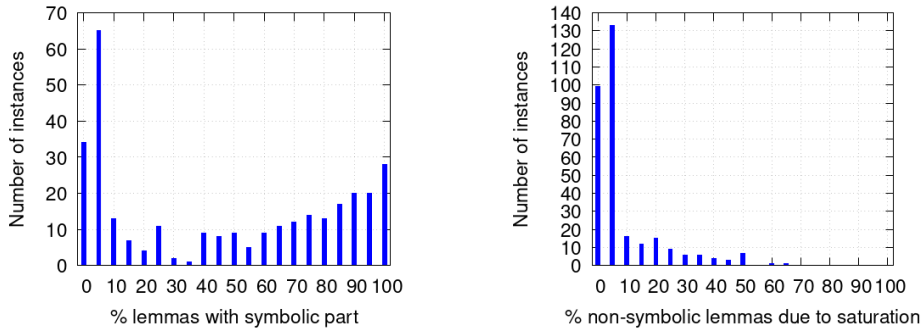


Figure 4 The left histogram shows the number of instances with a certain percentage of constraints with non-constant symbolic part. The right histogram show the percentage of lemmas that are non-symbolic due to an application of saturation.

Another question we wanted to answer is to discover the reasons why certain lemmas are non-symbolic (i.e. $p = 0$ in $[\![p \cdot \delta + q]\!]$). One obvious reason is that it might happen that the constraint bounding the objective function has not been used in the derivation of this lemma. Another possibility is the interaction between saturation and our symbolic procedure, as we explained in Section 4.3. In our implementation, whenever saturation is to be applied on a constraint with non-constant symbolic part, we apply it and remove its symbolic information. If this happens too often, it could hinder the power of our symbolic reasoning. The right histogram in Figure 4 answers this question in a positive way: this situation is not common at all. For example, the bar with height 15 over the 20 label means that there are 15 instances for which, if we only consider non-symbolic constraints, the percentage of them that are non-symbolic due to an application of saturation is in the interval $(15, 20]$. As we can see, the main reason for a constraint not being symbolic is not the application of saturation.

Upper Bounding. An interesting feature of our procedure is the computation of upper bounds. The number of times that our procedure improves the upper bound is significant for several benchmarks. In the left plot of Figure 5 we can see, for three concrete instances, the evolution of the upper and the lower bound. Since these instances are minimization problems, finding a new solution improves the upper bound, whereas our new bounding techniques improve the lower bound. We can see that both bounds are improved several times during the execution of the system. However, we want to note that there are of course instances for which the production of new bounds is not as frequent. This is observed in the right plot of Figure 5. We have considered all 87 benchmarks for which our solver timed out in one hour and such that after the last solution found, our bounding technique is able to reduce the size of the interval $[LB, UB]$. The plot displays the number of instances for which the size of the last interval produced by the solver, divided by the size of the interval after the last solution was found is equal to a certain number. A small number implies that our bounding technique is able to produce a significant reduction on the interval size since the last found solution. As we can see, although for half of the benchmarks only small reductions are achieved, for the other half it does have a beneficial effect, with different magnitudes depending on the benchmark. This shows that, even when the solver is not able to find further solutions, the bounding technique still makes progress in the estimation of the optimum.

Binary Search. It is well known that in PB optimization, binary search is generally outperformed by linear search. Unfortunately, according to our experiments, this is still the case even if one is able to reuse all lemmas from all problems, as we can do thanks to our symbolic procedure.

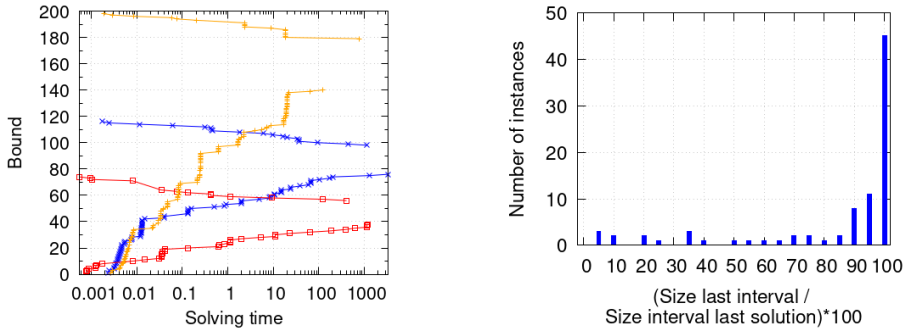


Figure 5 The left plot shows the evolution of the lower and upper bounds for three different instances. On the right, we can see the percentage of reduction of the interval $[LB, UB]$ after the last found solution.

6 Conclusions and Future Work

We have introduced a novel symbolic conflict analysis procedure for PB optimization. Despite being easy to implement and having no overhead, experimental results have shown that it is able to significantly improve the performance of our linear-search based solver. In addition, it has several benefits such as the computation of upper bounds and its use in binary search.

As future work, we plan to evaluate the impact of our technique on other PB solvers. Finding ways to improve our bound computation is also on our agenda, as well as developing proof-logging techniques to verify the correctness of the results given by this technique.

References

1 Elvira Albert, Pablo Gordillo, Alejandro Hernández-Cerezo, and Albert Rubio. A max-smt superoptimizer for EVM handling memory and storage. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 201–219. Springer, 2022. doi:10.1007/978-3-030-99524-9_11.

2 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.

3 Daniel Le Berre, Pierre Marquis, and Romain Wallon. On weakening strategies for PB solvers. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020 – 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 322–331. Springer, 2020. doi:10.1007/978-3-030-51825-7_23.

4 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.

5 The Boost C++ libraries. Available at https://www.boost.org/.

6 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. doi:10.1109/TCAD.2004.842808.

7 W. Cook, C. Coullard, and Gy. Turan. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18:25–38, 1987. doi:10.1016/0166-218X(87)90039-4.

8 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.

9 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.

10 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.

11 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.

12 Nick Feng and Fahiem Bacchus. Clause size reduction with all-uip learning. 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 28–45. Springer, 2020. doi:10.1007/978-3-030-51825-7_3.

13 Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing – SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006. doi:10.1007/11814948_25.

14 The GNU MP Bignum Library. Available at https://gmplib.org/.

15 Stephan Gocht, Jakob Nordström, and Amir Yehudayoff. On division versus saturation in pseudo-boolean solving. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1711–1718. ijcai.org, 2019. doi:10.24963/IJCAI.2019/237.

16 Ana Graça, João Marques-Silva, Inês Lynce, and Arlindo L. Oliveira. Efficient haplotype inference with combined CP and OR techniques. In Laurent Perron and Michael A. Trick, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 5th International Conference, CPAIOR 2008, Paris, France, May 20-23, 2008, Proceedings*, volume 5015 of *Lecture Notes in Computer Science*, pages 308–312. Springer, 2008. doi:10.1007/978-3-540-68155-7_28.

17 Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(2 & 3):297–308, 1985. doi:10.1016/0304-3975(85)90144-6.

18 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.

19 John N Hooker. Generalized resolution and cutting planes. *Annals of Operations Research*, 12:217–239, 1988.

20 Dejan Jovanovic and Leonardo Mendonça de Moura. Cutting to the chase – solving linear integer arithmetic. *J. Autom. Reasoning*, 51(1):79–108, 2013. doi:10.1007/s10817-013-9281-x.

21 Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Combining clause learning and branch and bound for maxsat. In 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 38:1–38:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CP.2021.38.

22 Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Boosting branch-and-bound maxsat solvers with clause learning. *AI Commun.*, 35(2):131–151, 2022. doi:10.3233/AIC-210178.

23 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.

24 António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided maxsat solving: A survey and assessment. *Constraints An Int. J.*, 18(4):478–534, 2013. doi:10.1007/S10601-013-9146-2.

25 Robert Nieuwenhuis and Albert Oliveras. On SAT Modulo Theories and Optimization Problems. In *Proceedings of SAT '06*, volume 4121 of *Lecture Notes in Computer Science*, pages 156–169. Springer, 2006. doi:10.1007/11814948_18.

26 Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Emma Rollon. Employee scheduling with sat-based pseudo-boolean constraint solving. *IEEE Access*, 9:142095–142104, 2021. doi:10.1109/ACCESS.2021.3120597.

27 Tobias Paxian and Bend Becker. Pacose: An Iterative SAT-based MaxSAT Solver. In Jeremias Berg, Matti Järvisalo, Ruben Martins, Andreas Niskanen, and Tobias Paxian, editors, *MaxSAT Evaluation 2024 : Solver and Benchmark Descriptions*, Department of Computer Science Series of Publications B, page 26. University of Helsinki, 2024. URL: http://hdl.handle.net/10138/584878.

28 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.

29 Albert Oliveras Robert Nieuwenhuis and Enric Rodríguez-Carbonell. Intsat: integer linear programming by conflict-driven constraint learning. *Optimization Methods and Software*, 2023. doi:10.1080/10556788.2023.2246167.

30 J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM, JACM*, 12(1):23–41, 1965. doi:10.1145/321250.321253.

31 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.

32 Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Pseudo-boolean optimization by implicit hitting sets. 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 51:1–51:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CP.2021.51.

33 Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Improvements to the implicit hitting set approach to pseudo-boolean optimization. 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 13:1–13:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.SAT.2022.13.

34 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.

35 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.

36 L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *International Conference on Computer-Aided Design, ICCAD '01*, pages 279–285, 2001.

SAT-Based CEGAR Method for the Hamiltonian Cycle Problem Enhanced by Cut-Set Constraints

Ryoga Ohashi
Kobe University, Kobe, Japan

Takehide Soh
Kobe University, Kobe, Japan

Daniel Le Berre
Univ. Artois, CNRS, UMR 8188 CRIL, Lens, France

Hidetomo Nabeshima
Yamanashi University, Kofu, Japan

Mutsunori Banbara
Nagoya University, Nagoya, Japan

Katsumi Inoue
National Institute of Informatics, Tokyo, Japan

Naoyuki Tamura
Kobe University, Kobe, Japan

Abstract

In this paper, we propose an enhancement to the SAT-based counterexample-guided abstraction refinement (CEGAR) approach for solving the Hamiltonian Cycle Problem (HCP). Many SAT-based methods for HCP have been proposed, including a CEGAR-based method that repeatedly solves a relaxed version of HCP strengthened by counterexamples. However, when the counterexample space – represented by the full set of subcycle partitions – is large, it becomes difficult to find a solution. To address this, we introduce *cut-set constraints* in the refinement step, replacing traditional subcycle blocking constraints. Our evaluation shows that these cut-set constraints achieve equal or better reduction in the counterexample space, making it easier to find valid solutions. We further assessed performance using all 1001 instances from the FHCP challenge set and confirmed that the proposed method solved 937 instances within 1800 seconds, outperforming both the existing eager and CEGAR encodings (which solved at most 666 instances). This demonstrates the effectiveness of incorporating cut-set constraints into SAT-based CEGAR approaches.

2012 ACM Subject Classification Computing methodologies → Discrete space search

Keywords and phrases Hamiltonian Cycle Problem, SAT Encoding, CEGAR

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.24

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.15621774`

Funding This work was financially supported by JSPS KAKENHI (23K11047), and by ROIS NII Open Collaborative Research 2024 (24FP04).
Takehide Soh: JSPS KAKENHI Grant Number 23K11047.
Mutsunori Banbara: JSPS KAKENHI Grant Number 25K03097.
Katsumi Inoue: JSPS KAKENHI Grant Number JP25K03190, JST CREST Grant Number JPMJCR22D3.

1 Introduction

The *Hamiltonian Cycle Problem (HCP)* asks whether a given graph contains a cycle that visits each node exactly once. HCP is one of the 21 NP-complete problems identified by Karp in 1972 [10]. Due to its close link with the Traveling Salesman Problem (TSP), efficient methods for HCP have remained a key research focus. As a result, *FHCP Challenge* [7] was recently held as a year-long challenge aimed at tackling particularly difficult HCP instances.

The *Satisfiability (SAT) Problem* is another NP-complete problem, deciding whether there exists an assignment of truth values to propositional variables that makes a given logical formula true. Despite the theoretical complexity of SAT, SAT solvers are very efficient



© Ryoga Ohashi, Takehide Soh, Daniel Le Berre, Hidetomo Nabeshima, Mutsunori Banbara, Katsumi Inoue, and Naoyuki Tamura;
licensed under Creative Commons License CC-BY 4.0



in practice [2, 6] on a wide range of combinatorial problems reduced to SAT, including HCP [11, 14, 13, 15, 8, 16]. SAT-based methods for solving HCP usually depend on degree constraints and connectivity constraints. Various SAT encodings for these constraints have been studied. Recently, binary adder encoding [15] showed good performance for selected 7 instances of the FHCP Challenge. Combined with degree constraints, it assigns a total order to each node connected in a solution graph, and a binary adder manages the order. Subsequent work [8] utilizes the Chinese remainder theorem (CRT) to further shrink the number of clauses generated by binary adder encoding. Another approach is vertex elimination [16], which avoids no-sub-cycle constraints unlike other methods. However, these methods suffer from scalability issues or excessive clause generation on challenging FHCP instances, making a more efficient SAT-based method a key challenge.

To overcome the limitations of existing SAT-based methods, in this paper, we revisit the SAT-based CEGAR method [13]. This approach uses a SAT solver to solve a relaxed version of the HCP that consists solely of degree constraints. If the obtained solution contains subcycles (counterexamples), the method iteratively adds subcycle blocking constraints to eliminate them; otherwise, it returns the obtained solution as a Hamiltonian cycle. This approach has the advantage of not needing to encode upfront the connectivity constraint, which tends to result in a large number of clauses. However, for difficult instances, the issue is that the number of counterexamples found is huge, preventing the SAT solver from finding a Hamiltonian cycle. The main idea of the proposed method is to use *cut-sets*, which are sets of edges that connect subcycles to other subcycles, and add constraints to include these edges in the solution rather than relying on traditional methods that only add constraints to eliminate the detected subcycles. This improvement allows for the elimination of more subcycles compared to previous methods, leading to better performance. To evaluate the effectiveness of our proposed method, we conducted experiments using all 1001 benchmark instances from the FHCP Challenge. We also compared our approach with recently proposed SAT-based methods [13, 8, 16] to further assess its performance.

2 Preliminaries

Hamiltonian cycle problem (HCP). Let $G = (V, E)$ be a graph where V is a set of n nodes and E is a set of edges. Let C be a simple cycle in G. We also denote a cycle by a sequence of nodes $(v_{i_1}, v_{i_2}, \ldots, v_{i_k})$, where $v_{i_j} \in V$ are all distinct and $\{v_{i_j}, v_{i_{j+1}}\} \in E$ for all $1 \leq j < k$, with $\{v_{i_k}, v_{i_1}\} \in E$. We also sometimes treat a cycle C as a set of edges. We use $|C|$ as the size of the cycle C and use $V(C)$ as the set of nodes in a cycle C, i.e., the set of nodes consisting of the cycle C. We also denote by G_C the subgraph of G induced by $V(C)$, i.e., $G_C = G[V(C)]$. The *Hamiltonian cycle problem* (HCP) is the problem of finding a cycle C s.t. $V(C) = V$.

Constraints of HCP. Let $G' = (V, E')$ be a subgraph (or a solution graph) of G where $E' \subseteq E$. Let x_{ij} $(i \neq j)$ be a Boolean variable for each edge $\{i, j\} \in E$, which is equal to 1 when $\{i, j\} \in E'$ is true. Then, constraints of HCP can be written as follows: $\Psi_{DEG} := \bigwedge_{i=1}^{n} \sum_{\{i,j\} \in E} x_{ij} = 2$ and $\Psi_{CON} := \bigwedge_{\substack{S \subset V \\ 2 \leq |S| \leq n-2}} \sum_{i,j \in S} x_{ij} \leq |S| - 1$. The *degree constraints* Ψ_{DEG} force that, for each node, its degree is exactly two in G'. Consequently, it forces each node to belong to a subcycle in G'. The *connectivity constraint* Ψ_{CON} prohibits subcycles, i.e., cycles on proper subsets of V. As mentioned above, compared to degree constraints (which can be encoded with a linear number of clauses to the number of nodes), connectivity constraints are more expensive in SAT encoding; in the worst case of a naive encoding, an exponential number of clauses is necessary.

Algorithm 1 SAT-based CEGAR for HCP.

Input: Graph $G = (V, E)$
Output: Hamiltonian cycle (if exists), or "No"

1: $\Psi :=$ `DegreeConstraint`(G)
2: **while** Ψ is satisfiable **do**
3: $P :=$ Decode the model of Ψ found ▷ P is a subcycle partition of G
4: **if** $|P| == 1$ **then**
5: **return** P ▷ Hamiltonian cycle found
6: **end if**
7: $\Psi := \Psi \wedge$ `RefinementConstraints`(P)
8: **end while**
9: **return** "No"

Subcycle Partition. Consider a set of cycles $P = \{C_1, C_2, \ldots, C_k\}$. We say P is a *subcycle partition* of a given graph $G = (V, E)$ iff $V(C_1) \cup V(C_2) \cup \cdots \cup V(C_k) = V$ and $V(C_i) \cap V(C_j) = \varnothing$ for all $1 \leq i < j \leq k$. In other words, each node of G belongs to exactly one cycle in the set P, and together, these cycles cover every node in the graph without overlap. Clearly, a subcycle partition P contains a Hamiltonian cycle iff $|P| = 1$. Let Π be a function that maps a graph G to the set of all its subcycle partitions. Define $\Pi^{>1}$ as another function that maps a graph G to all its subcycle partitions excluding Hamiltonian cycles, i.e., $\Pi^{>1}(G) = \{\, P \in \Pi(G) \mid |P| > 1\}$.

Cut-set. A *cut* (S, T) is a partition of the node set V of a graph $G = (V, E)$ into two disjoint subsets S and T. The *cut-set* of a cut (S, T) is defined as $\{(u, v) \in E \mid u \in S, v \in T\}$, which is the set of edges having one endpoint in S and the other endpoint in T.

3 Existing SAT-based CEGAR for HCP [13]

Unlike typical SAT-based methods, SAT-based *Counterexample-Guided Abstraction Refinement* (CEGAR) relaxes constraints and incrementally uses a SAT solver on a partial HCP encoding, instead of fully encoding connectivity into CNF at once.

The algorithm of SAT-based CEGAR for HCP is presented in Algorithm 1. In Line 1, the formula Ψ is initialized with the degree constraints Ψ_{DEG}. Line 2 checks satisfiability repeatedly. If satisfiable, Line 3 decodes a subcycle partition P from the obtained model. If Line 4 detects that P is a single cycle, Line 5 immediately returns it as a Hamiltonian cycle. Otherwise, Line 7 adds the refinement constraints derived from P to refine Ψ, and Line 8 resumes the checking. Finally, when no further solution can be found, Line 9 concludes that no Hamiltonian cycle exists.

If we initialize the abstraction using only degree constraints, the counterexamples are always a subcycle partition $P = \{C_1, C_2, \ldots, C_m\}$ of G. Then, in the existing SAT-based CEGAR [13], `RefinementConstraints`(P) is implemented by `SubCycleBlocking`(P) as follows:

$$\texttt{SubCycleBlocking}(P) := \bigwedge_{C \in P} \bigvee_{\{u,v\} \in C} \neg x_{u,v} \tag{1}$$

An illustration of a possible counterexample is shown in Figure 1. In this example, the subcycle partition consists of $C_1 = (1, 2, 3, 10, 11, 12)$ and $C_2 = (4, 5, 6, 7, 8, 9)$. For this

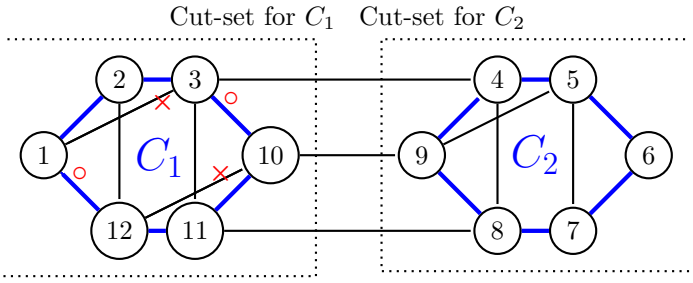


Figure 1 A counterexample consists of the subcycle partition $P = \{C_1, C_2\}$ where $C_1 = (1, 2, 3, 10, 11, 12)$ and $C_2 = (4, 5, 6, 7, 8, 9)$. The dashed line represent the cut-sets between the set of nodes $V(C_i)$ (i.e., S) and the rest of the graph $V \setminus V(C_i)$ (i.e., T), as explained in Section 4.1. The $\times$ marks indicate the deleted edges {3,1} and {12,10}, while the $\circ$ marks indicate the added edges {3,10} and {1,12} according to the `MergeCycles` operation explained in Section 4.3.

counterexample, the existing SAT-based CEGAR refines constraints Ψ by two blocking clauses. In the case of the given example, the following two clauses are generated.

$$\neg x_{1,2} \vee \neg x_{2,3} \vee \neg x_{3,10} \vee \neg x_{10,11} \vee \neg x_{11,12} \vee \neg x_{12,1}$$
$$\neg x_{4,5} \vee \neg x_{5,6} \vee \neg x_{6,7} \vee \neg x_{7,8} \vee \neg x_{8,9} \vee \neg x_{9,4} \tag{2}$$

Note that, even in the worst case, we do not always need to block all subcycle partitions since each subcycle is forbidden independently. For instance, if we refine the constraints for the subcycle partition $\{C_1, C_2\}$, then we do not need to care $\{C_1, (4, 8, 9), (5, 6, 7)\}$ since C_1 cannot appear after the first refinement. And it is not necessary to block all subcycles, e.g. $(2, 3, 11, 12)$ since the remaining nodes cannot construct any subcycle partition.

From another perspective, this CEGAR approach can be regarded as a method for turning the following eager encoding into a lazy encoding.

$$\Psi_{SCB} := \bigwedge_{P \in \Pi^{>1}(G)} \bigwedge_{C \in P} \bigvee_{\{u,v\} \in C} \neg x_{u,v} \tag{3}$$

Instead of the connectivity constraints Ψ_{CON} presented in Section 2, the conjunction $\Psi_{DEG} \wedge \Psi_{SCB}$ ensures connectivity, forcing any solution graph to form a Hamiltonian cycle.

4 Proposed SAT-based CEGAR Enhanced by Cut-Set Constraints

This paper proposes an improvement to the existing SAT-based CEGAR method. While the basic algorithm remains the same, we modify how refinement constraints are added.

4.1 Cut-Set Constraints

In the following, we will denote the cut-set of (S, T) simply as $\Delta(S)$. Our idea is to replace the base constraint Ψ_{SCB} shown in the equation (3) by the following constraint[1]:

$$\Psi_{CSF} := \bigwedge_{P \in \Pi^{>1}(G)} \bigwedge_{C \in P} \bigvee_{\{u,v\} \in \Delta(V(C))} x_{u,v} \tag{4}$$

Then, the following proposition holds.

[1] Cut-set based constraints have been studied in the context of the TSP solving (e.g., [4]). We adapt these ideas to the setting of SAT-based CEGAR for HCP.

▶ **Proposition 1.** Any model of $\Psi_{DEG} \wedge \Psi_{CSF}$ forms a Hamiltonian cycle of G.

Proof. The constraint Ψ_{DEG} ensures that any solution graph can be decomposed into a subcycle partition of G. Meanwhile, the constraint Ψ_{CSF} ensures that for each subcycle in the subcycle partition, there is at least one edge connecting it to its complement. This forbids any subcycle partition in $\Pi^{>1}(G)$ that is not a single cycle. Consequently, any model of $\Psi_{DEG} \wedge \Psi_{CSF}$ necessarily constructs a Hamiltonian cycle. ◀

We then propose to implement the function `RefinementConstraints`(P) of Algorithm 1 by `CutSetForcing`(P) as follows (see also Figure 1 for an example):

$$\texttt{CutSetForcing}(P) := \bigwedge_{C \in P} \bigvee_{\{u,v\} \in \Delta(V(C))} x_{u,v} \tag{5}$$

For the same counterexample $P = \{C_1, C_2\}$ shown in Figure 1, the following clause is generated by `CutSetForcing`(P).

$$x_{3,4} \vee x_{9,10} \vee x_{8,11} \tag{6}$$

Note that there is a single clause since this example has the partition $|P| = 2$, and there is only one cut-set constraint for two subcycles. However, in general, we need the $|P|$ clauses as same as `SubCycleBlocking`(P).

4.2 Properties of Cut-Set Constraints

The advantage of using the proposed cut-set constraints instead of the existing subcycle blocking constraints is its ability to better prune the counterexample space $\Pi^{>1}(G)$. In general, the following proposition holds.

▶ **Proposition 2.** Given a graph $G = (V, E)$ and a subcycle partition P, the number of subcycle partitions that `CutSetForcing`(P) prunes from $\Pi^{>1}(G)$ is *equal to or greater than* the number that `SubCycleBlocking`(P) prunes.

Proof. Both clauses commonly forbid individual subcycles contained in P. Thus, all subcycle partitions that contain those subcycles are pruned from $\Pi^{>1}$, which can be represented by $\Theta_1 = \{P' \in \Pi^{>1}(G) \mid \{C\} \subseteq P' \wedge C \in P\}$. Clearly, `SubCycleBlocking`$(P)$ cannot prune subcycle partitions other than Θ_1. On the other hand, `CutSetForcing`(P) additionally prunes $\Theta_2 = \{P' \in \Pi^{>1}(G) \mid P'' \subseteq P' \wedge P'' \in \bigcup_{C \in P} (\Pi(G_C) \setminus \{C\})\}$ where G_C is a subgraph of G induced by C. ◀

To illustrate the example, we assign the following symbols $C_1, \ldots, C_{12}$ to the subcycles in Figure 1: $C_1 = (1, 2, 3, 10, 11, 12)$, $C_2 = (4, 5, 6, 7, 8, 9)$, $C_3 = (1, 2, 3)$, $C_4 = (10, 11, 12)$, $C_5 = (1, 2, 12)$, $C_6 = (3, 10, 11)$, $C_7 = (4, 8, 9)$, $C_8 = (5, 6, 7)$, $C_9 = (1, 2, 12, 10, 11, 3)$, $C_{10} = (1, 2, 12, 11, 10, 3)$, $C_{11} = (1, 2, 3, 11, 10, 12)$, $C_{12} = (4, 9, 5, 6, 7, 8)$.

Again, suppose that $P = \{C_1, C_2\}$ is obtained as a counterexample. Then, two refining clauses in formulae (2) and (6) commonly forbid the following subcycle partitions, which include an element of $P = \{C_1, C_2\}$. The pruned partitions Θ_1 become as follows:

$$\{\{C_1, C_2\}, \{C_1, C_7, C_8\}, \{C_1, C_{12}\}, \{C_2, C_3, C_4\}, \{C_2, C_5, C_6\}, \{C_2, C_9\}, \{C_2, C_{10}\}, \{C_2, C_{11}\}\}$$

In addition, the proposed constraint in formula (6) prohibits the following subcycle partitions, which include an element of $\{\{C_3, C_4\}, \{C_5, C_6\}, \{C_9\}, \{C_{10}\}, \{C_{11}\}\} = \Pi(G_{C_1}) \setminus \{C_1\}$ or $\{\{C_7, C_8\}, \{C_{12}\}\} = \Pi(G_{C_2}) \setminus \{C_2\}$ – subcycle partitions of G induced by C_1 or C_2.

$$\begin{aligned}\Theta_2 = &\{\{C_3, C_4, C_7, C_8\}, \{C_5, C_6, C_7, C_8\}, \{C_9, C_7, C_8\}, \{C_{10}, C_7, C_8\}, \{C_{11}, C_7, C_8\},\\ &\{C_3, C_4, C_{12}\}, \{C_5, C_6, C_{12}\}, \{C_9, C_{12}\}, \{C_{10}, C_{12}\}, \{C_{11}, C_{12}\}\}\end{aligned}$$

Algorithm 2 Procedure of Merging Subcycles.

Input: Subcycle Partition $P = \{C_1, C_2, \ldots, C_m\}$
Output: Smaller Subcycle Partition $P = \{C'_1, C'_2, \ldots, C'_{m'}\}$

1: **while** there exists a pair $(C_i, C_j) \in P$ such that $\texttt{MergeCycles}(C_i, C_j) \neq ()$ **do**
2: $\quad C_{\text{merged}} := \texttt{MergeCycles}(C_i, C_j)$
3: $\quad P := (P \setminus \{C_i, C_j\}) \cup \{C_{\text{merged}}\}$
4: **end while**
5: **return** P

When subcycles in a counterexample contain multiple subcycle partitions, Θ_2 becomes exponentially large. The reason is that existing methods prune only subcycle partitions containing the counterexample subcycle. In contrast, the proposed method enforces a cut-set, which prunes *all* subcycle partitions enclosed by the subgraph induced by that subcycle. In the above case, the difference arises because the subgraph induced by C_1 has multiple subcycle partitions – C_1, C_9, C_{10}, C_{11}, C_3, C_4, and C_5, C_6 – similarly for C_2.

In other words, if the subgraph induced by the counterexample subcycle does not have multiple subcycle partitions, then $\Theta_2 = \varnothing$, and there is no difference between the existing approach and the proposed approach. In order to use the pruning capability of the proposed method – i.e., to increase the cardinality of Θ_2 – we perform subcycle merging, as described in the next section.

4.3 Merging Subcycles for Improving Runtime

The previous section suggests that, when the induced subgraph of a discovered subcycle has other subcycle partitions, it prunes a larger number of subcycle partitions. We thus introduce a post-processing method that merges subcycles to utilize the advantage of the proposed refinement constraints. The post-processing procedure is shown in Algorithm 2. Its input is a subcycle partition, and it returns a smaller partition if possible. This algorithm searches for a pair of cycles that can be merged using a function `MergeCycles`. If such a pair is found, the algorithm replaces them with the merged cycle. This process is repeated until no mergeable pair remains. Let $C_i = (v_1, v_2, \ldots, v_k)$ and $C_j = (u_1, u_2, \ldots, u_l)$ be two subcycles to merge. Like the 2-opt method in TSP [3], `MergeCycles` are realized as follows.

$$\texttt{MergeCycles}(C_i, C_j) := \begin{cases} \texttt{Swap}\big(C_i, C_j, \{s, t\}, \{s+1, t+1\}\big), & \text{if } \exists\, s, t \text{ such that } v_s \in V(C_i),\ u_t \in V(C_j), \\ & \{v_s, u_t\} \in E,\ \{v_{s+1}, u_{t+1}\} \in E, \\ \texttt{Swap}\big(C_i, C_j, \{s, t+1\}, \{s+1, t\}\big), & \text{if } \exists\, s, t \text{ such that } v_s \in V(C_i),\ u_t \in V(C_j), \\ & \{v_s, u_{t+1}\} \in E,\ \{v_{s+1}, u_t\} \in E, \\ (), & \text{otherwise.} \end{cases}$$

Here, `Swap` forms a single cycle combining C_i and C_j by replacing the edges $\{v_s, v_{s+1}\}$ and $\{u_t, u_{t+1}\}$ with two given edges. For example, suppose that the subcycles $(1, 2, 3)$ and $(10, 11, 12)$ are found in the instance graph shown in Figure 1. Then we can merge the two subcycles by replacing the edges $\{3, 1\}$ and $\{12, 10\}$ with $\{3, 10\}$ and $\{1, 12\}$ that will create the larger subcycle $(1, 2, 3, 10, 11, 12)$.

4.4 Incorporating Existing Techniques for Efficiency

Conversion to Directed Graph. For simplicity, we have explained the method using undirected graphs so far. However, the method can be naturally extended to handle directed graphs. In fact, as shown in previous work [13], experiments suggest that applying the method to directed graphs yields better performance. Therefore, in the subsequent sections, we use the directed-graph version of the method. When an undirected graph $G = (V, E)$ is given, we introduce a set of auxiliary arcs $A = \{(i, j),\ (j, i) \mid \{i, j\} \in E\}$. And, the degree constraint becomes $\Psi_{DEG} = \bigwedge_{i=1}^{n} \sum_{(i,j)\in A} x_{ij} = 1 \wedge \bigwedge_{j=1}^{n} \sum_{(i,j)\in A} x_{ij} = 1$. The refinement constraint becomes as follows. Here, Δ^{in} and Δ^{out} are incoming and outgoing arcs in the cut-set, respectively.

$$\texttt{CutSetForcing}(P) := \bigwedge_{C_i \in P} \left(\bigvee_{(u,v)\in\Delta^{in}(C_i)} x_{u,v} \wedge \bigvee_{(u,v)\in\Delta^{out}(C_i)} x_{u,v} \right) \tag{7}$$

Hint Constraints. Because a large number of cycles of two nodes occur when the undirected graph is converted to a directed graph, an additional constraint is introduced to prevent them [13]. We refer to it as "2loop." Furthermore, symmetry is broken by directing the edges to uniquely determine the orientations of the cycles [8]. We refer to it as "asym."

5 Experiments

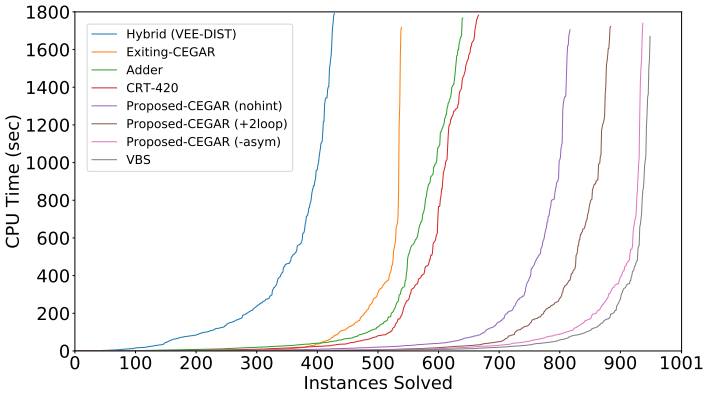
5.1 Set Up

Computer. Core i5 12400 (2.5GHz) CPU with 6 cores, 64GB of RAM. A single thread is used for each run in the experiments. Timelimit is 1800 seconds per instance.

Benchmarks. Recent studies [15, 8] used 7 instances selected from the FHCP Challenge Set[2] [7]. In contrast, we used all 1001 instances provided in the benchmark set to conduct comprehensive experiments.

Compared Methods. We compared the proposed method with an existing CEGAR approach [13] (Existing-CEGAR), as well as three recently proposed SAT encodings (Adder [15], CRT-420 [8], Hybrid (VEE-DIST) [16], respectively) for the Hamiltonian cycle problem. For Adder and CRT-420, we used an implementation available on GitHub[3]. For VEE-DIST, we use the recent version of Picat[4] (v3.8 with Kissat). Note that Existing-CEGAR is implemented using Rustsat [9] as well as the proposed CEGAR. In addition to those tools, we also tried to compare another implementation of the existing CEGAR[5], but could not run it reliably (418 instances were aborted by OS signals). Therefore, we do not include its performance.

[2] https://sites.flinders.edu.au/flinders-hamiltonian-cycle-project/fhcp-challenge-set/
[3] https://github.com/marijnheule/ChineseRemainderEncoding
[4] https://picat-lang.org/
[5] https://github.com/kfazekas/incremental-examples



Hybrid [16]	428
E. CEGAR [13]	539
Adder [15]	640
CRT-420 [8]	666
P. CEGAR (nohint)	817
P. CEGAR (+2loop)	884
P. CEGAR (-asym)	937
VBS	949

Figure 2 Cactus Plot of Proposed Method and SAT-based Prior Work.

Implementation. To encode the degree constraints in CNF, we have used sequential counter encoding [12]. We have implemented the proposed method on Rustsat. The implementation is available [6]. We use CaDiCaL [1] for all methods except Picat.

5.2 Result

Ablation study of the proposed approaches. We first evaluate the three acceleration options "2loop," "asym," "merge" described in Section 4.3 and Section 4.4. The number of solved instances is shown in the table below where the symbol + denotes "nohint" plus one additional option, and the symbol - denotes "all" minus some individual option. The Virtual Best Solver (VBS) is a hypothetical solver which represents the best possible performance among all evaluated options.

	# Ins.	nohint	+2loop	+asym	+merge	-merge	-asym	-2loop	all	VBS
Total	1001	817	884	822	858	895	**937**	863	936	948

The results show that "2loop" and "merge" are essential, and "-asym" performs best.

Comparisons with Existing SAT-based Methods. Figure 2 presents a cactus plot, where each approach is represented by a curve. On the x-axis, we have the number of instances solved within a specified time limit, and on the y-axis, the required timelimit for solving those instances is shown. Among individual approaches, the proposed CEGAR variants solved more instances (937) within the timeout. Meanwhile, the existing CEGAR method solved significantly fewer instances than the proposed approaches. Because the existing CEGAR method includes "2loop," the difference compared to "+2loop" can be viewed as the impact of the proposed cut-set constraints. Note that CRT-420 is also a type of abstraction method and can occasionally return counterexamples. For this figure, we plotted only the problems for which a correct solution was returned, totaling 666. Even if we consider counterexamples of CRT-420 as valid solutions, the number of instances solved would be 733, whereas the proposed method successfully solved more instances.

[6] `https://doi.org/10.5281/zenodo.15621774`

6 Conclusion

In this paper, we propose an enhancement to the existing SAT-based counterexample-guided abstraction refinement (CEGAR) approach to solving the Hamiltonian cycle problem (HCP). Our key contribution is to replace the conventional constraints for subcycle blocking with cut-set constraints. By doing so, we can prune a larger portion of the counterexample space, especially in cases where subcycles contain nested subpartitions. In addition, we introduced an acceleration technique, merging subcycles, which further speeds up the proposed method. In our experiments, we confirmed that the proposed method exhibited improved performance compared to existing CEGAR and eager encoding methods. While the existing approach solved 666 out of 1001 FHCP challenge set instances, the proposed method solved 937 instances within 1800 seconds. Future work includes a more efficient implementation using IPASIR-UP [5] and using parallelization to achieve further speed-ups.

References

1 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.

2 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.

3 Georges A Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

4 George B. Dantzig, Delbert R. Fulkerson, and Selmer M. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954. doi:10.1287/OPRE.2.4.393.

5 Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. Satisfiability modulo user propagators. *J. Artif. Intell. Res.*, 81:989–1017, 2024. doi:10.1613/JAIR.1.16163.

6 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artif. Intell.*, 301:103572, 2021. doi:10.1016/J.ARTINT.2021.103572.

7 Michael Haythorpe. FHCP Challenge Set: The first set of structurally difficult instances of the Hamiltonian cycle problem, 2019. arXiv:1902.10352.

8 Marijn J. H. Heule. Chinese remainder encoding for Hamiltonian cycles. In Chu-Min Li and Felip Manyà, editors, *Proc of 24th International Conference on Theory and Applications of Satisfiability Testing (SAT 2021)*, volume 12831 of *LNCS*, pages 216–224. Springer, 2021. doi:10.1007/978-3-030-80223-3_15.

9 Christoph Jabs. RustSAT: A library for SAT solving in Rust. In *Proc of 28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025)*, volume 341 of *LIPIcs*, pages 15:1–15:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.SAT.2025.15.

10 Richard M. Karp. *Reducibility among Combinatorial Problem*, pages 85–103. Springer, Boston, MA, 1972. doi:10.1007/978-1-4684-2001-2_9.

11 Steven D. Prestwich. SAT problems with chains of dependent variables. *Discret. Appl. Math.*, 130(2):329–350, 2003. doi:10.1016/S0166-218X(02)00410-9.

12 Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In Peter van Beek, editor, *Proc. of 10th International Conference on Principles and Practice of Constraint Programming (CP 2005)*, pages 827–831, Berlin, Heidelberg, 2005. Springer. doi:10.1007/11564751_73.

13 Takehide Soh, Daniel Le Berre, Stéphanie Roussel, Mutsunori Banbara, and Naoyuki Tamura. Incremental SAT-based method with native Boolean cardinality handling for the Hamiltonian cycle problem. In Eduardo Fermé and João Leite, editors, *Proc. of 14th European Conference on Logics in Artificial Intelligence*, pages 684–693. Springer, 2014. doi:10.1007/978-3-319-11558-0_52.

14 Miroslav N. Velev and Ping Gao. Efficient SAT techniques for absolute encoding of permutation problems: Application to Hamiltonian cycles. In Vadim Bulitko and J. Christopher Beck, editors, *Proc. of 8th Symposium on Abstraction, Reformulation, and Approximation (SARA 2009)*. AAAI, 2009. URL: http://www.aaai.org/ocs/index.php/SARA/SARA09/paper/view/837.

15 Neng-Fa Zhou. In pursuit of an efficient SAT encoding for the Hamiltonian cycle problem. In Helmut Simonis, editor, *Proc. of 26th International Conference on Principles and Practice of Constraint Programming (CP 2020)*, volume 12333 of *Lecture Notes in Computer Science*, pages 585–602. Springer, 2020. doi:10.1007/978-3-030-58475-7_34.

16 Neng-Fa Zhou. Encoding the Hamiltonian cycle problem into SAT based on vertex elimination (short paper). In Paul Shaw, editor, *Proc. of 30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 307 of *LIPIcs*, pages 40:1–40:8. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CP.2024.40.

QRP+Gen: A Framework for Checking Q-Resolution Proofs with Generalized Axioms

Mark Peyrer
Institute for Symbolic Artificial Intelligence, Johannes Kepler University, Linz, Austria

Martina Seidl
Institute for Symbolic Artificial Intelligence, Johannes Kepler University, Linz, Austria

Abstract

Q-resolution is a proof system for quantified Boolean formulas (QBFs) that forms the foundation for search-based QBF solvers with clause and cube learning. To derive stronger clauses and cubes, Q-resolution was extended with so-called generalized axioms. The derivation of such generalized axioms relies on solving oracles that could be, for example, SAT solvers or even QBF solvers.

While the correctness of results obtained with classical QCDCL-based solving can be efficiently certified by an independent checker, until now, proof generation had to be turned off to benefit from generalized axioms. Consequently, the results obtained with reasoning under generalized axioms could not be certified independently. To overcome this restriction, we present `QRP+Gen`, a novel framework to automatically generate and check Q-resolution proofs that contain generalized axioms. To this end, we extended the Q-resolution format `QRP` such that all necessary information is included to verify the correctness of generalized axioms. Our extension allows to integrate certificates produced by any oracle which can produce automatically checkable proofs. Furthermore, we developed a proof checker that orchestrates the proof checking of the core Q-resolution proof and the proofs produced by the oracles. As a case study, we equipped the search-based QBF solver `DepQBF` with proof-producing oracles for the SAT-based techniques trivial truth and trivial falsity.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Automated Reasoning, Quantified Resolution Proof, Generalized Axioms

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.25

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.16307432`

Funding Parts of this work have been supported by the LIT AI Lab funded by the state of Upper Austria and by the Austrian Science Fund (FWF) [10.55776/COE12].

1 Introduction

The solving landscape for quantified Boolean formulas (QBFs) is very heterogeneous [17]. In contrast to SAT, which has conflict-driven clause learning (CDCL) [18] as predominant solving paradigm, the situation is less clear for QBFs. Besides QCDCL, the CDCL variant for QBF, expansion- and abstraction-based approaches have proven to be very powerful and of orthogonal strength [11]. To combine different approaches, different directions are taken: a very loose integration is obtained by portfolio-based solving [9, 8] and by pre- and inprocessing [5, 14]. A very tight integration is based on the idea of learning generalized axioms [15]: during search QCDCL solvers learn clauses (conflicts) and cubes (solutions) that are justified by Q-resolution [10]. A generalized axiom is a clause or cube that is learned by calling an oracle solver. Such an oracle can be any complete or incomplete QBF solver.

So far, the usage of generalized axioms has one major drawback: if used in a Q-resolution proof, the QBF variant of resolution proofs as produced by QCDCL solvers, its correctness has no justification, because it was derived by using an oracle solver. Consequently, Q-resolution proofs that rely on generalized axioms cannot be checked by an independent checking tool and the result of the QBF solver cannot be certified. Therefore, generalized axioms have to be disabled if a certificate is required.


© Mark Peyrer and Martina Seidl;
licensed under Creative Commons License CC-BY 4.0



To overcome this limitation and certify the correctness of Q-resolution proofs with generalized axioms, we present the novel certification framework `QRP+Gen` which extends the Q-resolution certification framework `QRPcheck` with generalized axioms. To this end, we furthermore extended the Q-resolution format `QRP` to incorporate their certification. As a proof-of-concept we equipped the QCDCL-based solver `DepQBF` [13] with proof generation for generalized axioms obtained by deciding certain propositional abstractions of the considered QBF. Our implementation is publicly available at

`https://doi.org/10.5281/zenodo.16307432`

2 QCDCL with Generalized Axioms

Traditional search-based QBF solving with conflict-driven clause learning and solution-driven cube learning (QCDCL) [23, 7] successively generates variable assignments for a given QBF $\phi = \Pi : \Psi$ until ϕ is decided. Usually ϕ is in prenex conjunctive normal form (PCNF). Here, $\Pi = Q_1 x_1, \dots Q_n x_n$ is a quantifier prefix with $Q_i \in \{\forall, \exists\}$ and the propositional matrix Ψ is a conjunction of clauses. A clause is a disjunction of literals and a literal is a variable or a negated variable. Note that not every variable that occurs in the matrix has to be part of the prefix (free variables). In contrast to SAT, variables may not be assigned in an arbitrary order, but the dependencies imposed by the quantifier prefix have to be considered. Still, clauses are learned from assignments that falsify the formula to prune the search space. In addition, also cubes (conjunctions of literals) are learned from assignments that satisfy the propositional part of the formula (solutions). While clauses are conjunctively added to the propositional matrix, cubes are disjunctively collected in a cube database. The addition of clauses and cubes preserves the truth/falsity of the original formula, because they are derived via some variant of Q-resolution, the extension of propositional resolution for QBF. On this basis, QCDCL solvers can produce Q-resolution proofs as certificates for the solving results. For false QBFs, clause Q-resolution proofs are generated, and, dually, cube Q-resolution proofs are generated for true formulas. Both types of proofs can efficiently be checked [20]. We want to underline how Q-resolution is in general not expressive enough for QCDCL, which causes the need to disable certain techniques in actual implementations when proof generation is required.

In general, learning smaller clauses and cubes w.r.t the amount of literals is favorable during the search. However, in classical QCDCL implementations, especially large cubes are learned because the cube database is empty in the beginning. Hence, there are no cubes serving as axioms in cube Q-resolution proofs. To initialize the cube database, assignments satisfying the propositional matrix are taken, from which smaller cubes are derived until the empty cube is found [24]. Axioms of clausal Q-resolution proofs are the clauses from the propositional matrix of the original formula.

To learn smaller clauses and cubes faster, the concept of *generalized axioms* has been introduced [15]. Consider a QBF $\phi = \Pi : \Psi$ and assume that during the search a partial assignment $\sigma = \{l_1, \dots, l_n\}$ has been found respecting the dependencies imposed by the quantifier prefix. If a variable $x \in \sigma$, then x is set to true, whereas if $\neg x \in \sigma$, then x is set to false. Furthermore, there is no variable x with $x, \neg x \in \sigma$. Now assume that $\phi' = \phi[\sigma]$ is not decided where $\phi[\sigma]$ is the formula obtained by assigning all variables according to σ. Using an oracle solver that is able to decide ϕ', either a clause or a cube can be learned dependent on the value of ϕ'. If ϕ' is false, then the clause $(\bar{l}_1 \vee \dots \vee \bar{l}_n)$ can be learned. Otherwise, the cube $(l_1 \wedge \dots \wedge l_n)$ can be learned. The oracle can be any solver that is able to decide a QBF. It could be, for example, a solver that is based on a decision procedure of orthogonal

strength like expansion-based solving [11]. Also incomplete methods can be used as oracles. If no solution is found, then the QCDCL search is resumed under the assignment σ. The following example illustrates the power of the approach.

▶ **Example 1.** A formula family of false instances that is known to be hard for Q-resolution are the *parity formulas* [1] $\mathsf{parity}(n) = \exists x_1 \dots x_n \forall z \exists t_2 \dots t_n : \psi(n)$ where $\psi(n)$ is defined by

$$\psi(n) = xor(x_1, x_2, t_2) \wedge \bigwedge_{i=3}^{n} xor(t_{i-1}, x_i, t_i) \wedge (z \vee t_n) \wedge (\neg z \vee \neg t_n)$$

with $xor(a, b, c) = (\neg a \vee \neg b \vee \neg c) \wedge (a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (a \vee \neg b \vee c)$. To obtain formulas that are hard for Q-resolution, but easy with generalized axioms consider the formula family

$$\mathsf{parity}^*(n) = \exists x \exists x_1 \dots x_n \forall z \exists t_2 \dots t_n : (x \to \psi(n)) \wedge (\neg x \to \psi(n)) \wedge (z \leftrightarrow t_n)$$

After assigning x to either value, a solver needs to solve a parity problem of a given size n with the additional equivalence $(z \leftrightarrow t_n)$. Obviously, this contradicts with $(z \not\leftrightarrow t_n)$ that is in $\psi(n)$. However, if the variables are assigned according to the quantifier prefix as it is done by a QCDCL-solver, this obvious contradiction is detected very late. Assuming that a QCDCL-solver uses Q-resolution as a proof format, the algorithm will be restricted by such formulas causing the $\mathsf{parity}^*(n)$ to be hard for rather small values of n.

However, if an oracle call to a SAT solver is made, this will almost immediately conclude unsatisfiability due to being able to neglect the quantifier dependencies. It is easy to prove that the propositional formula is false for both values of x. In consequence, also the QBF is false. By deriving x and $\neg x$ as generalized axioms, short proofs can be found for the QBF.

The only problem with this approach is that the proof containing generalized axioms is not a valid certificate for the solving result, because the axioms are not justified. For this reason, QCDCL solvers do not provide certificates when generalized axioms are enabled so far. Therefore, no independent checking is possible and the result of the solver (and the internally called oracles) has to be trusted without witness.

3 The QRP+Gen Certification Framework

To enable certification for QCDCL solvers that use solver oracles to learn generalized axioms either in terms of clauses or cubes, we developed the `QRP+Gen` framework. It is based on the `QRPcheck` certification tool chain [19] which provides a checker for Q-resolution proofs in the `QRP` format. To close the holes in the proofs introduced by generalized axioms, the oracles have to also provide witnesses which are used to justify the soundness of the learned axioms. The general architecture of our framework is shown in Figure 1 which allows for an automatic integration of the different proofs giving a full witness for the input formula.

The QRP+Gen Proof Format

The `QRP` format was extended to the `QRP+Gen` format. This extension allows to link the proof produced by an oracle with the respective generalized axiom. Proofs in the `QRP+Gen` format may contain classical axioms in the form of clauses that occur in the input formula, or in the form of covering cubes which satisfy the propositional matrix. Furthermore, it may contain clauses/cubes that are derived via resolution by referring to their antecedents as well as clauses/cubes simplified by universal/existential reduction. The soundness of deriving

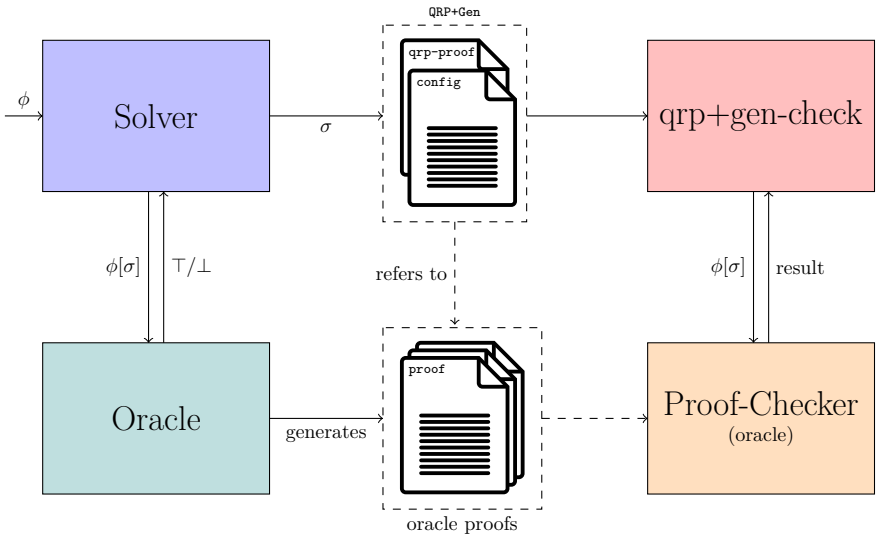


Figure 1 Architecture of `QRP+Gen`.

those clauses/cubes can be efficiently checked based on the clauses of the input formula and clauses/cubes that have previously been checked. In addition, *generalized axioms* are allowed in the `QRP+Gen` format. These are neither derived by Q-resolution, nor occur in the input formula, nor are covering cubes. Here, we need the proof of the oracle. A generalized axiom is indicated by one *negative* antecedent in the format, which absolute value describes the ID of the corresponding oracle proof. Information on the respective proof is given in a configuration file which has to be provided to the checker. The configuration file has the following format:

```
CONFIG = HEADER \n {ENTRY \n}
HEADER = " Path " ; number
ENTRY = number ; transformer-specifier ; format-specifier ; " Path "
```

The *HEADER* consists of the path to the original qdimacs-formula as well as the amount of proofs that are referenced in the following (the number of entries). Each *ENTRY* is identified by a unique proof ID. Next, a transformer is specified, which applies simple syntactical modifications on the input QBF that can efficiently be checked. For example, it could be necessary to drop the prefix of the input QBF. Next, the proof format of the proof produced by the oracle is indicated. Finally, the path to the proof file is provided. An example of a config-file is shown in the middle of Figure 2.

Generation of Proofs

A QBF ϕ in PCNF is passed to a QCDCL solver that calls oracle solvers to learn generalized axioms. It is assumed that these oracle solvers are able to produce certificates that can be efficiently checked. During the classical QCDCL search, a Q-resolution trace with clauses and cubes is produced as usual. If a generalized axiom is learned based on a partial assignment σ, then the oracle solver has to provide a certificate for the truth value of $\phi[\sigma]$. An entry in the configuration file is generated and the generalized axiom is added to the Q-resolution trace with the respective pointer to the configuration file. By storing the generalized axiom,

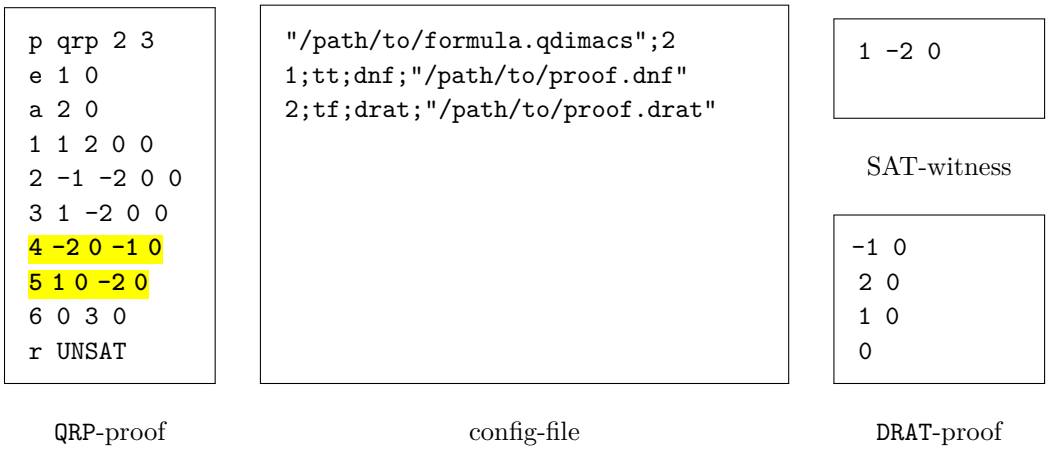


Figure 2 Example of a QRP+Gen proof.

ϕ under assignment σ can be reconstructed. The Q-resolution proof with generalized axioms refers to the oracle proofs which are stored in separate files. The solving oracle does not need any information about the QCDCL solver and its proof generation does not have to be modified. The left part of Figure 2 shows a Q-resolution proof with two generalized axioms (the highlighted clause/cube). On the right, there are two oracle proofs: one is a DRAT proof and the other one is a satisfying assignment.

Proof Checking

We extended the tool QRPcheck [19] to be able to check QRP+Gen-proofs. For checking, a config-file has to be specified as a parameter and is loaded before starting with verifying the QRP+Gen-proof. An example of such a file can be seen in the middle of figure 2. This way, all information necessary to find and check the oracle proofs is provided. In case a generalized axiom is found in the proof, an external checker as specified in the config file is called. Before calling the external checker, some transformation steps might be necessary to ensure that $\phi[\sigma]$ has the correct format. Furthermore, the correctness of all clauses/cubes derived by standard Q-resolution is checked.

4 Certifying Generalized Axioms in DepQBF

In the following, we present how we extended the QBF solver DepQBF to obtain certificates for generalized axioms that use two SAT solvers as oracles. The SAT solvers are called with certain propositional abstractions of the input QBF under the current assignment σ. Clauses and cubes learned as generalized axioms are justified by the techniques *Trivial Truth* and *Trivial Falsity* [6, 15]. To certify the correctness of such generalized axioms, different types of proofs are combined as shown in Figure 3. In addition, DepQBF learns generalized axioms by calling the expansion-based solver Nenofex [12] which is not able to produce proofs. Extending Nenofex with proof generation capabilities is beyond the scope of this work, hence we disabled learning generalized axioms with Nenofex. In the original implementation, DepQBF employed PicoSAT [3] as SAT solver. Instead of PicoSAT, we use the latest version of the SAT solver CaDiCaL [4] which is able to produce DRAT proofs [22].[1]

[1] We thank Maximilian Heisinger for integrating CaDiCaL into DepQBF.

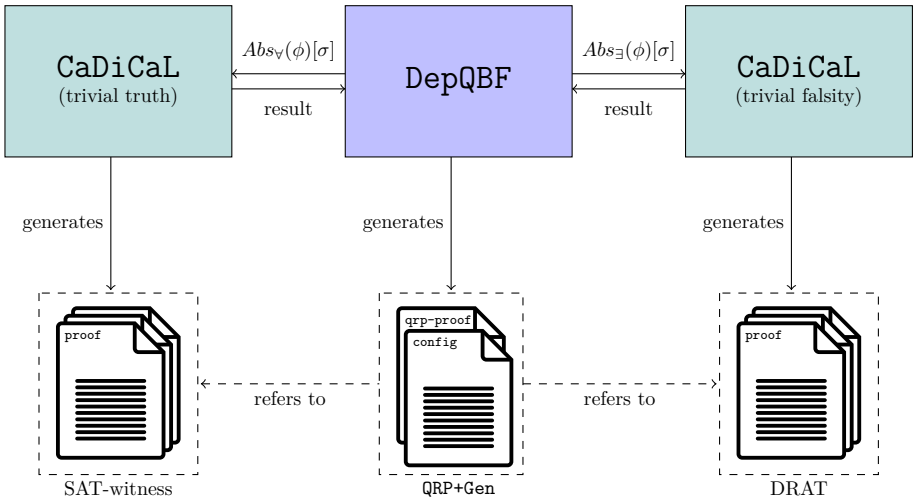


Figure 3 Combination of proofs in `QRP+Gen`.

Trivial Falsity

Trivial falsity is based on the following observation: if the matrix Ψ of a given QBF $\phi = \Pi : \Psi$ is already propositionally unsatisfiable, then the QBF is false. Given a partial assignment σ that is consistent with the Q-resolution rules, then the clause $\neg\sigma$ can be learned as generalized axiom if $\Psi[\sigma]$ is unsatisfiable. The `DRAT` proof of the unsatisfiability of $\Psi[\sigma]$ produced by `CaDiCaL` serves as witness for the generalized axiom and is referenced in the `QRP+Gen` proof. In the proof checking phase, our checker `QRP+GenCheck` can easily check the correctness of the generalized axiom $\neg\sigma$ by stripping the original formula of its prefix (done by a transformer in `QRP+GenCheck`) and pass it to `DRAT-trim` [22] together with the corresponding `DRAT` proof.

Trivial Truth

Trivial truth is based on the following observation: a QBF $\Pi : \Psi$ is true if the propositional formula Ψ' is satisfiable where Ψ' is obtained from Ψ by removing all universal literals from each clause. Given a partial assignment σ that is consistent with the quantifier prefix Π, then the cube σ can be learned as generalized axiom if $(\Psi[\sigma])'$ (i.e. the first σ is applied on Ψ and then the remaining universal literals are dropped) is satisfiable. As a witness of σ, a satisfying assignment τ of $(\Psi[\sigma])'$ is stored. For checking the correctness of generalized axioms justified by trivial truth, we implemented a checker called `ttCheck` that takes Ψ as well as σ and τ as input. It checks the consistency of τ and σ, i.e., it ensures that there is no literal l with $l \in \tau$ and $\bar{l} \in \sigma$. Furthermore, it checks that for each clause $C \in \Psi$, there exists literal $l \in C$ such that $l \in \tau$ and l is existential or that $l \in \sigma$. This allows `ttCheck` to certify the correct application of trivial truth in a very efficient manner.

5 Experimental Evaluation

We conducted two experiments showing the impact of `QRP+Gen` applied on the previously described case study. Both experiments were executed on a Ubuntu 24.04 machine with dual-socket AMD EPYC 7313@3.7 GHz CPUs. We assigned 8 GB RAM as well as 1 h per task as resource limitations.

Crafted Formulas

Going back to example 1, the introduced crafted formulas $\mathsf{parity}^*(n)$ are hard for Q-resolution but easy when generalized axioms are used. We implemented a generator for the parity^* formulas on the base of `qbffam` [2] and tested the results. `depqbf-qrp` does not use generalized axioms, whereas `depqbf-qrpgen` uses trivial truth and trivial falsity. Both are producing a proof trace. This first experiment used the range $n = 2 \dots 25$. The needed resources per formula size can be seen in figure 4. Note that the space-data represent the size of the proof, not the RAM used.

While `DepQBF` without generalized axiom shows an exponential increase in both solving time and proof size, the version using generalized axioms only has a linear increase in space. The reason for that is `CaDiCaL`, which is able to solve the trivial falsity represented by the formula directly whereas the Q-resolution of `DepQBF` has to face the difficulty of the parity family [1]. This underlines the potential of using generalized axioms, allowing the oracle to exploit structures unknown to the QBF-solver.

QBF-Gallery 23

In the second experiment, we used the QBF-Gallery 2023 [21] as a test set to evaluate the runtime of different configurations. As a secondary set, we used the preprocessor `Bloqqer` [5] to generate simplified versions of the formulas from the Gallery. Per test set, four different configurations were investigated. Techniques like advanced dependency management and lazy QPUP [16] were deactivated in general as they do not support proof generation. The rest of the configuration was dependent on the run:

- `depqbf-no-gen` is without tracing and no generalized axioms.
- `depqbf` is the base configuration without tracing but with generalized axioms.
- `depqbf-qrp` is with normal `QRP`-tracing (thus without generalized axioms)
- `depqbf-qrpgen` is with full `QRP+Gen`-tracing

The results can be seen in figure 5. Additionally, we tested how many samples can be verified using `QRP+GenCheck` by granting an additional 1 h of resources. For the QBF-gallery 2023 (left part of figure 5), out of 50 solved examples, 48 instances were verified w.r.t. the resource limitations. For the preprocessed samples (right part of figure 5), all 126 could be verified. In terms of proof sizes, `QRP` and `QRP+Gen` are similar for all instances of both test sets.

The results of Lonsing et al. [15] are reflected in both benchmark-sets. While `DepQBF` with Dynamic Quantified Blocked Clause Elimination (dyn-QBCE) and generalized axioms is able to outperform its previous version in general, using a version applying only generalized

n	time [s]		space [byte]	
	`depqbf-qrpgen`	`depqbf-qrp`	`depqbf-qrpgen`	`depqbf-qrp`
2	0.02	0.02	308	924
3	0.02	0.02	448	2.3k
...	...	...	...	...
22	0.02	208	3.8k	8.2G
23	0.02	544	4.0k	16G
24	0.02	1442	4.2k	38G
25	0.02	N/A	4.4k	N/A

Figure 4 The first experiment conducted with the crafted formulas.

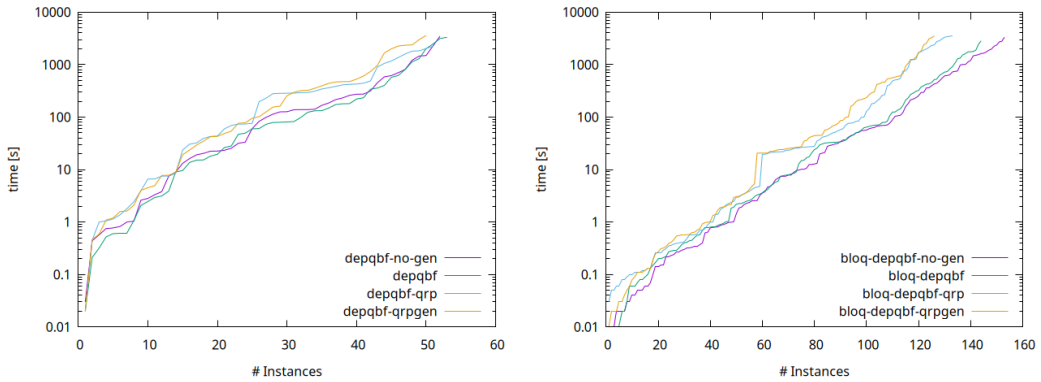


Figure 5 Experiments conducted on QBF gallery 23 and preprocessed QBF gallery 23.

axioms is not. Q-resolution is not powerful enough to certify the deletion of clauses which are blocked under a specific assignment. Providing a solution (e.g. by extending the proof system) would be beyond the scope of this show case. However, we have shown that even without dyn-QBCE generalized axioms can be beneficial. Moreover, as there is only little drawback enabling `QRP+Gen`-certifications, we lay the foundation towards a more general proof system enabling the full potential of the techniques applied by `DepQBF` with proper certification.

6 Conclusion

We presented `QRP+Gen`, the – to the best of our knowledge – first framework allowing certificates for reasoning with generalized axioms. We extended the well-established `QRP` proof format with justifications for clauses and cubes derived by generalized axioms based on results obtained from oracle solvers. In our framework, any oracle solver able to produce proofs for which checking tools are available can be considered.

As a case study, we implemented proof generation and checking for generalized axioms obtained by the abstraction-based techniques trivial truth and trivial falsity in the solver `DepQBF`. We evaluated the framework on different types of benchmarks and observed only little overhead caused by proof generation. With our framework, it is now possible to independently confirm the result of a QBF solver that relies on general axioms.

The presented approach is, however, not limited to generalized axioms. Our framework can be employed in other scenarios where different solving systems based on different proof systems are combined. Potential applications include interfacing proofs produced by preprocessing tools and complete solvers. Our framework could also be applied in the context of cube&conquer-based solving [8].

References

1 Olaf Beyersdorff, Leroy Chew, and Mikoláš Janota. New resolution-based QBF calculi and their proof complexity. *ACM Transactions on Computation Theory*, 11(4):1–42, 2019. `doi:10.1145/3352155`.

2 Olaf Beyersdorff, Luca Pulina, Martina Seidl, and Ankit Shukla. Qbffam: A tool for generating QBF families from proof complexity. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 21–29. Springer, 2021. `doi:10.1007/978-3-030-80223-3_3`.

3 Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, 2008. doi:10.3233/SAT190039.

4 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In *36th International Conference on Computer Aided Verification*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

5 Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In *23rd International Conference on Automated Deduction*, pages 101–115. Springer, 2011. doi:10.1007/978-3-642-22438-6_10.

6 Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An algorithm to evaluate quantified boolean formulae. *AAAI/IAAI*, 98:262–267, 1998. URL: http://www.aaai.org/Library/AAAI/1998/aaai98-036.php.

7 Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Clause/term resolution and learning in the evaluation of quantified boolean formulas. *Journal of Artificial Intelligence Research*, 26:371–416, 2006. doi:10.1613/JAIR.1959.

8 Maximilian Heisinger, Martina Seidl, and Armin Biere. Paraqooba: A fast and flexible framework for parallel and distributed QBF solving. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 426–447. Springer, 2023. doi:10.1007/978-3-031-30823-9_22.

9 Holger H Hoos, Tomáš Peitl, Friedrich Slivovsky, and Stefan Szeider. Portfolio-based algorithm selection for circuit qbfs. In *24th International Conference of Principles and Practice of Constraint Programming*, pages 195–209. Springer, 2018. doi:10.1007/978-3-319-98334-9_13.

10 Mikoláš Janota. On Q-Resolution and CDCL QBF solving. In *19th International Conference on Theory and Applications of Satisfiability Testing*, pages 402–418. Springer, 2016.

11 Mikoláš Janota and Joao Marques-Silva. Expansion-based QBF solving versus Q-Resolution. *Theoretical Computer Science*, 577:25–42, 2015. doi:10.1016/J.TCS.2015.01.048.

12 Florian Lonsing and Armin Biere. Nenofex: Expanding nnf for qbf solving. In *11th International Conference on Theory and Applications of Satisfiability Testing*, pages 196–210. Springer, 2008. doi:10.1007/978-3-540-79719-7_19.

13 Florian Lonsing and Uwe Egly. DepQBF 6.0: A search-based QBF solver beyond traditional QCDCL. In *26th International Conference on Automated Deduction*, pages 371–384. Springer, 2017. doi:10.1007/978-3-319-63046-5_23.

14 Florian Lonsing and Uwe Egly. QRATPre+: effective QBF preprocessing via strong redundancy properties. In *22nd International Conference on Theory and Applications of Satisfiability Testing*, pages 203–210. Springer, 2019. doi:10.1007/978-3-030-24258-9_14.

15 Florian Lonsing, Uwe Egly, and Martina Seidl. Q-Resolution with Generalized Axioms. In *19th International Conference on Theory and Applications of Satisfiability Testing*, pages 435–452. Springer, 2016. doi:10.1007/978-3-319-40970-2_27.

16 Florian Lonsing, Uwe Egly, and Allen Van Gelder. Efficient clause learning for quantified Boolean formulas via QBF pseudo unit propagation. In *16th International Conference on Theory and Applications of Satisfiability Testing*, pages 100–115. Springer, 2013. doi:10.1007/978-3-642-39071-5_9.

17 Paolo Marin, Massimo Narizzano, Luca Pulina, Armando Tacchella, and Enrico Giunchiglia. Twelve years of QBF evaluations: QSAT is PSPACE-hard and it shows. *Fundamenta Informaticae*, 149(1-2):133–158, 2016. doi:10.3233/FI-2016-1445.

18 Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of satisfiability*, pages 133–182. ios Press, 2021. doi:10.3233/FAIA200987.

19 Aina Niemetz. QRPcheck. URL: https://fmv.jku.at/qrpcheck/.

20 Aina Niemetz, Mathias Preiner, Florian Lonsing, Martina Seidl, and Armin Biere. Resolution-based Certificate Extraction for QBF: (tool presentation). In *15th International Conference on Theory and Applications of Satisfiability Testing*, pages 430–435. Springer, 2012. doi:10.1007/978-3-642-31612-8_33.

21 Luca Pulina, Martina Seidl, and Simone Heisinger. QBFGallery 2023. URL: https://qbf23.pages.sai.jku.at/gallery/.

22 Nathan Wetzler, Marijn JH Heule, and Warren A Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *17th International Conference on Theory and Applications of Satisfiability Testing*, pages 422–429. Springer, 2014.

23 Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified Boolean satisfiability solver. In *IEEE/ACM international conference on Computer-aided design*, pages 442–449, 2002. doi:10.1145/774572.774637.

24 Lintao Zhang and Sharad Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In *International Conference on Principles and Practice of Constraint Programming*, pages 200–215. Springer, 2002. doi:10.1007/3-540-46135-3_14.

Analyzing Reformulation Performance in Core-Guided MaxSAT Solving

André Schidler
University of Freiburg, Germany
TU Wien, Austria

Stefan Szeider
TU Wien, Austria

Abstract

Core-guided algorithms like OLL are among the best methods for solving the Maximum Satisfiability problem (MaxSAT). Although some performance characteristics of OLL have been studied, a comprehensive experimental analysis of its reformulation behavior is still missing. In this paper, we present a large-scale study on how different reformulations of a MaxSAT instance produced by OLL affect solver performance. By representing these reformulations as a directed acyclic graph (DAG), we isolate the impact of structural features – such as the size and interconnectivity of unsatisfiable cores – on solver runtime. Our extensive experimental evaluation of over 600k solver runs reveals clear correlations between DAG properties and performance outcomes. These results suggest a new avenue for designing heuristics that steer the solver toward more tractable reformulations. All OLL DAGs and performance data from our experiments are publicly available to foster further research.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases maximum satisfiability, OLL, core-guided

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.26

Supplementary Material *Dataset*: `https://doi.org/10.5281/zenodo.15584371`

Funding This work was supported by Austrian Science Fund (FWF) grants 10.55776/P36420 and 10.55776/COE12, as well as an Amazon Research Award (Fall/2023).

1 Introduction

MaxSAT (Maximum Propositional Satisfiability) extends SAT by optimizing solutions under constraints. Applications span software analysis [26], post-silicon fault detection [28], concurrent program debugging [16], smartphone app security [13], and computer-aided design [23, 24]. As the complexity of these domains grows, the need for efficient and scalable MaxSAT solvers becomes increasingly important.

Core-guided algorithms are among the most effective approaches for solving MaxSAT instances, which iteratively refine the problem using information extracted from unsatisfiable cores. One prominent example of such an algorithm is *OLL* [1, 22] which has consistently performed well in MaxSAT competitions [9][1]. OLL's ability to compute optimal solutions with proof logging [7] makes it both efficient and verifiable.

Despite its strengths, OLL's performance can vary significantly depending on internal design choices, particularly the reformulations it generates during solving. Reformulations play a pivotal role in the algorithm's efficiency by converting the MaxSAT instance into a series of SAT instances. However, not all reformulations are equal: some result in SAT instances that are computationally hard to solve, leading to longer runtimes or solver failures.

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


© André Schidler and Stefan Szeider;
licensed under Creative Commons License CC-BY 4.0



Several known factors affect the runtime of OLL: (i) instance-specific reasons, such as the SAT instance being inherently hard to solve; (ii) heuristic-specific reasons, where the heuristics used during OLL take too much time; (iii) reformulation-specific reasons, where the SAT instance becomes too hard due to the reformulation; and (iv) SAT solver variability, which may also impact performance, as different solvers behave differently.

This paper focuses on reason (iii), examining how reformulations alone can lead to significant variations in runtime. By isolating this factor, we aim to understand why some reformulations perform better than others, enabling targeted improvements in OLL's implementation.

To achieve this, we propose a framework for analyzing reformulations using the *OLL DAG*, a directed acyclic graph representation of the reformulations. The OLL DAG abstracts away heuristic-specific and instance-specific issues, allowing us to focus exclusively on the structural properties of the reformulations and their impact on solver runtime. Our results on the correlation between OLL DAG properties and solver performance pave the way for analyzing other OLL-specific features, such as heuristics, by observing their influence on the OLL DAG structure.

Our main contributions are as follows:

1. A framework for analyzing the impact of different OLL reformulations on solver performance using a graph-based representation.
2. A systematic, large-scale experimental analysis based on the structure of OLL DAGs, incorporating over 600 000 abstract OLL runs with runtimes of up to five hours.
3. A publicly available collection of over 100 000 OLL DAGs and their associated performance data intended to foster further research into MaxSAT solving.
4. Insights into how heuristics can be evaluated and improved by analyzing their impact on OLL DAG structure.

By isolating the complexity introduced by reformulations, we provide actionable insights into improving solver performance and developing new heuristics that steer OLL toward simpler, more efficient reformulations.

1.1 Related Work

Solvers, and by extension MaxSAT algorithms, are evaluated each year at the annual MaxSAT evaluation [10]. In the course of solver development, much empirical work is done by the developers improving their solvers. Unfortunately, this work is usually focused on a single solver and not published. Little empirical published research exists, apart from an analysis of using different SAT solvers in the core-guided solver RC2 [17].

The impact of reformulations has been evaluated on a theoretical level. Bacchus and Narodytska [5] provided an early formalization of how cores discovered by core-guided algorithms relate to the original formula, showing that each core found corresponds to a union of cores from the input formula. They identified residue subsumption as a potential optimization where some solutions of the cardinality constraints could be ignored while maintaining correctness. Building on this work, Narodytska and Bjørner [25] conducted a comprehensive analysis of both cores and correction sets, establishing key properties like MinHS-monotonicity that connect core-guided and MinHS-style approaches. They proved that cores extracted by core-guided solvers possess the same important characteristics as those found by implicit hitting set methods, demonstrating an intrinsic connection between these different solving paradigms. Together, these papers established much of the theoretical framework for understanding modern core-guided MaxSAT solving available so far. However, these papers do not contain any empirical analysis and are, therefore, orthogonal to the work in this paper.

2 Preliminaries

Propositional Satisfiability

A literal is a Boolean variable v or its negation $\neg v$, a clause C is a disjunction of literals $C = \ell_1 \vee \cdots \vee \ell_k$, and a formula F in conjunctive normal form (CNF) is a conjunction of clauses $F = C_1 \wedge \cdots \wedge C_m$. We view clauses as a set of literals and a formula as a set of clauses. A variable assignment σ for the variables in F maps each variable to either 1 or 0 corresponding to *true* and *false*. We extend assignments beyond variables as follows. The value assigned to a negated variable is $\sigma(\neg v) = 1 - \sigma(v)$, the value of a clause C is $\sigma(C) = \max_{\ell \in C} \sigma(\ell)$, and the value of a formula is $\sigma(F) = \min_{C \in F} \sigma(C)$. A *model* is an assignment σ such that $\sigma(F) = 1$ and a formula is *satisfiable* if it has a model, and, otherwise, *unsatisfiable*.

MaxSAT

A Maximum Satisfiability (MaxSAT) instance (F, O) consists of a CNF formula F, that we assume is satisfiable, and an objective function of its variables $O = \mathit{Var} \to \mathbb{N}_0$, where Var refers to F's variables. Whenever $O(v) > 0$ for a variable v, we say v is an objective variable. We use b to denote objective variables and v for general variables. An assignment has cost $\Sigma_{v \in \mathit{Var}} O(v) \cdot \sigma(v)$. Hence, any objective variable with $\sigma(b) = 1$ *incurs cost*. An optimal solution to a MaxSAT instance is a model of F with minimum cost. The above definition defines *Weighted Partial MaxSAT* and is equivalent to other definitions, such as using soft and hard clauses. Whenever $O(v) \in \{0, 1\}$ for all $v \in \mathit{Var}$, we say that the instance is *unweighted*.

Cores

Given a MaxSAT instance (F, O), an *unsatisfiable core* C or just *core* is a clause such that C only contains unnegated objective variables and C is entailed by F. Hence, a core states that at least one of its objective variables has to incur cost.

Assumptions

Generally, a SAT solver checks if a given formula is satisfiable or not. Modern SAT solvers also allow for solving under *assumptions* [12, 27]: given a set of literals $\mathcal{A}$ – the assumptions – the SAT solver checks if F has a model σ such that for all $\ell \in \mathcal{A}$ holds that $\sigma(\ell) = 1$, i.e., $\mathcal{A}$ can be extended to a model. Whenever no such model exists, the SAT solver returns a core that is a subset of $\mathcal{A}$.

Limited SAT solving

Limited SAT solving refers to a heuristic use of SAT solvers, where the SAT solver stops after a set limit. This limit can be on the number of conflicts, runtime, or enforced from outside the solver and whenever the limit is reached, the solver returns *unknown*. Limited SAT solving allows controlling how many computing resources are invested into a single SAT call at the price but limited SAT calls might not produce usable results. In this paper we only consider limits on the number of conflicts.

3 The OLL Algorithm

We give a quick overview over the OLL algorithm [1, 22]. The basic OLL algorithm is shown in Algorithm 1. In each iteration, one core is extracted (Line 4) and the instance is reformulated based on this core (Line 7). Each core increases the lower bound by the minimum cost over the core's objective variables (Line 6). The optimal cost has been found as soon as no more cores can be found for the reformulated instance (Line 5).

Algorithm 1 The OLL algorithm for an instance (F, O).

```
1: lb ← 0
2: (F', O') ← (F, O)
3: while True do
4:     C ← extract_core(F', O')
5:     if C = None then return lb end if
6:     lb ← lb + min{ O'(b) : b ∈ C }
7:     (F', O') ← reformulate(F', O', C)
8: end while
```

Next, we describe the two core details of the algorithm: *core extraction* and *reformulation*.

3.1 Reformulation

Reformulation is a central concept of the OLL algorithm. Given an extracted core C, we determine its cost $c = \min\{ O'(b) : b \in C \}$. Since at least one of the objective variables in C has to incur cost, c gives us the minimum cost increase implied by C.

We now compute the reformulated formula F'. Let $n = |C|$. We reformulate F' by adding for each $1 < i \leq n$ a variable b_i^C and clauses that express $\Sigma_{b \in C} b \geq i \rightarrow b_i^C$. Hence, the variable b_i^C is true whenever at least i many objective variables in the core incur cost. The clauses are created by using a cardinality constraint over the core. In this paper, we only consider the *Totalizer* constraints, a common choice in MaxSAT solvers [6, 21].

We adapt O' by setting for each $b \in C$ its cost $O'(b) \coloneqq O'(b) - c$. As this will reduce at least one variable's cost to zero, some variables become non-objectives after reformulation. Further, for each new variable b_i^C we set $O'(b_i^C) \coloneqq c$ expressing that a cardinality increase of C incurs at least cost c. In our implementation, the variables b_i^C with $i > 2$ and the respective constraints are added lazily: they are introduced only when b_{i-1}^C becomes a non-objective, as they are not required before.

3.2 Core Extraction

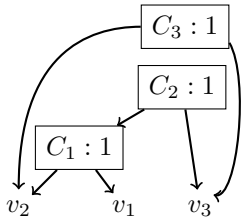
In this paper, we assume that cores are extracted using a SAT solver, which is common in state-of-the-art OLL solvers [9]. Assumptions are used during core extraction: the OLL algorithm assumes that no objective variable in O' incurs cost. The SAT solver then either returns satisfiable or unsatisfiable. In case the result is satisfiable, the model found by the SAT solver is an optimal solution to the reformulated MaxSAT instance that translates to an optimal solution to the original instance and lb – the sum over all core costs – is equal to the optimal cost. Given infinite time, the SAT solver will always return satisfiable whenever lb reaches the optimal value.

In case of an unsatisfiable result, we obtain a core C using the SAT solver and perform the reformulation. Ideally, we want this core to be subset minimal (an *MUS*), but SAT solvers rarely return subset minimal cores. This necessitates an important heuristic in core-guided MaxSAT solving: *core minimization*. We will discuss core minimization and other heuristics commonly used in OLL MaxSAT solving later. First, we introduce the OLL DAG.

4 The OLL DAG

We propose a representation of the OLL reformulations based on a directed acyclic graph (DAG). For a given instance (F, O) and reformulation (F', O'), we define the corresponding OLL DAG $D = (V, E)$ as follows. Let $\mathcal{C}$ be the set of all extracted cores used in (F', O') and $\mathcal{B} = \{ b \in Var : O(b) > 0 \}$. Then, $V = \mathcal{C} \cup \mathcal{B}$ and E contains two types of arcs. The first type consists of arcs $(C, b) \in \mathcal{C} \times \mathcal{B}$, where $(C, b) \in E$ if and only if $b \in C$. Further, the second type of arc is of type $(C, C') \in \mathcal{C} \times \mathcal{C}$ where $(C, C') \in E$ if and only if some $b_j^{C'} \in C$ for $1 < j \leq |C'|$. Hence, the OLL DAG connects cores with the objective variables it contains, or the respective core in case the objective variable was introduced by a reformulation. We annotate D with some extra information: the cost of the core and the number of the core (since cores are extracted sequentially, we give them a consecutive number). Nodes in $\mathcal{B}$ that have in-degree 0 are ignored in our analysis.

▶ **Example 1.** We give an example OLL run and the corresponding OLL DAG. Let $F \coloneqq \{v_1, v_2\}, \{v_2, v_3\}, \{v_1, v_3\}$ and $O \coloneqq v_1 + 2 \cdot v_2 + 3 \cdot v_3$. Any model has to set at least two variables to true and the minimum cost would be 3. In the first iteration, we find core $C_1 = \{v_1, v_2\}$, which causes the reformulation $F' \coloneqq \{v_1, v_2\}, \{v_2, v_3\}, \{v_1, v_3\}, \{\neg v_1, \neg v_2, b_2^{C_1}\}$, which adds the cardinality semantics for $b_2^{C_1}$. Further, $lb \coloneqq 1$ and $O' \coloneqq v_2 + 3 \cdot v_3 + b_2^{C_1}$: the adaption of the coefficients reduces the coefficient of v_2 and makes v_1 a non-objective. Let the OLL solver find as the second core $C_2 = \{v_3, b_2^{C_1}\}$, for brevity we skip the reformulation as it is analogous to the previous one. This sets $lb \coloneqq 2$. Finally, the OLL solver finds core $C_3 = \{v_2, v_3\}$, which sets $lb \coloneqq 3$. In the next iteration, the OLL solver will assume $\{\neg v_3, \neg b_2^{C_2}, \neg b_2^{C_3}\}$, which can be extended to the model $\{v_1, v_2, \neg v_3, b_2^{C_1}, \neg b_2^{C_2}, \neg b_2^{C_3}\}$. The lb matches the optimal cost of 3 and the corresponding OLL DAG is:



The graph representation allows us to use common graph metrics and analyze if they correlate with successful or unsuccessful solver runs. We analyze several structural properties of D and in this paper, we focus on the following ones, where we consider, where possible, the maximum, minimum, average, and standard deviation over all nodes in the graph. We refer to these as *metrics*.

- The number of cores $|\mathcal{C}|$.
- Let $D[\mathcal{C}]$ be the subgraph induced by $\mathcal{C}$. We define the *width* of D as the size of the largest antichain of the partial order defined by $D[\mathcal{C}]$.
- The in-degrees of the nodes in D.

- The number of and length of source-sink paths. The *depth* is the maximum length over all source-sink paths.
- The number of weakly connected components.
- The out-degrees of the nodes in D, which corresponds to the core sizes. We disregard nodes with out-degree 0 from aggregates.
- The cost of the cores and the objective variables in the cores.
- The *number of combinations*: for a core C, let i be the smallest i such that $O'(b_i^C) > 0$, then the number of combinations for C is $\binom{|C|}{i-1}$. This is the number of assignments to variables in C that satisfy $\Sigma_{b \in C} < i$.
- We define the *child-degree* of a node as the sum over the in-degrees of its children.
- How many nodes can reach a specific node. Let $r(v)$ be the number of nodes that reach $v \in V$, then we define the *interconnectedness* of D as $\frac{\Sigma_{v \in V} r(v)}{|V| \cdot (|V|-1)}$.

Limitations

The OLL DAG is an abstraction that does not capture two important sources that can cause long runtimes in MaxSAT solving: (i) the runtime of the heuristics is not captured and particularly core minimization can take the majority of the runtime, and (ii) the OLL DAG does not contain instance specific information, although different instances will result in different OLL DAGs. Hence, the OLL DAG is only a good model for analyzing complexity that arises from the reformulations. In our view, it is important to analyze this aspect, and we think the OLL DAG is a good option as it abstracts other sources of complexity away.

4.1 OLL Heuristics

In this section, we discuss several common heuristics used to speed up OLL MaxSAT solving and how they are expected to impact the OLL DAG.

Core Minimization [19]

Core minimization aims to convert a core into an MUS using SAT solving under assumptions. Let C be the core to be minimized, then, core minimization tests for each $b \in C$ whether $C' = C \setminus \{b\}$ is a core. In case the SAT solver returns unsatisfiable, C' is indeed a core and core minimization sets $C \coloneqq C'$, otherwise, the process is continued with an unchanged C. Usually the SAT calls are limited and core minimization may not result in an MUS. We expect that core minimization impacts most metrics, as the core size impacts the nodes' degrees and which nodes are connected, the number of combinations, and the cost of the cores.

Shuffling

Shuffling is a simple method that can also reduce the size of cores before core minimization. The order in which the assumptions are given to the SAT solver can change the core the SAT solver returns. Hence, after finding a core C, the objective variables in C are shuffled repeatedly and for each reordering the SAT solver extracts a new core. This can lead to the SAT solver extracting a new core $C' \subset C$. The expected impact on the OLL DAG is the same as for core minimization.

Disjoint Cores / Weight Aware Core Extraction [8]

Instead of reformulating the instance immediately after extracting a core, we only adjust the costs of the objective variables occurring in the core, without changing F' or introducing new variables. We repeat core extraction with limited SAT calls until no more cores can be extracted. Afterwards, we finish reformulating the instance for all remaining cores at once. Disjoint Cores is expected to increase the width of D, as extracting k cores before reformulation guarantees a width of at least k. Furthermore, this should decrease the depth of D.

Core Exhaustion [3]

A core states that at least one objective variable in the core has to incur cost. Core Exhaustion tries to establish if more than one objective variable has to incur cost. After reformulating the instance based on core C, starting with $i = 2$ we perform a limited SAT call assuming that b_i^C does not incur cost. Whenever the SAT solver returns unsatisfiable, $\{b_i^C\}$ is a core, we increment i and repeat the process until the SAT solver does not return unsatisfiable or $i > |C|$. Core exhaustions are represented as singleton cores in D. We do not count these singleton cores in the number of cores metric.

At-Most-One [15]

At-Most-One assumes for each $b \in \mathcal{B}$ that b does not incur cost and performs unit propagation using only this assumption. Whenever the SAT solver returns unsatisfiable, $\{b\}$ is a core. Otherwise, each objective b' that has value 1 after unit propagation corresponds to a core $\{b, b'\}$. Further, the cores $\{b, b'\}$ can be viewed as edges of an auxiliary graph G, and every clique C of G is also a core that can be exhausted up to $|C| - 1$. At-Most-One repeatedly extracts cliques from G, removing used objective variables from G, and adding C as a core. We expect At-Most-One to create high in-degree nodes.

Stratification [2, 20]

Let $\mathcal{B}'$ be the objective variables defined by O', and *score* $m : \mathcal{B}' \to \mathbb{R}$. Stratification partitions $\mathcal{B}'$ into k classes $(P_1, \dots, P_k)$ based on their score. Core extraction is then split into k phases, where in phase $1 \leq j \leq k$ only the objective variables in $\bigcup_{1 \leq i \leq j} P_i$ are assumed not to incur cost.

A common choice for m is the objective variables' costs O'. The expectation is that this leads to cores introducing new objective variables with high (or low) m, where different choices for m may impact the OLL DAG differently.

Hardening [2]

Whenever we have an upper bound ub on the cost, which is often found during Stratification, we can immediately set certain objective variables to 0. Let $gap = ub - lb$, where lb is the lower bound established by the cores as in Algorithm 1. We can now mark each $(b, c) \in O'$ where $c > gap$ as a non-objective, i.e., $O'(b) \coloneqq 0$, as b incurring cost would exceed the upper bound. Hardening does not impact the OLL DAG, as it does not change the cores being found.

5 Experimental Setup

We use an experimental setup that tries to analyze the impact of the OLL DAG structure independently of the solver that created it, as we are not interested in improving a specific solver but the correlation between OLL DAG structure and successful and fast OLL runs.

We base our experiments on the MaxSAT Evaluation 2023 instance set[2] providing 572 unweighted and 558 weighted instances, as at the time we started our experiments, this was the most recent one available. Although a larger set of instances would be preferable, the comprehensiveness of our experiments requires that we focus on a smaller instance set. Eventually, we gathered over 100 000 OLL DAGs and over 600 000 runtime samples over these OLL DAGs.

We split our experiments into three phases:

1. *Pre-Evaluation* removes instances that are too hard to solve, which reduces the runtime of our experiments.
2. *Sampling* runs the OLL solver using many different configurations and stores the OLL DAGs of these runs.
3. *Timing* measures the runtime of an abstract OLL run which is MaxSAT solver and heuristic independent.

We use our own OLL implementation with all the features described in Section 3. Most importantly, using our implementation, we can vary how all heuristics are applied using parameters. Each run in each phase is limited to five hours. That ensures that each run terminates, but the high timeout also enables slower configurations to complete their run.

5.1 Configurations

We cannot force a solver to produce a specific structure. Hence, we use different configurations aimed towards producing a diverse set of OLL DAGs for our analysis. As the goal is diversity, not all configurations are good configurations in the sense that they will solve instances fast. We vary the limits for the heuristics discussed in Section 4.1 and use several different functions for stratification, including a function that returns a random value.

Concretely, we vary the following parameters.

- The conflict limit for the disjoint core phase, where possible values are for unweighted $\{0, 25\,000, 83\,000, 10\,000\,000\}$ and for weighted $\{0, 2\,500\}$.
- The number of shuffles during core shuffling, possible values are for unweighted $\{0, 5, 100, 250, 500\}$ and for weighted $\{0, 6, 100\}$.
- The conflict limit for core minimization, possible values are for unweighted $\{0, 1, 200, 250, 10\,000\}$ and for weighted $\{0, 1, 150, 2\,000, 10\,000\}$.
- Different scoring functions $\mathcal{B}' \to \mathbb{R}$ where $\mathcal{B}'$ is the set of objective variables in O'. The scoring function is used to determine scores for the order during core minimization, the order in which the assumptions are given to the SAT solver and is used during stratification as described in Section 4.1. Besides the objective variable's coefficient in O' we consider the following scoring functions. We always prioritize $b_i \in O$ over objective variables introduced by reformulation. The remaining objective variables $b_j^C \in O'$ are then ordered based on:
  - cost of C, here, $b_i \in O$ are also ordered by cost;
  - number of possible combinations as defined in Section 4;

[2] https://maxsat-evaluations.github.io/2023/

  - depth of the subgraph rooted at C;
  - length of the longest path from any source to C;
  - number of descendants of C;
  - number of ancestors of C;
  - in-degree of C;
  - size of C;
  - length of the longest source-sink path containing C;
  - the child-degree of C as defined in Section 4;
  - a random value.

  Further, we consider ordering the objective variables in ascending and descending order.
- For unweighted instances, scoring functions using cost are not considered as cost is uniform. Further, for weighted instances, we consider either cost and then another scoring function, only the cost, or only the scoring function.
- The number of minimization rounds, either one or five. In the first round, the order of elimination is based on the scores. In subsequent rounds, the order is random. The best core according to cost, then size, then metric is chosen, where ties are broken arbitrarily.
- The number of classes for stratification, possible values for unweighted instances are no stratification or 10 classes. For weighted instances we consider the values no stratification, score-based stratification with 3 classes, cost-based stratification with 14 classes, and cost- then score-based stratification with 42 classes.
- The conflict limit for stratification, which is used for weight- and score-based stratification. We always use 2 650 for unweighted instances and 2 650, 100 000, or 1 000 000 for weighted instances.
- Randomized, where the scoring function assigns random scores, and the order of literals in the cardinality constraints is randomized.

5.2 Pre-Evaluation

The goal of the Pre-Evaluation phase is to determine which instances are very unlikely to be solved, as these instances do not provide useful data while taking up a large fraction of the total runtime. We first establish two good default configurations, one for unweighted and one for weighted instances, using the tool SMAC [18]. After running these configurations, we manually created 18 further configurations that complement the initial configurations, with the goal that combined, these 20 configurations would solve as many instances as possible.

After running all configurations, we discard all instances not solved by a single run. In total, 434 unweighted and 443 weighted instances remain.

5.3 Sampling

In the sampling phase we try to gather as many different OLL DAGs from OLL runs as possible. We manually create different configurations by varying the settings as stated in Section 5.1. In total, we use 113 different unweighted configurations of which 20 are based on randomization. Further, we use 129 weighted configurations, of which 37 use randomization.

Not every instance can be solved by every configuration. We expected to sample 49 042 OLL DAGs from unweighted instances and sampled 49 041, as not a single core could be found for one instance and configuration. In total 46 528 of the 49 042 runs were successful in finding the optimal solution. For weighted instances we expected 57 147 OLL DAGs and obtained 57 136, of which 53 148 were successful.

5.4 Timing

Algorithm 2 Timing instance (F, O) and set of cores $\mathcal{C}$.

$t \leftarrow 0$
$(F', O') \leftarrow (F, O)$
for $C \in \mathcal{C}$ **do**
    $t \leftarrow t + \text{time_SAT_call}(F', O')$
    $(F', O') \leftarrow \text{reformulate}(F', O', C)$
end for
$t \leftarrow t + \text{time_SAT_call}(F', O')$

The last phase establishes the solver-independent runtimes. We abstract away the MaxSAT solver and measure runtime based on only the SAT solver and the OLL DAG. We use Algorithm 2: we iterate over all cores and measure for each call how long a SAT call takes on the reformulated instance with a timelimit of five hours for all SAT calls. Only the sum of SAT runtimes is relevant for our measurement. Afterwards, a single SAT call is made that either results in a satisfiable result, whenever the original OLL run found the optimal solution, and unsatisfiable otherwise, if there is at least one more core.

The SAT call simulates the call necessary to establish the existence of another core. This abstracts away the use of most heuristics discussed in Section 4.1, except for the At-Most-One heuristic. Further, we do not perform hardening, as the upper bound is dependent on a heuristic result, and making insights dependent on the quality of the bound.

We use the SAT solvers Cadical 1.9.5 [11] and Glucose 3 [4] from PySAT 1.8.dev13 [14]. Both SAT solvers are commonly used in state-of-the-art OLL solvers [10]. Since there is some variance in SAT solving runtimes, we time each OLL DAG three times per SAT solver and, thereby, obtain 294 240 unweighted runtime samples and 342 810 weighted samples. Each sample has one of the following results:

1. *Successful*: The optimal solution has been found.
2. *Failed*: The lower bound does not reach the optimal solution and the timing exceeds the timeout.
3. *Inconclusive:* The lower bound does not reach the optimal solution, but timing finishes within the timeout. Hence, some heuristics like core minimization caused the timeout, not the reformulations.

In our analysis, we ignore the runtime of the last call in case it returned satisfiable and focus on the SAT solver calls corresponding to extracted cores.

6 Analysis

The analysis poses several challenges that require special consideration. The main challenge is that each instance defines its own distribution. Consequently, we cannot normalize and create one dataset over all instances, but instead analyze per instance and then aggregate the analyses. The other special consideration is that runtime is likely to be impacted by several structural properties. Hence, even though the value for one metric stays the same, the runtime may still vary.

Failed Runs

We cannot directly compare failed and successful runs, as the corresponding OLL DAGs are not comparable, since one of the DAGs is incomplete and the metrics would differ simply due to the different number of nodes. Since the two DAGs are from the same instance, we have samples from the same distribution and can normalize. We use the sample with the lowest runtime as the baseline. First, we divide each metric for each successful sample by the corresponding metric value of the baseline. Hence, whenever the sample has the same value as the baseline, it gets value 1 and values between 0 and 1 indicate values smaller than the baseline, while values above 1 indicate values above the baseline. Then, for each failed run we consider the *failure point* of the run: the number of cores extracted at the time of failure. We then gather the baseline's metrics for the OLL DAG at the failure point and normalize the failed run using the baseline's metric at this point. Using the fastest run gives stability towards outliers.

Runtimes

Runtimes have a large variance, even when considering only one SAT solver. We reduce noisiness in our results and focus on harder instances, by discarding every instance from our analysis where every run finishes within 300 seconds. The analysis results are similar for any limit that is at least one second[3]. The Glucose and Cadical runtimes are rarely the same and correlate very weakly. We use the average over the three Glucose and three Cadical runtimes in our analysis. Nonetheless, the analysis results remain similar when using only Cadical or using only Glucose runtimes.

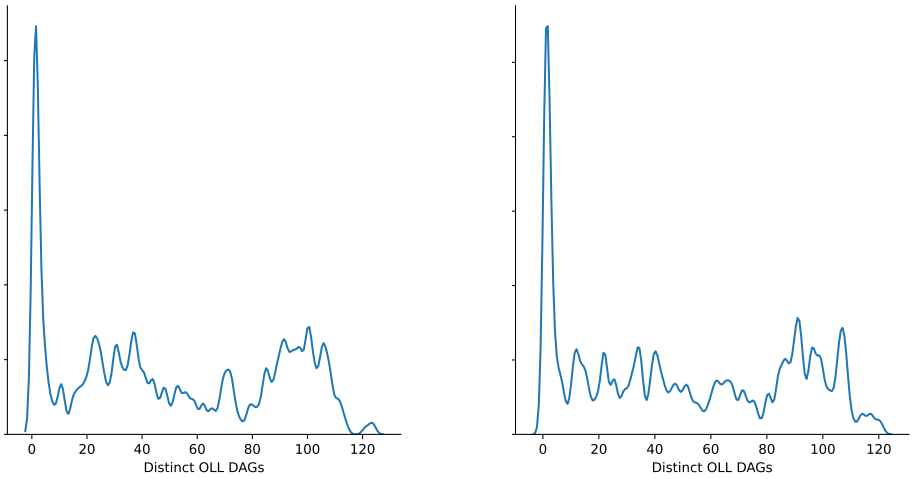
6.1 Variance Analysis

The first step in our analysis is ensuring that we are indeed able to have enough variance in our results. We first remove duplicate OLL DAGs. From the 48 914 gathered OLL DAGs for unweighted instances 21 657 are distinct. For weighted instances the respective numbers are 56 193 and 21 855. Figure 1 shows how these distinct OLL DAGs are distributed over the instances. While we are able to sample several distinct OLL DAGs for most instances, we only obtained a single distinct OLL DAG for 46 unweighted and 46 weighted instances. It is not clear if these instances allow for more than one distinct OLL DAG, as some instances, e.g., instances where we obtain only unit cores, do not allow for different OLL DAGs.

Further, we analyze how much our metrics vary over these distinct OLL DAGs. We normalize our data separately by instance as stated above and then rescale the values such that the same value as the fastest run results in 0, positive values indicate higher values, negative values indicate lower values. Precisely, a value of 1 signifies that the value is double that of the baseline and -1 indicates a halving of the baseline. From there on the values increase such that 2 indicates triple the baseline and -2 one third.

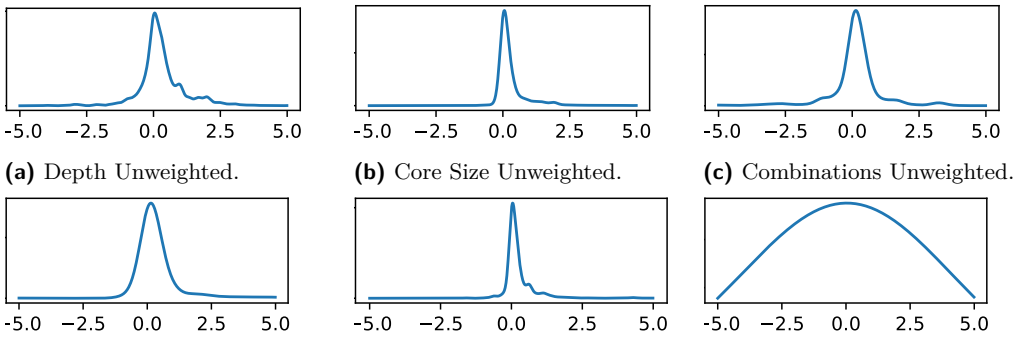
The density plots in Figure 2 show that we are indeed able to sample OLL DAGs with considerable variance in the metric values. In case of the number of combination, the logarithm was used due to the size of the values. The majority of samples has a similar value to the baseline and we have many samples that are within half and double the baseline, often wider. The figures are zoomed in and for all the displayed metrics, we have several outliers as far as ten times or more of the baseline.

[3] We give more results on this in the supplementary material.



(a) Unweighted. **(b)** Unweighted.

Figure 1 Density plots showing the number of distinct OLL DAGs found per instance.



(a) Depth Unweighted. **(b)** Core Size Unweighted. **(c)** Combinations Unweighted.

(d) Depth Weighted. **(e)** Core Size Weighted. **(f)** Combinations Weighted.

Figure 2 Density plots for different metrics for unweighted and weighted instances.

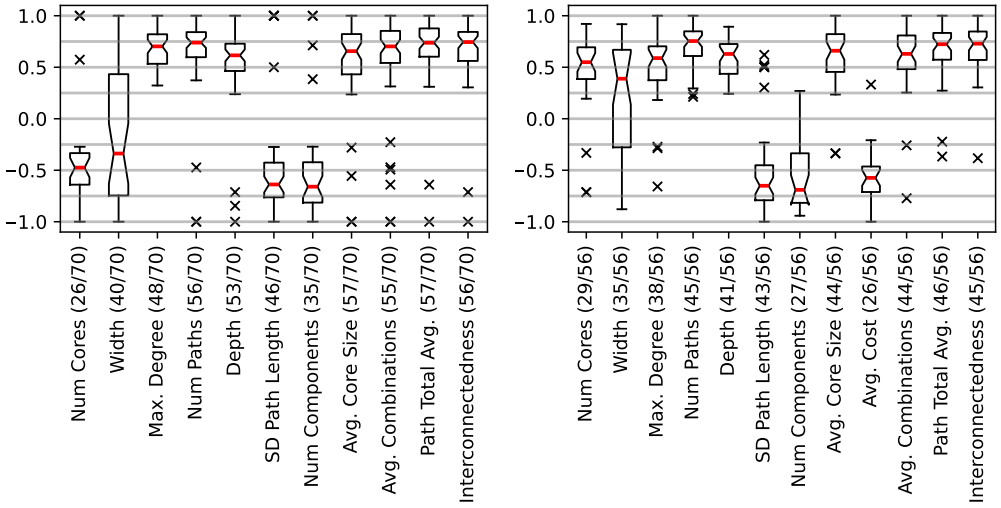
Our results indicate that the heuristics' impact on the metrics is largely as we expected. A notable exception is that disjoint cores does not seem to correlate well with the width of the OLL DAG. However, our experimental setup cannot conclusively confirm this impact, as this would require an ablation study where only configurations varying a single heuristic are used. Due to the already large number of configurations, this was out-of-scope for this paper.

6.2 Runtime Analysis

In this analysis, we ask which metrics correlate well with the runtime. We compute for each metric and instance pair the correlation coefficient. We use the Spearman correlation, as we do not expect a linear correlation, but a rank correlation. Figure 3 shows the results per metric for unweighted and weighted instances respectively[4]. The graph only contains results, where the p-value is smaller or equal than 0.05, the number of instances where the p-value was small enough is indicated in the labels. A high p-value is mainly caused by too

[4] Further boxplots in the appendix show that these results are stable given different parameterizations.

few samples, as too many runs are inconclusive, or too little variation in the metric's value. Further, our hypothesis is that changes in the metric correlate with changes in the runtime. This is slightly different from correlation, as we do expect changes in the runtime, even if the metric does not change. Hence, high correlation confirms our hypothesis, but potentially to a higher degree than the correlation coefficient shows.



(a) Unweighted. **(b)** Weighted.

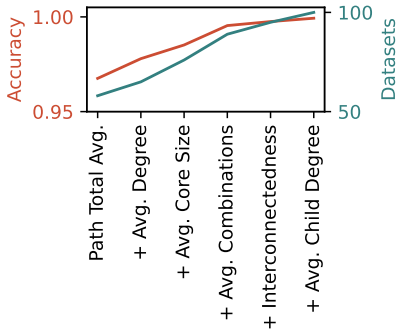
Figure 3 Boxplot of runtime correlations over different instances. The labels shows for how many instances the result was statistically significant, insignificant results are not included.

The results show that high OLL DAG complexity indeed correlates well with high runtime. The best metric is the number of paths in the OLL DAG, as it has a high correlation that does not vary too much from instance to instance and is statistically significant on most instances. Interconnectedness performs almost as well. Surprising is the good correlation for the standard deviation over the longest path lengths among the source-sink paths and the overall inconclusive correlation of the DAG's width.

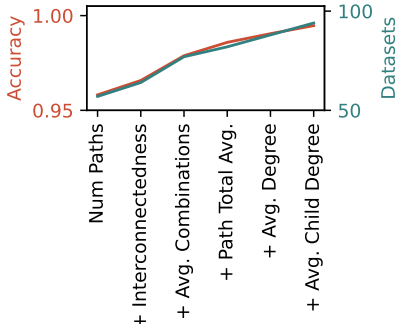
The OLL DAG analysis can also capture important properties that are not directly associated with the graph structure, such as the average number of combinations, which also has a good correlation with the runtime. Further, the average cost tends to correlate with low runtime as is expected. However, many results are statistically not significant and for the remainder the correlation varies a lot.

The negative correlation of the number of cores for unweighted instances is interesting. As the number of cores extracted in a successful run is always the same for unweighted instances, the only reason for this metric to vary is core exhaustion. Therefore, core exhaustion correlates with higher runtime. This does not necessarily indicate that core exhaustion is detrimental to good runtime, but rather that core structures without exhaustible cores might be preferable.

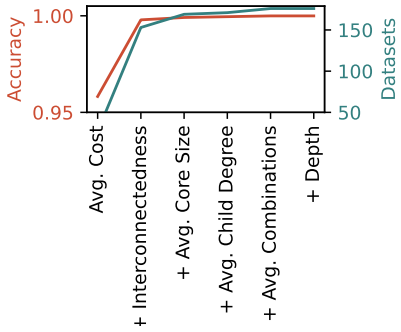
Overall, our results show that the OLL DAG is indeed a good model for the complexity arising from reformulation, as many metrics have a high correlation with the runtime. Selecting cores that result in a simple OLL DAG structure seem to correlate well with low runtime.



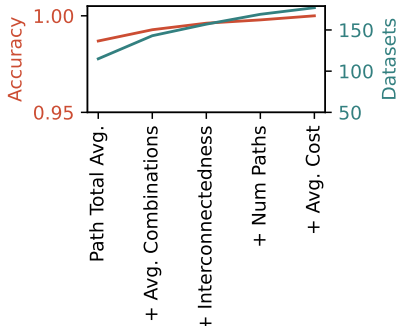
(a) Decision tree/Unweighted.



(b) Decision List/Unweighted.



(c) Decision Tree/Weighted.



(d) Decision List/Weighted.

Figure 4 Constrained models explaining failed runs.

6.3 Reasons for Failure

In our last analysis we want to find indicators for a run failing or succeeding. For this purpose, we analyze the 32 unweighted instances and 23 weighted instances where we have both successful and failed runs. We compare runs on these instances by analyzing the OLL DAGs at the various failures points, i.e., when the runs extracted the same number of cores. We treat each instance and failure point as a separate classification dataset and thereby obtain 101 datasets for unweighted instances and 177 datasets for weighted instances. Each run at a specific failure point is represented by one sample. 87.0% of the 5 667 samples in the unweighted datasets are from successful runs as are 95.2% of the 9 430 samples in the weighted datasets.

The first question we ask is how well any metric on its own can explain the failures. Thus, we analyze how well a single metric can separate all the samples in all datasets into successful and failed samples, i.e., how well we can *split* the datasets using only one metric and a different value per dataset. We measure this using the number of datasets that are fully explained, i.e., after the split we have only successful samples in one half and only unsuccessful in the other. Further, we are interested in the *accuracy*: how many samples agree with the majority class in their half.

For unweighted instances the best metric is the path total average – the average over all nodes of the number of source-sink paths passing that node – explaining 58 datasets and achieving an accuracy of 96.9%. The samples from weighted instances are easier to explain: the average child degree explains 124 datasets and achieves over 99.0% accuracy.

Next, we explore if the combination of metrics can explain more failures. We use specialized decision trees for this purpose, where the decision tree for each dataset has to split on the same sequence of metrics in each branch and can split on each metric at most

once. The decision trees and branches only differ in the values they split on. Further, we also use decision lists, i.e., decision trees that have only one branch with predicate nodes, thereby restricting the model even further. We enforce the same sequence of metrics in every decision tree such that we obtain explanations that apply to all datasets. The respective results are in Figures 4a to 4d.

The datasets from the weighted instances are easier to explain. Although a bad split on its own, average cost in conjunction with metrics measuring the graph complexity can explain almost all failures. This indicates that low average cost might not lead to failures, as long as the OLL DAG remains comparatively simple. Further splits manage to explain all remaining datasets, with decision tree and list performing almost the same. The decision list's splits must separate the samples such that one of the halves contains only a single class, which will become the leaf. Hence, the average cost is not a viable split, and the combination of the path metric and the number of combinations can already explain almost all failures.

The datasets from unweighted instances are much harder to classify correctly. The two models agree mostly on the important metrics in different orders. The number of paths is again very important, followed by in-degree metrics, either the average degree or child degree. The average number of combinations is important as is the interconnectedness, but to a lesser degree than the path and degree metrics. This suggests that a large number of paths and the resulting DAG structure is the best indicator for a failing unweighted run.

We analyze the OLL DAGs of instances with hard to explain failures further, by looking at the instances individually. On those instances, specific combinations of metrics are important. E.g., a large number of cores does not lead to a failure as long as the number of paths stays comparatively low; OLL DAGs with a comparatively large number of paths do not fail, as long as the core sizes are comparatively small and the number of combinations low. Hence, these instances do not offer a simple reason for failures, but require complex explanations. However, the OLL DAG still contains the necessary information. This shows that the explanations provided by the decision models are overall very succinct, as they capture all instances we analyzed, while explanations for single instances may already require three or more metrics as an exhaustive explanation.

7 Conclusion and Future Work

We showed that the OLL DAG is a good abstraction to analyze the impact of different possible reformulations of a MaxSAT instance. This abstraction allowed us to collect a large number of OLL DAGs and analyze why some OLL runs perform well and others do not. This analysis revealed that certain structural properties of the OLL DAG, foremost the number of root-leaf paths, do indeed have a strong correlation with solving performance.

We see several avenues of future research based on OLL DAGs. The first one would be to develop heuristics that specifically target structural properties using and extending our results. Further, we have seen that complex structure correlates with high runtime. This is not always true, as width has a very weak correlation. Hence, further distinguishing which structural complexities correlate with bad performance is of great interest. The large number of duplicate OLL DAGs also shows how difficult it is to sample distinct OLL DAGs. Further methods for varying the OLL DAGs might provide more insights and these methods could also lead to performance improvements. Finally, we are interested in incorporating instance specific properties for a more focused analysis.

References

1. Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in clasp. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *LIPIcs*, pages 211–221. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.ICLP.2012.211.
2. Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving sat-based weighted maxsat solvers. In Michela Milano, editor, *Principles and Practice of Constraint Programming – 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2012. doi:10.1007/978-3-642-33558-7_9.
3. Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving WPM2 for (weighted) partial maxsat. In Christian Schulte, editor, *Principles and Practice of Constraint Programming – 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 117–132. Springer, 2013. doi:10.1007/978-3-642-40627-0_12.
4. 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.
5. Fahiem Bacchus and Nina Narodytska. Cores in core based maxsat algorithms: An analysis. 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 7–15. Springer, 2014. doi:10.1007/978-3-319-09284-3_2.
6. Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 – October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003. doi:10.1007/978-3-540-45193-8_8.
7. Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided maxsat solving. In Brigitte Pientka and Cesare Tinelli, editors, *Automated Deduction – CADE 29 – 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2023. doi:10.1007/978-3-031-38499-8_1.
8. Jeremias Berg and Matti Järvisalo. Weight-aware core extraction in sat-based maxsat solving. 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 652–670. Springer, 2017. doi:10.1007/978-3-319-66158-2_42.
9. Jeremias Berg, Matti Järvisalo, Benjamin Kaufmann, Ruben Martins, Andreas Niskanen, and Tobias Paxian. MaxSAT Evaluation 2024. Presented at SAT 2024, 2024. URL: https://maxsat-evaluations.github.io/2024/mse24-talk.pdf.
10. Jeremias Berg, Matti Järvisalo, Benjamin Kaufmann, Ruben Martins, Andreas Niskanen, and Tobias Paxian. MaxSAT Evaluation 2024 : Solver and Benchmark Descriptions. Technical report, University of Helsinki, 2024.
11. Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. Cadical 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification – 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.
12. Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In Ofer Strichman and Armin Biere, editors, *First International Workshop on Bounded Model Checking, BMC@CAV 2003, Boulder, Colorado, USA, July 13, 2003*, volume 89(4) of *Electronic Notes in Theoretical Computer Science*, pages 543–560. Elsevier, 2003. doi:10.1016/S1571-0661(05)82542-3.

13 Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 – March 1, 2017*. The Internet Society, 2017. URL: `https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/automated-synthesis-semantic-malware-signatures-using-maximum-satisfiability/`.

14 Alexey Ignatiev, António Morgado, and João Marques-Silva. Pysat: A python toolkit for prototyping with SAT oracles. 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 428–437. Springer, 2018. `doi:10.1007/978-3-319-94144-8_26`.

15 Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient maxsat solver. *J. Satisf. Boolean Model. Comput.*, 11(1):53–64, 2019. `doi:10.3233/SAT190116`.

16 Sepideh Khoshnood, Markus Kusano, and Chao Wang. ConcBugAssist: constraint solving for diagnosis and repair of concurrency bugs. In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 165–176. ACM, 2015. `doi:10.1145/2771783.2771798`.

17 Stepan Kochemazov, Victor Kondratiev, and Irina Gribanova. Empirical analysis of the RC2 maxsat algorithm. In *46th MIPRO ICT and Electronics Convention, MIPRO 2023, Opatija, Croatia, May 22-26, 2023*, pages 1027–1032. IEEE, 2023. `doi:10.23919/MIPRO57284.2023.10159820`.

18 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. *J. Mach. Learn. Res.*, 23:54:1–54:9, 2022. URL: `https://jmlr.org/papers/v23/21-0888.html`.

19 João Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *40th IEEE International Symposium on Multiple-Valued Logic, ISMVL 2010, Barcelona, Spain, 26-28 May 2010*, pages 9–14. IEEE Computer Society, 2010. `doi:10.1109/ISMVL.2010.11`.

20 João Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: algorithms & applications. *Ann. Math. Artif. Intell.*, 62(3-4):317–343, 2011. `doi:10.1007/S10472-011-9233-2`.

21 Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for maxsat. 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 531–548. Springer, 2014. `doi:10.1007/978-3-319-10428-7_39`.

22 António Morgado, Carmine Dodaro, and João Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. 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 564–573. Springer, 2014. `doi:10.1007/978-3-319-10428-7_41`.

23 António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided maxsat solving: A survey and assessment. *Constraints An Int. J.*, 18(4):478–534, 2013. `doi:10.1007/s10601-013-9146-2`.

24 Alexander Nadel. Anytime weighted MaxSAT with improved polarity selection and bit-vector optimization. In Clark W. Barrett and Jin Yang, editors, *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, pages 193–202. IEEE, 2019. `doi:10.23919/FMCAD.2019.8894273`.

25 Nina Narodytska and Nikolaj S. Bjørner. Analysis of core-guided maxsat using cores and correction sets. 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 26:1–26:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.SAT.2022.26.

26 Xujie Si, Xin Zhang, Radu Grigore, and Mayur Naik. Maximum satisfiability in software analysis: Applications and techniques. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification – 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 68–94. Springer, 2017. doi:10.1007/978-3-319-63387-9_4.

27 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. doi:10.3233/978-1-58603-929-5-131.

28 Charlie Shucheng Zhu, Georg Weissenbacher, Divjyot Sethi, and Sharad Malik. Sat-based techniques for determining backbones for post-silicon fault localisation. In Zeljko Zilic and Sandeep K. Shukla, editors, *2011 IEEE International High Level Design Validation and Test Workshop, HLDVT 2011, Napa Valley, CA, USA, November 9-11, 2011*, pages 84–91. IEEE Computer Society, 2011. doi:10.1109/HLDVT.2011.6113981.

Streamlining Distributed SAT Solver Design

Dominik Schreiber
Karlsruhe Institute of Technology, Germany

Niccolò Rigi-Luperti
Karlsruhe Institute of Technology, Germany

Armin Biere
University of Freiburg, Germany

Abstract

Distributed clause-sharing SAT solvers have recently been established as powerful automated reasoning tools that can conquer previously infeasible instances. A common design of distributed SAT solvers is to run many off-the-shelf sequential solvers in parallel, employ some diversification (e.g., restart intervals or decision orders), and share conflict clauses among the solver threads. This approach, naïvely, adopts all best practices of sequential solver design for distributed solving, where these practices may be less useful or even actively detrimental. In this work we diagnose such shortcomings in the state-of-the-art system MallobSat and propose first effective mitigations. In particular, we replace the redundant pre– and inprocessing at all threads with single-core preprocessing that runs next to the parallel search, remove LBD values from the clause-sharing operation, and slim down solver diversification to very few lightweight and uniform methods. Experimental evaluations on up to 3072 cores (64 nodes) confirm that our measures improve performance while also drastically simplifying the SAT solving program that is run in parallel.

2012 ACM Subject Classification Hardware → Theorem proving and SAT solving; Theory of computation → Distributed algorithms

Keywords and phrases Satisfiability, parallel SAT solving, distributed computing, preprocessing

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.27

Supplementary Material
Software (Source Code): `https://github.com/nrilu/SAT_2025_supplement_63` [54]
archived at `swh:1:dir:bdecc26fc577de49d3fe853d07e4ba59e3cbe2f8`

Funding Research reported in this paper was supported by an Amazon Research Award (Fall 2023).

Acknowledgements The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (`https://www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (`https://www.lrz.de`).

1 Introduction

Over the last years, parallel and distributed approaches to propositional satisfiability (SAT) solving have gained significant traction and established themselves as powerful automated reasoning tools to conquer previously infeasible problems [22, 56]. In large parts, this success is due to careful exchange of *conflict clauses* that are learned by individual sequential SAT solver threads during their search [41]. Recent findings [56] indicate that such clause sharing can serve as its own powerful kind of parallelization, even if all solver threads are initially completely identical, challenging the predominant view that solver threads need to either explicitly operate on different sub-spaces (*search space splitting*) or be explicitly *diversified* in terms of their approaches (*portfolio*) [56]. In light of these new insights, we explore ways to improve both simplicity *and* performance of distributed SAT solving.


© Dominik Schreiber, Niccolò Rigi-Luperti, and Armin Biere;
licensed under Creative Commons License CC-BY 4.0



State-of-the-art parallel and distributed solvers commonly employ cutting-edge SAT solvers as sequential building blocks. These sequential solvers, most recently CaDiCaL [11] and Kissat [12] in particular, are highly engineered pieces of software optimized for sequential, general-purpose SAT solving performance [26]. To this end, they feature a large number of simplification or *preprocessing* techniques, some of which are heuristically scheduled into their solver program (*inprocessing*) [17]. In most established parallel solvers, these tasks are performed by all solver threads in a fully redundant fashion, with the exception of some coordinated preprocessing [21, 49] and disabling individual techniques for individual threads in the name of diversification [4, 55]. While inprocessing significantly improves sequential SAT solving performance [17], its impact for parallel and distributed clause-sharing solving is largely unexplored (cf. [20]). One notable caveat is that pre– and inprocessing tasks, performed redundantly at all solver threads, are not parallelized and can therefore present a scalability bottleneck. This holds especially if the effort spent on a pre–/inprocessing task is not a fixed, small *share* of each thread's CPU time (e.g., 10%) but rather takes up large portions of the *total, parallel* CPU time – which we do confirm to be the case for some techniques. Another aspect is that simplification techniques often heavily modify the original formula. When two solvers operate on different representations of the problem, efficient and effective information exchange can be obstructed. For instance, Schreiber and Sanders noticed that bounded variable elimination (BVE) can prevent solver threads from importing many incoming clauses because the clauses feature variables that are locally already eliminated [56].

In this work, we diagnose and mend some of the mentioned shortcomings in modern distributed SAT solver design. A large portion of our contributions are of an analytical and observational nature. In particular, we profile the cutting-edge distributed SAT solver MallobSat [55, 56] in terms of the amount of time individual solver threads spend in different tasks. We observe that pre–/inprocessing provides little to no net benefit for distributed solving and that disabling it does in fact improve performance. Similarly, we complement recent findings [35, 50] questioning the significance of the Literal Block Distance (LBD) [2] metric for clause-sharing solving. We subsequently propose first improvements as a reaction to these insights: We implement a single-core preprocessing approach that runs while "pure" CDCL solving is already ongoing, remove LBDs from distributed clause sharing, and drastically cut diversification of solver threads down to two trivial methods (random seeds and variable phases). Our enhanced setup of MallobSat outperforms the prior state of the art, especially on application-oriented benchmarks, while only executing pure, uniform, clause-sharing-powered CDCL search at the parallel cores – further departing from a "portfolio solver". Going forward, our results call for designing distributed SAT solvers in a more holistic and unified manner, whereas our proposed solving approach opens up new opportunities for understanding, analyzing, and advancing distributed clause-sharing SAT solving.

The paper at hand is structured as follows. Section 2 introduces relevant preliminaries and related work. Two empirical studies follow, first on examining pre–/inprocessing in distributed SAT solving (Section 3) and secondly on the role of LBDs (Section 4). In Section 5, we propose an alternative approach to integrating simplification techniques into distributed SAT solving in a more scalable manner. Section 6 features a thorough experimental evaluation. We discuss results in Section 7 and conclude our work in Section 8.

2 Background

In the following, we provide some necessary background for the work at hand.

Given a propositional formula $F = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{k_i} l_{ij}$, where each l_{ij} is a Boolean variable or its negation, the propositional satisfiability (SAT) problem is to find a variable assignment that satisfies F or, if impossible, to report *unsatisfiability* [16]. Most sequential state-of-

the-art SAT solvers are based on the *Conflict-Driven Clause Learning* paradigm [41], which encompasses a heuristic search over the space of partial variable assignments. When a CDCL solver encounters a logical conflict under its current assignment, it can derive a so-called *conflict clause* and add it to its knowledge base. Intuitively, such a clause represents a *pruned sub-space* of search space that has been deemed unsatisfiable.

In *distributed computing*, multiple machines with no shared memory perform a joint computation. They usually coordinate by exchanging messages over a network. Parallel and distributed approaches to SAT solving have mostly been separated into two major classes of approaches: search space splitting and portfolio solving [5]. *Search space splitting* (a.k.a. search space partitioning, divide&conquer, later cube&conquer [34]) was the earliest parallelization technique, forcing workers to search pairwise disjoint sub-spaces of search space. While this is situationally powerful [31, 33], load balancing problems can occur if the problem is split in an uneven or ineffective manner [57]. *Portfolio solving* has been proposed to circumvent these problems, instead aiming to achieve speedups by running many *different* sequential SAT solvers on the same, original, formula [30]. Both parallelization paradigms have been enhanced by *clause sharing*, i.e., mechanisms to exchange learned conflict clauses across workers [30, 58]. The result is a form of *parallel search space pruning*: If solver thread A learns clause c and transmits it to thread B, then B will not re-explore the sub-space prohibited by clause c. This pruning is imperfect in nature since solver threads periodically *forget* (i.e., discard) clauses, losing pruning information in the process [2, 43]. Earlier works still showed that clause sharing is beneficial for performance of portfolios [4, 30] (resulting in *clause-sharing portfolio solvers*) and search space splitting solvers [58].

Modern distributed SAT solving systems include PRS [62], Painless [48], and MallobSat [55, 56], which generally fit the clause-sharing portfolio paradigm. MallobSat in particular features a highly engineered form of periodic *all-to-all* clause sharing. Its authors demonstrated that this clause sharing is the central driver of the system's scalability. Notably, MallobSat performs rather competitively even if *all solver threads are initially identical* – down to the exact same seed – the reason being that individual threads import slightly different clause sets at slightly different points in time due to the non-determinism of the distributed program. This alone suffices for clause sharing to act as effective search space pruning [56]. While this finding does not immediately translate to a new distributed solving approach – usual diversification of solvers still benefits performance – it raises the question of whether the notion of *clause-sharing portfolios* still applies to a system that does not depend on any actual portfolio. Schreiber & Sanders suggested the term *clause-sharing solver* [56] to emphasize that clause sharing is, demonstrably, its very own kind of parallelization and should not be considered a mere supplement to portfolio solving or search space splitting.

We refer to the Handbook of Satisfiability [17] for an overview of pre– and inprocessing techniques. Regarding *parallel* pre- and inprocessing, Gebhardt et al. [27] presented a shared-memory parallelization of certain techniques including *subsumption*, *clause strengthening*, and *bounded variable elimination* (BVE). The scalability of this approach is rather limited due to the employed memory lock schemes. Wieringa et al. [60] showed that vivification-like inprocessing could be elegantly achieved in a parallel fashion by adding a second pure clause strengthening thread to each CDCL solver. The ParKissat system [61] applies some single-core preprocessing prior to parallel solving, including techniques to handle XOR and cardinality constraints, which was adopted by PRS [21] and subsequently by the state-of-the-art shared-memory solver PL-PRS-BVA-KISSAT [48]. The latter system also runs a number of *bounded variable addition* (BVA) [29, 40] threads next to CDCL [49]. The results from

all but one BVA thread are discarded; other threads can seamlessly integrate the "winning" result in their ongoing CDCL search. To our knowledge, the solver threads in the mentioned systems still perform inprocessing during search, on top of the coordinated preprocessing.

Focusing on shared clauses, Vallade et al. [59] proposed to boost sharing efficiency by concurrently strengthening *shared* clauses since, in a parallel setting, these might be the most rewarding clauses to invest work in. They observed improved performance for unsatisfiable but negative results for satisfiable instances. Iida et al. [35, 36] recently tested several metrics for selecting clauses to share. In contrast to the older LBD metric [2], which depends on the solver state [3], their proposed metrics are based on the formula's (static) graph structure.

Osama et al. [44] introduced utilizing GPUs for pre-/inprocessing, achieving speedups by more than an order of magnitude for garbage collection and gate detection. For variable elimination however, the speedups were limited to around 1.5, highlighting the difficulty of uniting the rather irregular program flow of SAT solving with monolithic GPU architectures.

3 Examining Pre–/Inprocessing in Distributed SAT

In the following, we present and analyze running time profiles of a state-of-the-art distributed clause-sharing solver in order to gain insights in its behavior and directions for improvements.

We use MallobSat, essentially configured as in the International SAT Competition 2024 [51] but using only 2024 Kissat [12] as its solver backend. In the scope of this work, we decided to focus on a single cutting-edge solver backend (Kissat) in order to obtain clean results and keep the experimental design space more manageable. We use the HPC cluster SuperMUC-NG, where each node features a two-socket Intel Skylake Xeon Platinum 8174 processor clocked at 2.7 GHz with 48 physical cores (96 hardware threads) and 96 GB of RAM. For the following experiments, we use 16 nodes (768 cores), which amounts to a decently large scale and still allows us to run more experiments than we could at larger scales.[1] In order to obtain profiling results, we make use of Kissat's integrated profiling capabilities: each solver thread bookkeeps the accumulated time it spends in relevant sub-procedures. Given that used CPU time is challenging to obtain asynchronously for a solver thread in MallobSat's multi-threaded processes, we changed Kissat's profiling to use *wallclock* rather than CPU time.

In terms of problem inputs, we need to balance responsible use of computational resources with instance diversity and robustness of results [51]. To ensure a great variety of benchmark families, we consider all 796 unique instances from the last *two* SAT Competitions, 2023 and 2024. We then reduce the resulting benchmark set to a random selection of 396 instances – reserving the remaining 400 instances for scaling experiments (Section 6).

Fig. 1 shows the ratio of time the solver threads spend in selected tasks for single-core Kissat (left) and for 768-core MallobSat (right) for each of the 396 instances. Fig. 2 (left) shows according statistics; more details are provided in the appendix (Tab. 2). First, it is worth mentioning that Kissat, as other sequential solvers participating in the SAT Competition, is tuned for running times up to 5000 s. At distributed scales, we usually aim for much lower running times on the order of 300 s [56]. For such time scales, the inprocessing scheduling may turn out suboptimal. We thus ran Kissat for up to 5000 s (vs. 300 s for MallobSat) to aggregate more representative statistics for it.

[1] We acknowledge that we spent a total of 709 880 core-hours for all experiments in the paper (including testing and debugging), which amounts to ≈2.3 tons of CO_2eq (320 g CO_2eq / kWh) – similar to the emissions caused by six economy passengers on a flight from Frankfurt to New York. The SAT'2021 Mallob paper [55] spent a similar amount of resources (also in the order of 700k ch).

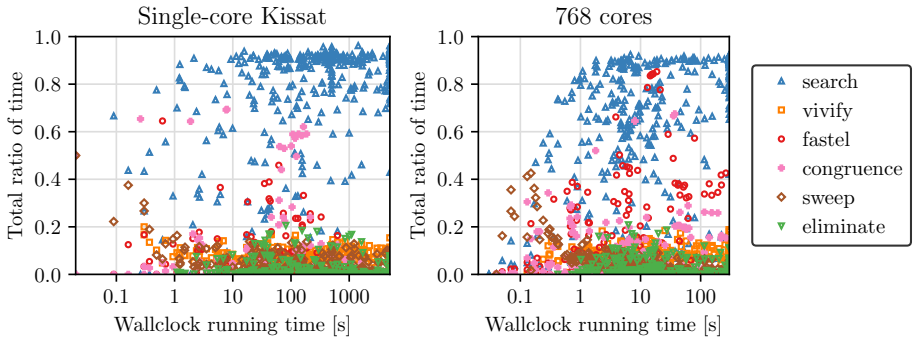


Figure 1 By-input distribution over the total solver time ratio spent in actual (CDCL) search and in selected pre–/inprocessing tasks, for single-threaded Kissat (left) and at 768 cores (right), relative to overall (wallclock) running time.

	Seq. Kissat		768 cores	
Category	mean	total	mean	total
search	66.74	86.32	63.29	79.15
vivify	6.30	7.03	8.32	8.35
fastel	0.30	0.30	0.76	2.96
congruence	0.24	0.46	0.55	1.85
sweep	2.08	2.18	2.88	1.54
eliminate	0.31	1.01	0.75	1.33

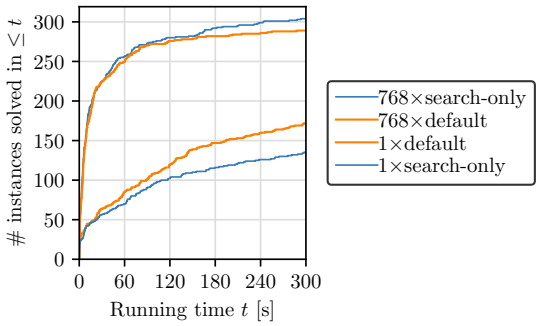


Figure 2 Left: Percentages of time spent in selected tasks, for sequential Kissat and 768-core MallobSat, in terms of the geometric mean over all non-zero time ratios reported across all solver threads and inputs ("mean") and in terms of adding up the total time all solver threads spent on a technique and then dividing by the sum of total running times across all solver threads and inputs ("total"). Right: Performance with and without pre–/inprocessing, at one and 768 cores each.

The label "`search`" in Fig. 1 denotes actual CDCL search, which is the only parallelized task (via clause sharing). Visually, the markers for sequential Kissat corresponding to search are strongly clustered at about 0.9, while they appear lowered and more scattered for distributed MallobSat. The total ratio of solver time spent in search is 86% for single-core Kissat but only 79% for 768-core MallobSat. By contrast, most pre–/inprocessing techniques [12] take proportionally more time in our distributed setting. Some of the most costly tasks are vivification [39], initial fast variable elimination ("fastel") and, to a lesser extent, congruence closure, SAT sweeping, and variable elimination during inprocessing. For example, the relative cost of vivification increases from 7% at one core to 8.4% at 768 cores, possibly because of the increased number of vivification candidates due to clause sharing.

Other techniques however, such as initial variable elimination, do not follow this logic and are expected to incur a *fixed* amount of work. At a distributed scale, where CDCL search profits from speedups, the redundant and non parallelized preprocessing on all cores accounts for increasing portions of the total work performed (from 0.3% to 2.96% for "fastel"). SAT sweeping is an exception to this effect (relative cost *decrease* from 2.18% to 1.54%), the probable reason being that it only incurs significant relative cost at very low running times (< 1 s), where MallobSat performs similar or even worse than sequential Kissat. Also note how some tasks such as vivification use relatively consistent shares of time since Kissat can

control their computational budget in an accurate manner. Other tasks that are highly dependent on problem structure, such as initial variable elimination, show high variances in the time spent and occasionally even account for the majority of compute time.

The impact on solving performance is shown in Fig. 2 (right). We ran MallobSat as before and also in a search-only configuration where pre– and inprocessing is disabled for all threads. At a sequential scale, Kissat with pre–/inprocessing significantly outperforms its search-only variant, as one would expect. At 768 cores, however, many instances are slowed down by the cost of preprocessing and simplification. As such, our "search-only" setup achieves a geometric mean speedup of 3.7% over the default setup (computed over commonly solved instances) and solved 12 more instances. Let us discuss some potential causes for this effect:

- As presented above, certain pre–/inprocessing techniques can constitute a large share of the *total* (parallel) work performed, whereas their merit remains constant regardless of the number of cores computing it. Due to this scalability bottleneck, the techniques incur increasing opportunity cost that eventually outweighs their benefits: Our search-only setup raises the share of time spent on search from 79.15% to 95.4% (appendix Tab. 2).
- Clause sharing may functionally "simulate" certain pre–/inprocessing techniques to some degree. This is plausible for techniques related to identifying short clauses that are implied by the input formula, such as subsumption or vivification: Since MallobSat's clause sharing aggregates the globally shortest distinct clauses [56], it may become unlikely that vivifying a clause results in a shorter clause that *no other solver thread* has found so far. That being said, we are yet to find specific evidence for (or against) this suspicion.
- Deviating preprocessing levels across solver threads may result in "language barriers" across solvers, hindering cooperation and rendering clause sharing less effective. This may especially apply to techniques eliminating variables, which restrict the clauses that a solver thread can subsequently import [56]: In our 768-core experiment with pre–/inprocessing, the geometric mean ratio of incoming clauses that were imported successfully by a solver thread is 62%. The median ratio is higher (75%) since every tenth solver thread is configured to skip variable elimination completely and can therefore import all clauses. Still, about 22% of solver threads could import *less than half* of their incoming clauses.

In terms of benchmark families [37], we observed that pre–/inprocessing is most beneficial for unsatisfiable miter instances (i.e., combinational circuit equivalence checking; nine instances with mean speedup 3.4 and two more instances not solved by "search-only"), which is unsurprising since Kissat's SAT sweeping and congruence closure are designed for this exact application [13, 14]. Notable benchmark families where the search-only setup performed better include satisfiable instances on scheduling (9 instances, speedup 1.59) and Folkman graphs (5 instances, speedup 1.83). For the latter, the mean time ratio spent on CDCL search increases from 89.1% to 99.2% when disabling pre–/inprocessing, mostly due to vivification (7.7%). Additional family-specific results are given in the appendix (Table 3).

All in all, the presented results paint a clear picture: Pre– and inprocessing techniques are currently being employed in distributed SAT solving in a suboptimal manner. They are neither designed nor tuned for clause-sharing solving and incur significant cost which limits the attainable speedup and, in many cases, outweighs their benefits.

4 Revisiting LBDs in Distributed Solving

In sequential solving, the Literal Block Distance (LBD) of a clause indicates the number of distinct decisions involved in the according conflict [2]. LBDs have proven useful as a quality metric, i.e., to decide whether to keep or to discard a clause [43]. LBDs have also been used

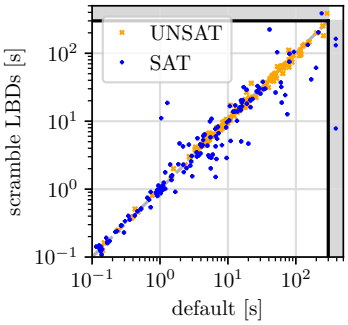


Figure 3 By-instance solving time comparison between a default run of MallobSat and a run where LBD scores of shared clauses are shuffled within each clause length, both at 768 cores.

in many parallel SAT solvers [1, 4, 25, 38] as a metric to decide which learned clauses a solver unit should contribute to clause sharing. However, the usefulness of LBDs does not necessarily translate to parallel clause sharing: A clause's LBD depends on the solver state and may thus not be meaningful to a different solver (thread) [3]. As recently evidenced, using LBD as a clause selection metric is not beneficial compared to clause length [35, 55]. MallobSat currently uses LBDs only as a tie-breaker, after clause length, when prioritizing clauses to share. Its authors also reported very similar performance for different ways of manipulating the LBD values of *incoming* shared clauses, just before a solver thread imports them [56]. However, some of the underlying tests have the potential flaw that they do not only change individual LBD values but also their overall *distribution*. This distribution can be relevant since it influences the volume of kept clauses and, therefore, the propagation speed and search behavior. Changing it, e.g., by assigning random LBD values to incoming clauses, may thus result in side effects which improve or degrade performance.

In the following, we examine two aspects of LBDs in distributed solving separately: The significance of a particular incoming clause carrying a particular LBD, and subsequently the *distribution* over the LBDs of incoming clauses.

To reexamine the significance of individual LBDs, we suggest the following experiment: After globally aggregating all clauses to share, we *shuffle* the mapping of clauses to LBD values *within* each clause length uniformly at random. As such, we can cleanly observe the effect of disconnecting LBDs from their meaning while the distribution over LBD values in solver threads remains fixed. As Fig. 3 shows, we were unable to discern any meaningful differences in solving performance with vs. without shuffling of LBD scores: Shuffling LBD scores resulted in one additional solved instance (290 vs. 289) and a speedup of 1.27%. This complements recent evidence [35, 56] that communicating and forwarding the LBDs of individual shared clauses does not carry any useful information in parallel clause-sharing.

Given that *individual* LBD values do not appear to be of any significance, let us now separately evaluate the impact of different *distributions* over incoming LBD values. Specifically, we explore different strategies to fabricate LBDs synthetically. For a synthetic distribution to be sensible, our intuition is that high-LBD clauses should be more frequent, low-LBD clauses should be rare, and min-LBD clauses (LBD 1 or 2) should be avoided, reflecting the typical three-tier clause management in modern solvers [15, 43]. We refrain from assigning min-LBD clauses because solvers commonly keep them *indefinitely*, which may lead to overcrowding if too many such clauses arrive from external sources [56]. We compare four strategies:

1. **Original LBDs**: The original LBD values are kept, used as a secondary clause sharing selection metric (after clause length), and incremented upon import [56].

LBD Strategy	#	+	−	PAR2
Deactivated	322	153	169	137.2
Original	322	153	169	137.3
Triangular	319	150	169	139.9
Uniform	318	148	170	141.2

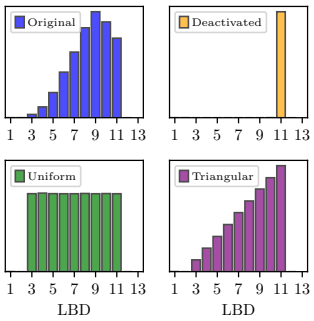


Figure 4 Left: Performance of different strategies for fabricated LBDs, showing number of solved instances (#), of which satisfiable (+) and unsatisfiable (−), and PAR2 scores. Right: Measured LBD distributions, observed directly at import in each thread. Here shown are statistics for clauses of size 11 during solving of a single instance. Each histogram represents ≈ 800 000 clauses.

2. **Deactivated LBDs**: When a clause is produced, its LBD value is set to its maximum value possible [24], the size of the clause. This functionally deactivates any logic related to LBDs in the distributed solver.
3. **Uniform LBDs**: When a clause is produced, its LBD value is set to its maximum. When a clause is imported and its size is $|c| \geq 3$, the importing thread randomly draws a new LBD $\in [3, \ldots, |c|]$ with uniform probability for each value. Initially setting LBDs to their maximum ensures that they play as little a role as possible during clause sharing.
4. **Triangular LBDs**: When a clause is produced, its LBD value is set to its maximum. When a clause is imported and its size is $|c| \geq 3$ the importing thread randomly draws a new LBD $\in [3, \ldots, |c|]$ whereas each LBD value s has a probability weight of $s - 2$. Visually, the probability density resembles a triangle; low LBDs receive a lower probablity to be sampled and high LBDs a higher probablity. This distribution produces fewer low-LBD clauses than the uniform distribution, which could affect the clause databases.

We ran each of these four strategies on 396 instances with a timeout of 300 s, distributed on 768 cores. We also deactivate all other sources of diversification (random seeds and sparse random variable phases [4]), as in ref. [56], such that any diversification introduced by just the LBD settings would be most observable. Results are shown in Fig. 4. Overall, all strategies performed similarly; best performance is achieved by using and incrementing original LBD scores or by removing them from the equation entirely. We were unable to observe any positive diversification effect due to randomized LBD scores in the triangular and uniform strategies. We suspect that these two strategies may still result in too many low-LBD clauses, at least at the tested scale and with MallobSat's default sharing configuration.

Put together, we found that removing LBD scores from clause sharing alltogether (and assigning the maximum LBD score to each incoming clause during import) preserves performance. Together with earlier findings [35, 56], this paints the clear picture that LBDs are not a useful metric for modern clause-sharing solvers. Going forward, we consider to either replace LBD with a different metric, as proposed by Iida et al. [35], or to remove them entirely from MallobSat's solver interfaces and sharing logic.

5 Streamlined Solver Design

Our profiling results (Fig. 1–2) have shown that distributed clause-sharing solving performs pre– and inprocessing in a highly suboptimal manner. We now suggest some first improvements to this situation, incorporating insights from the above discussion and measurements.

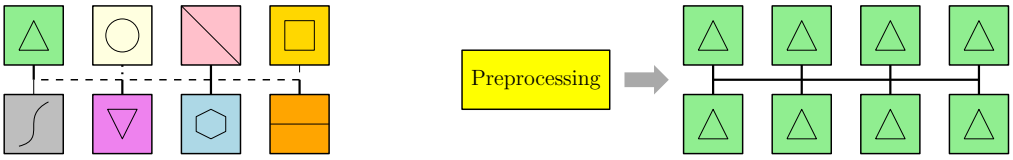


Figure 5 Left: Common distributed solver design, where a portfolio of solvers (colored squares) with diverse strategies (symbols in the squares) periodically exchange insights (connections between squares) with some "language barriers". Right: Streamlined design, where parallel solver threads are *uniform* CDCL searchers and operate on the results of preprocessing that is logically *separate*.

We can discern several general approaches for any single pre–/inprocessing technique:

1. **Keep** the technique as is, i.e., computed redundantly by all threads. This (non-)approach can be sensible if the technique consistently incurs little cost.
2. **Disable** the technique. This is reasonable especially if the technique is fairly expensive and is difficult or even infeasible to parallelize, or if it causes negative side effects for the parallelization. An example is variable elimination during inprocessing, which prevents the import of many clauses if not performed in a synchronized fashion [56].
3. **Sequentially** apply the technique with a single solver thread, then distribute the result. This can be a practical compromise for techniques that are challenging to parallelize, feasible to perform sequentially in little time, but still costly if performed by all threads.
4. **Parallelize** the technique. This can be an option if the technique is beneficial in a distributed setting and if it is expensive sequentially but feasible to parallelize. While there have been some works in this regard (see Section 2), so far next to no techniques have been explored in the context of state-of-the-art *distributed* SAT solving.

MallobSat's default configuration follows options 1 (for most threads) and 2 (for some threads, for the sake of diversification). In the long term, we are interested in finding scalable parallel algorithms for promising inprocessing tasks (option 4). In the scope of this mostly empirical work, we consider options 2 and 3 to obtain a more scalable distributed solver.

Specifically, we propose the following changes to MallobSat's setup: We drop all pre– and inprocessing, which, for the case of Kissat, amounts to disabling the options `preprocess`, `simplify`, `probe`, and `lucky`. Since many of MallobSat's *native* diversification options for individual solver backends [56] concern specific pre–/inprocessing techniques, we also completely disable this native diversification. This results in a "pure" clause-sharing solver with uniform worker threads (Fig. 5 right) and next to no characteristics of a *portfolio* (Fig. 5 left). We then configure a single thread to perform a single run of a number of crucial simplification techniques – similar to the preprocessing performed by PRS and PL-PRS-BVA-KISSAT (see Section 2) and also how the early portfolio solver Plingeling used to handle preprocessing [6].[2] Our approach differs in that all distributed resources are tasked *immediately* with search-only SAT solving as long as preprocessing is still ongoing. In terms of employed preprocessing techniques, we use the out-of-the-box preprocessing capabilities of Kissat, which includes not only bounded variable addition but also congruence closure and SAT sweeping, fast initial variable elimination, backbones, and lucky phases [12].

Once completed, the preprocessor reports the entire preprocessed formula. Since our execution environment Mallob natively supports concurrent execution and flexible load balancing of SAT solving tasks [47], we can simply introduce the preprocessed formula as

[2] In a sense, Plingeling's "preprocess-then-fork" model has gotten buried due to HordeSat's [4] slim and modular solver interface, prohibiting this mode of usage when integrating Plingeling's solver diversification, and thus never found its way into HordeSat and later MallobSat.

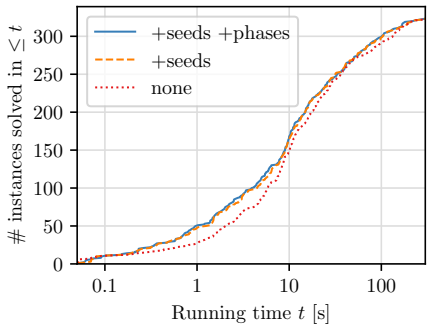
an additional task that runs next to the original solving task. This raises the question of how to distribute the available computational resources among these two tasks. Assume that the original task j_o has been running for time t_o whereas the preprocessed task j_p has been running for time t_p. Given equal time and resources, we assume that the preprocessed task is likely to be easier to solve. Therefore, as $\frac{t_p}{t_o}$ increases and eventually approaches 1, the relative worth of j_p over j_o increases. Following up on this intuition, we suggest to gradually shift computational resources from j_o to j_p until only a single process for j_o is remaining. Eventually, j_o is interrupted completely. A result is reported if the preprocessor manages to solve the formula directly or when either j_o or j_p report a result. A satisfying assignment reported by j_p must be converted to a satisfying assignment to the original formula, using the preprocessor's data structures for solution reconstruction.

6 Evaluation

In the following, we provide an evaluation of our improvements.

First of all, we examine the impact of different sources of solver thread diversification. Our search-only setup with prior single-core preprocessing only features a single solver backend (Kissat) and no longer features a hand-crafted sequence of solver configurations that are being cycled over (cf. [55]). This drastically cuts diversity and leaves what can be characterized rather as a uniform array of CDCL searchers than a classical "portfolio" (Fig. 5). In our study on LBDs (Section 4), we found that the diversification introduced by deviating LBD distributions across solver threads does not influence performance even when all other sources of diversification are disabled. We thus follow the "Deactivated" LBD strategy in our new setup. Two other computationally cheap, basic, and uniform diversification techniques we consider are setting different random seeds and variable phases per solver [4, 56]. We thus ran our baseline without any diversification, with seeds only, and with seeds and phases.

Results are reported in Fig. 6. While MallobSat still performs well *without* any explicit diversification – an effect examined in detail in ref. [56] and attributed to deviating clause imports across solver threads – we see that different random seeds are beneficial for short running times, especially below 10 s. Random seeds essentially provide a warm-start for deviating search behavior rather than having to wait for diversity induced by the first few clause imports. This contrasts randomized LBDs, which can only have any impact after the first sharing(s). Randomized phases only have a minimal effect, but still lead to a net



Diversification	#	+	−	PAR2
+seeds +phases	324	155	169	131.9
+seeds	323	154	169	133.2
none	322	153	169	137.2

Figure 6 Left: Effect of basic diversification techniques on performance in our 768-core search-only setup with single-core preprocessing. Right: Corresponding number of solved instances (#), of which satisfiable (+) and unsatisfiable (−) instances, and PAR2 scores.

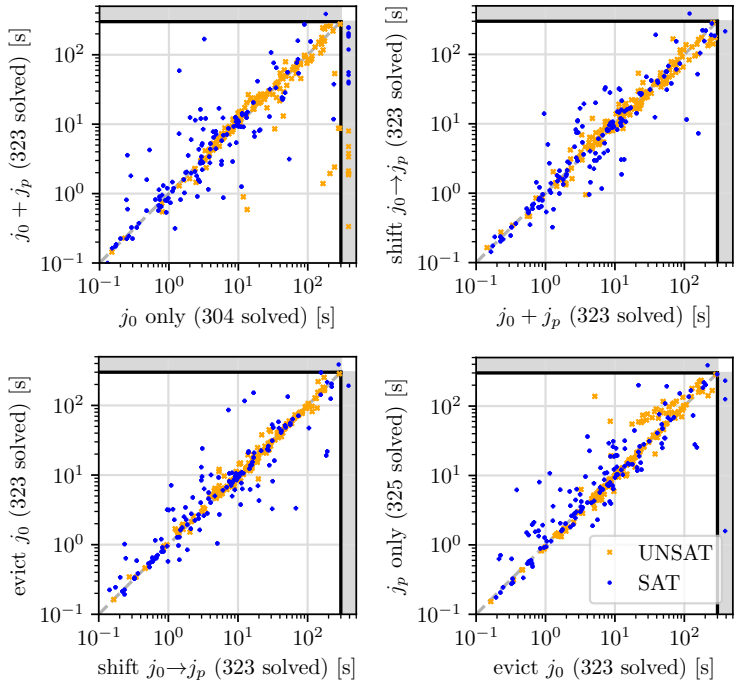


Figure 7 768-core performance comparisons. Top left: Only running a search-only task j_0 vs. adding a preprocessed task j_p via sequential preprocessing ("$j_0 + j_p$"). Top right: "$j_0 + j_p$" vs. shifting resources gradually from j_0 to j_p ("shift $j_0 \to j_p$"). Bottom left: "shift $j_0 \to j_p$" vs. evicting j_0 immediately once j_p enters. Bottom right: evicting j_0 vs. never launching j_0 in the first place.

positive and incur no opportunity cost. We also explored some additional diversification, such as randomizing VSIDS score decays or clause database reduction ratios, without observing significant improvements; according results are given in the appendix (Fig. 12–13).

Next, we evaluate different strategies surrounding our single-core preprocessing. An overview is provided in Fig. 7. First, adding sequential preprocessing to an otherwise search-only run and launching a task j_p that operates on the preprocessed formula is highly beneficial to performance. The run with preprocessing solves 19 additional instances and improves performance for several unsatisfiable instances (which incur much less running time variance than satisfiable instances). The preprocessor solved 18 instances directly and mostly finishes in few seconds. Secondly, rather than allotting equal resources to j_0 and j_p (as in the prior run), we gradually shift resources from j_0 to j_p, over a time span corresponding to twice the preprocessor's running time, and then remove j_0 entirely. For long running times, this approach is preferable for unsatisfiable instances since it eventually commits all available resources to a single task. Next, we evict j_0 immediately as soon as j_p enters, which results in mildly worse performance (2.4% mean slowdown), indicating that j_o occasionally solves an instance after the arrival of j_p. Lastly, omitting j_o entirely, thus deferring distributed solving until a preprocessing result is available, clearly performs worse (14.7% slowdown).

We continue our experiments with the "shift" strategy since it assumedly results in the best scalability in the long term (i.e., eventually allotting all resources to a single task) while not immediately discarding the progress made by the plain searcher task.

We now assess the scaling of our streamlined setup, using the 400 remaining, previously unused instances from the 2023–2024 benchmark sets. We use 1–64 nodes (48–3072 cores), doubling the number of nodes at each step. As points of reference, we use two state-of-the-art

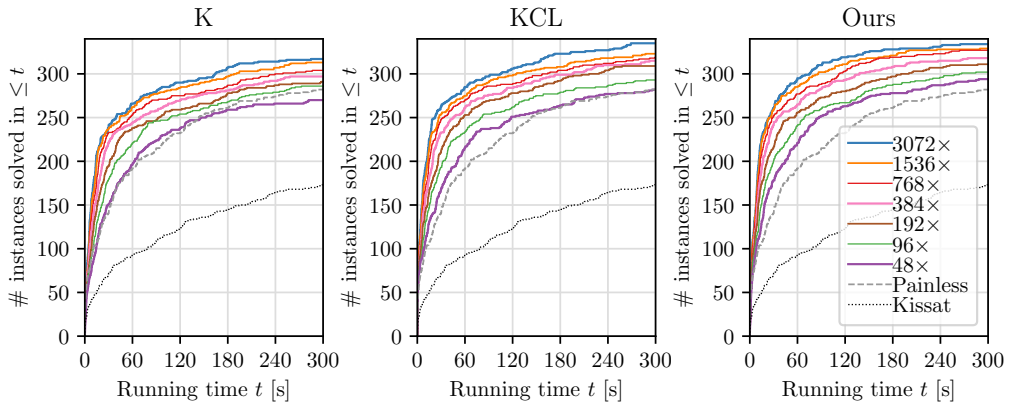


Figure 8 Strong scaling of MallobSat's baseline K and KCL configurations (left, center) and our new setup (right) in terms of absolute performance from 1 to 64 nodes (48–3072 cores). We also show running times of sequential Kissat and 48-core PL-PRS-BVA-KISSAT ("Painless").

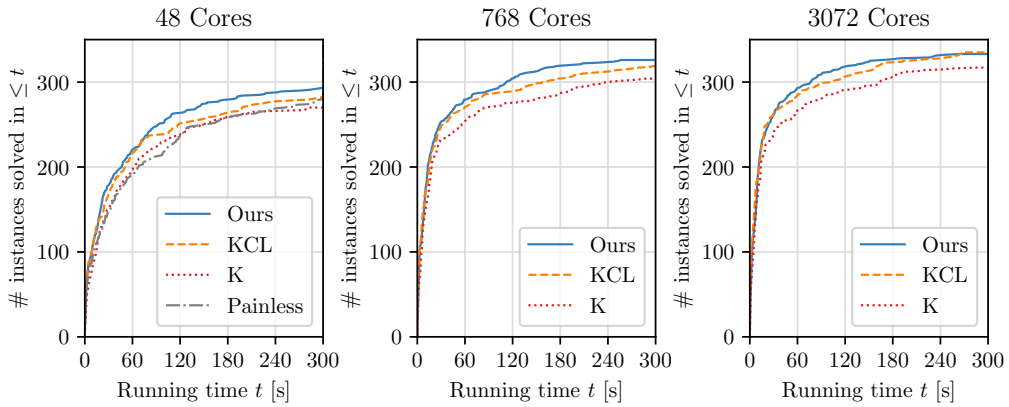


Figure 9 Direct comparison of the three tested MallobSat configurations at 1, 8, and 64 nodes. At one node (48 cores), the run of PL-PRS-BVA-KISSAT ("Painless") is shown as well.

configurations of MallobSat: Only running Kissat ("K"), and running Kissat, CaDiCaL, and Lingeling alternatingly ("KCL"). The first configuration is the (prior) single-solver configuration with assumedly best general-purpose performance whereas KCL corresponds to the version of MallobSat submitted to the 2024 SAT Competition Cloud Track (which it won). Notably, the inclusion of Lingeling [7, 9] is known to noticeably boost performance on the 2023–2024 benchmark sets due to some peculiar instances which can be solved immediately by Lingeling's cardinality constraint reasoning [10] (see also [52]). This circumstance provides KCL with a distinct advantage over Kissat-only configurations in terms of solving performance metrics. Due to native diversification, roughly 13% of solver threads in the K and KCL setups run without or with significantly reduced pre– and inprocessing.

Fig. 8–9 show the scaling behavior of the three considered configurations, with precise statistics given in Tab. 1. In terms of baselines, we included the performance of sequential Kissat (winner of the SAT Competition 2024 Main track) and of PL-PRS-BVA-KISSAT [49] ("painless-2" in SAT Competition 2024, winner of the Parallel track) executed at 48 cores. All configurations of MallobSat show consistent performance improvements up to the largest scale considered (3072 cores). The out-of-the-box Kissat-only configuration results in the worst performance. Notably, it features neither cardinality constraint reasoning (as in Lingeling)

Table 1 Performance of three MallobSat configurations at 48–3072 cores, showing the number of solved instances (#), thereof satisfiable (+) and unsatisfiable (−) instances, Penalized Average Runtime which rates each timeout as solved in twice the time limit, i.e., 600 s (PAR-2), geometric mean speedup vs. sequential Kissat over all commonly solved instances (S_g), and corresponding total speedup, i.e., summing up the sequential solver's running times and then dividing by the sum of the parallel solver's running times (S_t). At each scale, the best values of each metric are highlighted.

	K						**KCL**						**Ours**					
cores	#	+	−	PAR	S_g	S_t	#	+	−	PAR	S_g	S_t	#	+	−	PAR	S_g	S_t
48	270	132	138	226.4	7.8	17.9	282	129	153	208.6	10.8	18.2	294	135	159	191.2	8.4	20.0
96	286	140	146	202.2	11.1	29.3	293	132	161	188.7	14.3	24.4	302	137	165	177.0	11.5	30.4
192	292	143	149	191.3	13.4	36.9	309	140	169	165.3	17.7	35.7	311	142	169	161.0	14.7	40.2
384	297	144	153	180.7	17.1	52.7	315	140	175	155.1	21.6	50.7	318	145	173	147.6	17.2	63.5
768	305	147	158	169.4	19.0	57.3	319	143	176	146.9	26.0	61.4	327	149	178	133.6	21.7	82.7
1536	314	151	163	155.8	21.1	67.0	323	144	179	139.0	30.2	72.1	329	149	180	127.4	24.6	98.3
3072	318	152	166	148.5	23.3	75.6	335	154	181	122.7	34.4	88.3	334	151	183	119.4	26.7	111.5

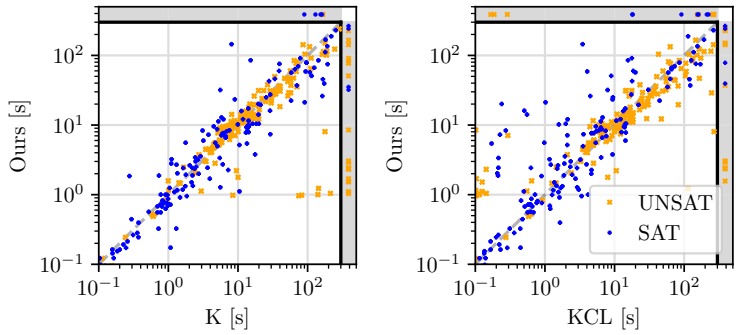


Figure 10 Direct performance comparison of our approach vs. the two baseline configurations K (left) and KCL (right) at 3072 cores.

nor any reasoning beyond general resolution (as in bounded variable addition, which is part of our preprocessing). The diverse Kissat–CaDiCaL–Lingeling portfolio improves performance substantially (25.8 PAR-2 score points at 3072 cores). Our approach, in turn, outperforms all other approaches, achieving the lowest PAR-2 scores at all scales.

The performance margin to KCL is the smallest at 3072 cores, and mean speedups are consistently the highest for KCL. This raises the question whether we are observing a distinct scalability limit of our new approach. A direct comparison of the runs at 3072 cores (Fig. 10) suggests that this is not the case: Compared to both K and KCL, our approach clearly tends to perform better as running times increase. A distinct advantage of the KCL setup is that it solves a number of unsatisfiable instances in the sub-second range which are challenging or even infeasible to solve with Kissat only. At 3072 cores, we identified 17 such instances; all of them are in fact solved by a Lingeling solver thread. The concerned instances are eleven register allocation graph coloring problems [28], four Tseitin formulas [19], and two Pidgeon hole problems [8]. Our approach compensates for this disadvantage with generally better scalability, solving other instances which KCL fails to solve; however, at the largest scale, our approach runs out of such instances while KCL manages to close the gap.

A second advantage of KCL is the dedicated Stochastic Local Search solver YalSAT [9] in Lingeling's portfolio. KCL outperforms our new setup by a factor of ≥ 5 for 21 satisfiable instances; 18 of them were solved by a Lingeling instance, and 14 of them specifically by YalSAT. This includes seven instances surrounding Folkman graphs [32] and six set covering problems [63]. On the flipside, our new approach performs remarkably well on many

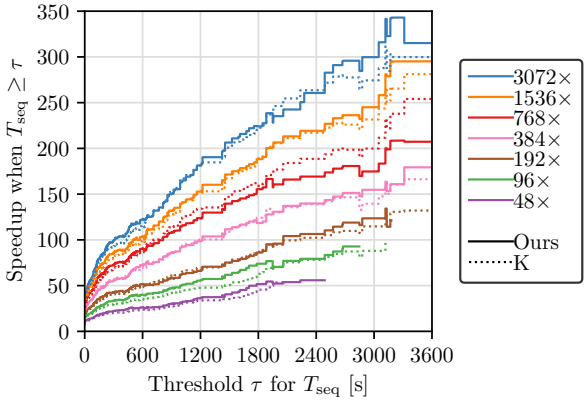


Figure 11 *Weak Scaling* of MallobSat. The color denotes the scale of solving (1–64 nodes); continuous lines denote our approach whereas dotted lines denote the "K" baseline configuration. Point (x, y) on a curve indicates that the corresponding run achieved a geometric mean speedup of y on the instances which took sequential Kissat at least time x to solve. A curve ends prematurely if this amounts to less than 20 considered instances.

benchmarks adjacent to real-world applications (see also appendix Tab. 4), such as bitvectors (4 instances, 67% mean speedup over KCL), verifying floating-point commutativity (6×, 66%), hashtable safety (7×, 48%), software (5×, 38%) and hardware verification (7×, 38%), school timetabling (9×, 26%), and hash function properties (18×, 23%). We attribute this success to our streamlined solver design (cf. Fig. 2); while KCL is missing BVA, this technique is highly unlikely to cause such scaling improvements on such kinds of instances [29].

The overall, direct geometric mean speedup of our configuration over KCL at 3072 cores is 0.815 (i.e., a slowdown). Removing the 17 instances solved immediately by Lingeling results in a speedup of 0.907 (still a slowdown). When we additionally remove the 14 instances where YalSAT found a solution $\geq 5\times$ faster than our approach did, the speedup is 1.036.

Fig. 11 provides insights into MallobSat's *weak scaling* for the "K" baseline and our new setup (cf. [56]). Intuitively, the further right a curve goes, the more difficult are the instances that we consider for computing the mean speedup. We measure "difficulty" in terms of the sequential baseline's running time. For example, our approach at 3072 cores reaches a speedup of 315.1 for instances that took sequential Kissat at least 1 h to solve, which corresponds to a resource efficiency of $315.1/3072 = 10.3\%$. Under the same circumstances, the baseline configuration "K" achieves a speedup of 299.8 (resource efficiency 9.8%). The KCL portfolio (not shown[3]) results in the best weak scaling, with a speedup of 451.5 and an according resource efficiency of 14.7%. This discrepancy can be explained by the fact that KCL performs noticeably better than our new approach in the first 10–20 s of solving, for reasons discussed above (i.e., due to Lingeling and YalSAT). The corresponding short running times cause relatively high speedups. In terms of *total* speedups (Table 1), which emphasize longer running times, our approach consistently achieves the best results.

[3] The weak scaling as we compute it heavily depends on the sets of instances a certain approach is able to solve. The weak scaling curves of the KCL configuration are therefore difficult to compare to the other configurations due to the different kinds of instances solved (mostly by Lingeling/YalSAT). We do provide a weak scaling plot of the KCL configuration in the appendix (Fig. 14).

7 Discussion

In the following, we provide a brief discussion of the obtained insights and results.

Our work results in a slimmed down and yet competitive solver system. Let us attempt to provide a coarse quantitative estimate of the achieved reduction by considering the involved lines of code. We counted roughly 25k effective lines of code (ELOC; disregarding blank lines and comments) in Lingeling's main source file `lglib.c`, 30k ELOC in CaDiCaL's source files (`*.cpp`) and also 30k ELOC in Kissat's source files (`*.c`). We estimate that roughly 20k ELOC of the Mallob system are in use. Our changes to Mallob amount to few hundred lines of code. Going by these measures, our setup reduces the code to be considered from 105k ELOC to 50k ELOC. Moreover, the code executed by the parallel solver threads decreases by roughly 13k ELOC (when excluding Kissat's source files related to pre–/inprocessing). While these measures are not fully reliable (as each solver backend already features some unused code), we believe it is safe to assume that our changes to MallobSat cut the lines of executed code roughly in half and drastically reduce the solver program executed in parallel.

Comparing our setup to MallobSat's KCL portfolio, we noticed some remaining shortcomings of our somewhat radical approach, mostly due to foregoing dedicated Stochastic Local Search (SLS) threads and due to missing special techniques for certain hard inputs. This motivates us to (a) reintegrate SLS in our approach, which presents only a minor increase in complexity (especially in terms of information flow), and (b) to integrate advanced preprocessing techniques (as in ref. [61]) that are required to conquer the kinds of instances which our current setup still struggles to solve. Specifically, our new approach to preprocessing can be adapted well to preprocessors that exploit advanced proof systems, such as extended resolution (which our preprocessing already features via bounded variable addition) [29], symmetry breaking (e.g., BreakID [18, 23]), or propagation redundancy (e.g., PReLearn [46]). Following our evaluation, we have made first simple steps in this direction by adding Lingeling's preprocessing and 5% YalSAT threads to our setup; we observed significantly improved performance at a shared-memory level (see appendix Fig. 15).

On a related note, transferring our findings to a *CaDiCaL-only setup* within MallobSat will allow to exploit MallobSat's proof production and checking capabilities (which, as of now, rely on CaDiCaL's LRUP proof output [45]). While it appears challenging to generalize distributed proof production [42] and real-time checking [52] to proof systems beyond resolution, our pragmatic approach to preprocessing allows to "outsource" advanced reasoning to a strict, sequential prefix of the distributed solver's reasoning, which could greatly facilitate combining scalable proof technology with modern proof systems.

Lastly, and perhaps most significantly, the presence of a *competitive* "pure" clause-sharing solver may have notable consequences for future research in parallel and distributed SAT solving. Rather than assuming a gathering of diverse and opaque reasoners who periodically exchange insights [56], we may now rather accurately consider the central building block of modern parallel SAT solving to be a *uniform parallel CDCL* procedure, parallelized by clause sharing. This more precise and transparent algorithmic model lends itself for establishing new theory, heuristics, and algorithms surrounding parallel SAT solving.

8 Conclusion

In this work, we investigated modern distributed clause-sharing SAT solver design. In particular, we empirically confirm earlier suspicions [20] that such distributed solvers are being held back by performing non-scalable tasks fully redundantly, such as pre– and inprocessing techniques. We also show conclusively that LBD values have no measurable significance

for MallobSat's distributed clause sharing. At the same time, we show that sequential preprocessing employed next to search-only solving can still greatly benefit distributed solving. Our practical result is a clean distributed clause-sharing solver whose solver threads are "pure" CDCL solving engines with very lightweight and uniform diversification. This system is able to achieve competitive performance, even compared to a mixed portfolio consisting of many different search strategies and advanced reasoning techniques.

In terms of future work, we intend to explore scalable algorithms for selected pre– and/or inprocessing tasks, such as vivification, variable elimination, or SAT sweeping. Distributed computing may allow for much more powerful simplification procedures while the parallel search logic could remain separate, uniform, and fully scalable. Orthogonally, we intend to retrofit our new setup from this work to MallobSat's CaDiCaL backend, which (as of yet) is MallobSat's only modern engine for incremental SAT solving [53] and for proofs of unsatisfiability [42, 52]. Lastly, it may be promising to allow sharing (certain) clauses between an original task j_o and a preprocessed task j_p, since this may allow to transfer made progress from j_o to j_p following preprocessing.

References

1 Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel SAT solving. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 200–213. Springer, 2012. doi:10.1007/978-3-642-31612-8_16.

2 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 399–404, 2009. URL: http://ijcai.org/Proceedings/09/Papers/074.pdf.

3 Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel SAT solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 197–205. Springer, 2014. doi:10.1007/978-3-319-09284-3_15.

4 Tomáš Balyo, Peter Sanders, and Carsten Sinz. Hordesat: A massively parallel portfolio SAT solver. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 156–172. Springer, 2015. doi:10.1007/978-3-319-24318-4_12.

5 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.

6 Armin Biere. Lingeling and friends entering the SAT challenge 2012. In *Proc. SAT Challenge*, pages 33–34, 2012.

7 Armin Biere. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In *SAT Competition 2013: Solver and Benchmark Descriptions*, page 1, 2013.

8 Armin Biere. Two pidgeons per hole problem. In *SAT Competition 2013: Solver and Benchmark Descriptions*, page 103, 2013.

9 Armin Biere. Yet another local search solver and Lingeling and friends entering the SAT competition 2014. In *SAT Competition 2014: Solver and Benchmark Descriptions*, page 65, 2014.

10 Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. Detecting cardinality constraints in CNF. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 285–301. Springer, 2014. doi:10.1007/978-3-319-09284-3_22.

11 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In *International Conference on Computer Aided Verification*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

12 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT competition 2024. *International SAT Competition 2024: Solver and Benchmark Descriptions*, page 8, 2024.

13 Armin Biere, Katalin Fazekas, Mathias Fleury, and Nils Froleyks. Clausal congruence closure. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 6:1–6:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SAT.2024.6.

14 Armin Biere, Katalin Fazekas, Mathias Fleury, and Nils Froleyks. Clausal equivalence sweeping. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 236–241. TU Wien Academic Press, 2024. doi:10.34727/2024/isbn.978-3-85448-065-5_29.

15 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *SAT Competition 2020: Solver and Benchmark Descriptions*, page 50, 2020.

16 Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2 edition, 2021. doi:10.3233/faia336.

17 Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In *Handbook of Satisfiability*, pages 391–435. IOS Press, 2021. doi:10.3233/FAIA200992.

18 Bart Bogaerts, Jakob Nordström, Andy Oertel, and Cagrı Uluç Yıldırımoglu. BreakID-kissat in SAT competition 2023 (system description). *SAT Competition 2023: Benchmark, Solver and Proof Checker Descriptions*, page 25, 2023.

19 Bart Bogaerts, Jakob Nordström, Andy Oertel, and Cagrı Uluç Yıldırımoglu. Crafted benchmark formulas requiring symmetry breaking and/or parity reasoning. In *SAT Competition 2023: Benchmark, Solver and Proof Checker Descriptions*, page 67, 2023.

20 Jannick Borowitz, Dominik Schreiber, and Peter Sanders. An empirical study on learned clause overlaps in distributed SAT solving. In *Pragmatics of SAT (PoS)*, 2024. URL: https://satres.kikit.kit.edu/papers/2024-pos-empirical.pdf.

21 Zhihan Chen, Xindi Zhang, Yuhang Qian, and Shaowei Cai. PRS: A new parallel/distributed framework for SAT. *SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, page 39, 2023.

22 Byron Cook. Automated reasoning's scientific frontiers. https://www.amazon.science/blog/automated-reasonings-scientific-frontiers, 2021. Amazon Science.

23 Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 104–122. Springer, 2016. doi:10.1007/978-3-319-40970-2_8.

24 Thorsten Ehlers and Dirk Nowotka. Tuning parallel SAT solvers. In *Pragmatics of SAT*, volume 59, pages 127–143, 2019. doi:10.29007/z3g2.

25 Thorsten Ehlers, Dirk Nowotka, and Philipp Sieweck. Communication in massively-parallel SAT solving. In *Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, pages 709–716. IEEE, 2014. doi:10.1109/ictai.2014.111.

26 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.

27 Kilian Gebhardt and Norbert Manthey. Parallel variable elimination on CNF formulas. In *Annual Conference on Artificial Intelligence*, pages 61–73. Springer, 2013. doi:10.1007/978-3-642-40942-4_6.

28 Andrew Haberlandt and Harrison Green. Python function register allocation benchmarks. In *SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, page 56, 2023.

29 Andrew Haberlandt, Harrison Green, and Marijn J. H. Heule. Effective auxiliary variables via structured reencoding. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 11:1–11:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SAT.2023.11.

30 Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *JSAT*, 6(4):245–262, 2010. doi:10.3233/sat190070.

31 Maximilian Heisinger, Mathias Fleury, and Armin Biere. Distributed cube and conquer with paracooba. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 114–122. Springer, 2020. doi:10.1007/978-3-030-51825-7_9.

32 Marijn Heule. Avoiding monochromatic triangles and L-shapes. In *SAT Competition 2024: Solver, Benchmark and Proof Checker Descriptions*, pages 41–42, 2024.

33 Marijn J. H. Heule, Oliver Kullmann, and Victor Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 228–245. Springer, 2016. doi:10.1007/978-3-319-40970-2_15.

34 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.

35 Yoichiro Iida, Tomohiro Sonobe, and Mary Inaba. Parallel Clause Sharing Strategy Based on Graph Structure of SAT Problem. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 17:1–17:18, 2024. doi:10.4230/LIPIcs.SAT.2024.17.

36 Yoichiro Iida, Tomohiro Sonobe, and Mary Inaba. Wance: Learnt clause evaluation method for SAT solver using graph structure. In *International Conference on Learning and Intelligent Optimization*, pages 190–204. Springer, 2024. doi:10.1007/978-3-031-75623-8_15.

37 Markus Iser and Christoph Jabs. Global Benchmark Database. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 18:1–18:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SAT.2024.18.

38 Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. PaInleSS: a framework for parallel SAT solving. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 233–250. Springer, 2017. doi:10.1007/978-3-319-66263-3_15.

39 Mao Luo, Chu-Min Li, Fan Xiao, Felip Manya, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In *Proc. IJCAI*, pages 703–711, 2017. doi:10.24963/ijcai.2017/98.

40 Norbert Manthey, Marijn JH Heule, and Armin Biere. Automated reencoding of Boolean formulas. In *Haifa Verification Conference*, pages 102–117. Springer, 2012. doi:10.1007/978-3-642-39611-3_14.

41 João Marques-Silva, Inês Lynce, and Sharad Malik. CDCL SAT solving. In *Handbook of Satisfiability*, pages 133–182. IOS Press, 2021. doi:10.3233/FAIA200987.

42 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.

43 Chanseok Oh. Between SAT and UNSAT: the fundamental difference in CDCL SAT. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 307–323. Springer, 2015. doi:10.1007/978-3-319-24318-4_23.

44 Muhammad Osama, Anton Wijs, and Armin Biere. Certified SAT solving with GPU accelerated inprocessing. *Formal Methods in System Design*, 62:79–118, 2023. doi:10.1007/s10703-023-00432-z.

45 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.

46 Joseph E. Reeves and Randal E. Bryant. Preprocessors PReLearn and ReEncode entering the SAT Competition 2023. In *SAT Competition 2023: Benchmark, Solver and Proof Checker Descriptions*, page 23, 2023.

47 Peter Sanders and Dominik Schreiber. Decentralized online scheduling of malleable NP-hard jobs. In *Euro-Par 2022: Parallel Processing*, pages 119–135. Springer, 2022. doi:10.1007/978-3-031-12597-3_8.

48 Mazigh Saoudi, Souheib Baarir, Julien Sopena, Thibault Lejemble, and Sabrine Saouli. Painless in SAT Competition 2024 with PL-PRS-BVA-KISSAT and PL-PRS-MPI. *SAT Competition 2024: Benchmark, Solver and Proof Checker Descriptions*, page 24, 2024.

49 Mazigh Saoudi, Thibault Lejemble, Souheib Baarir, and Julien Sopena. PL-PRS-BVA-KISSAT in SAT Competition 2024. In *Pragmatics of SAT*, 2024. URL: `https://dl.lre.epita.fr/papers/saoudi.24.pos.pdf`.

50 Dominik Schreiber. Mallob{32,64,1600} in the SAT competition 2023. In *SAT Competition 2023: Benchmark, Solver and Proof Checker Descriptions*, pages 46–47, 2023.

51 Dominik Schreiber. MallobSat and MallobSat-ImpCheck in the SAT Competition 2024. In *SAT Competition 2024: Solver, Benchmark and Proof Checker Descriptions*, pages 21–22, 2024.

52 Dominik Schreiber. Trusted scalable SAT solving with on-the-fly LRAT checking. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 25:1–25:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. `doi:10.4230/LIPIcs.SAT.2024.25`.

53 Dominik Schreiber. Distributed incremental SAT solving with Mallob: Report and case study with hierarchical planning, 2025. `arXiv:2505.18836`.

54 Dominik Schreiber, Niccolò Rigi-Luperti, and Armin Biere. Supplementary material for SAT'25 publication "Streamlining Distributed SAT Solver Design". Software, swhId: `swh:1:dir:bdecc26fc577de49d3fe853d07e4ba59e3cbe2f8` (visited on 2025-07-25). URL: `https://github.com/nrilu/SAT_2025_supplement_63`, `doi:10.4230/artifacts.24204`.

55 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`.

56 Dominik Schreiber and Peter Sanders. MallobSat: Scalable SAT solving by clause sharing. *Journal of Artificial Intelligence Research*, 80:1437–1495, 2024. `doi:10.1613/jair.1.15827`.

57 Sven Schulz and Wolfgang Blochinger. Cooperate and compete! A hybrid solving strategy for task-parallel SAT solving on peer-to-peer desktop grids. In *Int. Conf. HPC & Simulation*, pages 314–323. IEEE, 2010. `doi:10.1109/HPCS.2010.5547121`.

58 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`.

59 Vincent Vallade, Julien Sopena, and Souheib Baarir. Enhancing state-of-the-art parallel SAT solvers through optimized sharing policies. In *Pragmatics of SAT*, pages 35–45, 2023. URL: `https://ceur-ws.org/Vol-3545/paper3.pdf`.

60 Siert Wieringa and Keijo Heljanko. Concurrent clause strengthening. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 116–132. Springer, 2013. `doi:10.1007/978-3-642-39071-5_10`.

61 Xindi Zhang, Zhihan Chen, and Shaowei Cai. ParKissat: Random shuffle based and preprocessing extended parallel solvers with clause sharing. In *SAT Competition 2022: Benchmark and Solver Descriptions*, page 51, 2022.

62 Xindi Zhang, Zhihan Chen, and Shaowei Cai. PRS: A new parallel/distributed framework for SAT. In *SAT Competition 2023: Benchmark, Solver and Proof Checker Descriptions*, pages 39–40, 2023.

63 Jiongzhi Zheng, Mingming Jin, Kun He, Zhuo Chen, and Jinghui Xue. SAT instances based on the set covering problem with conflict. In *SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, page 80, 2023.

A Parameter Studies

We now report two additional parameter studies. They were done with the setting denoted "Ours", with one difference: original LBDs were still used instead of the Deactivated-LBDs setting. Each run consisted of 396 instances with 300 s timeout on 8 nodes (384 cores).

The first study explores clause database reduction. Kissat offers parameters `reduce-low` (default 500) and `reduce-high` (default 900), which control the aggressiveness of database reductions. The default parameters correspond to reductions by 50% early in the run and by up to 90% later in the run. We tested four parameter settings, described briefly.

Reduce setting	#	+	−	PAR2
Default	323	153	170	137.9
Point	323	154	169	139.2
Gaussian	321	151	170	140.8
Range	318	149	169	144.2

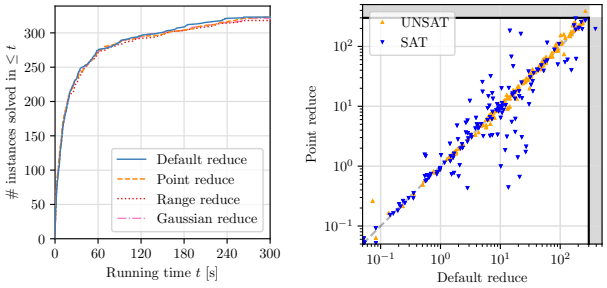


Figure 12 Effects of diversifying Kissat's `reduce-low` and `reduce-high` parameters.

Default reduce: Default reduce parameters. **Point reduce**: Value $r \in [0, ..., 1000]$ is uniformly sampled per solver and both `reduce-low` and `reduce-high` are set to it. This creates some extreme solvers that keep all clauses forever ($r = 0$) and others that delete every clause almost immediately ($r = 1000$). **Range reduce**: A value $r \in [-200, 1200]$ is uniformly sampled per thread; then we set `reduce-low=max(0,r-200)` and `reduce-high=min(1000,r+200)`. Intuitively this slides the default interval [500,900] randomly to higher or lower values and leaves per solver the flexiblity of shifting from low to high reductions. **Gaussian reduce**: A value r is sampled from a Gaussian distribution with mean 700 and standard deviation 150, then r is clipped to be within [300,980] and both `reduce-low` and `reduce-high` are set to it. This sampling specifically avoids the extremes from the other two settings.

The results of the four reduce settings are shown in Fig. 12. Default reduce performs overall best, whereas Point reduce performs strongly on some SAT instances, which might be due to some aggressive solvers being allowed to eliminate almost all learned clauses.

The second study focuses on the decay of (E)VSIDS scores controlled by the `decay` parameter. Its default value is 50 (per mille), corresponding to an update of variables activity scores to 95% of their former value. Higher `decay` results in more aggressive updates.

Decay setting	#	+	−	PAR2
Uniform [1,50]	324	154	170	135.5
Default [50]	323	153	170	137.9
Uniform [1,200]	322	152	170	140.1
Uniform [50,200]	321	151	170	141.1

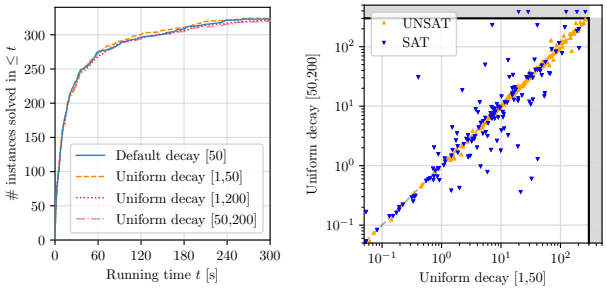


Figure 13 Effects of diversifying Kissat's `decay` parameter.

Kissat accepts values in the range of [1,...,200]. We explore this full range by testing three settings: **Uniform[1,50]**, **Uniform[1,200]** and **Uniform[50,200]**. In each setting every solver thread samples its `decay` value uniformly from the given interval. The third setting is thus much more eager than the default, while the first is more conservative.

The results of the different decay settings are shown in Fig. 13. Regarding PAR2 scores, the conservative updating with `decay` at or below 50 performs better than the more aggressive choices. However, similar to the database reductions, the more eager approaches perform better on some SAT instances, observable in the direct comparison plot.

B Supplementary Data

Table 2 Percentages of total ("CPU") time spent in different procedures of Kissat's SAT solving, for sequential Kissat, 768-core default setup, and 768-core search-only setup. Note that some categories subsume each another (e.g., focused and stable are disjoint sub-categories of search).

Procedure	Kissat	768×d	768×s-o
backbone	0.12	0.18	—
congruence	0.46	1.85	—
eliminate	1.01	1.33	—
extend	0.00	0.00	—
factor	0.55	—	—
fastel	0.30	2.96	—
focused	43.39	44.65	49.97
lucky	0.08	1.34	1.59
parse	0.15	—	—
preprocess	0.83	4.77	—
probe	11.15	13.52	—
reduce	1.37	2.68	3.24
search	86.32	79.15	95.41
simplify	12.61	13.86	1.77
stable	42.93	34.50	45.44
substitute	0.66	1.39	—
subsume	0.39	0.36	—
sweep	2.18	1.54	—
transitive	0.17	0.20	—
vivify	7.03	8.35	—
walking	0.96	0.65	1.77

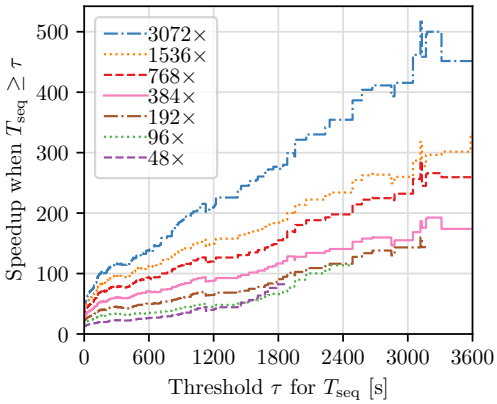


Figure 14 Weak Scaling of KCL configuration (as in Fig. 11).

Table 3 Geom. mean speedup of "search-only" configuration over default Kissat-only configuration, at 768 cores, for each GBD benchmark family and result (SAT/UNSAT) group with data on ≥ 5 instances ("#"). "Avg. time" denotes the default configuration's according average running time.

Family	#	Avg. time	Speedup
miter-unsat	14	34.86	0.30
profitable-robust-production-sat	5	7.19	0.35
heule-nol-sat	7	18.50	0.43
social-golfer-sat	6	8.23	0.52
grs-fp-comm-unsat	8	75.60	0.62
scheduling-unsat	9	26.66	0.80
software-verification-unsat	10	56.29	0.97
cryptography-simon-sat	8	0.13	0.98
random-circuits-sat	5	24.89	1.01
hamiltonian-unsat	11	7.11	1.05
cryptography-ascon-unsat	6	8.58	1.05
argumentation-unsat	18	7.45	1.06
satcoin-unsat	5	16.36	1.28
brent-equations-sat	7	0.96	1.29
hamiltonian-sat	12	2.14	1.35
maxsat-optimum-sat	5	5.78	1.36
scheduling-sat	9	40.00	1.59
heule-folkman-sat	5	81.82	1.82
school-timetabling-sat	8	21.01	2.00
cryptography-sat	6	12.77	2.05
set-covering-sat	14	11.07	2.09
mutilated-chessboard-unsat	6	31.48	2.27

Table 4 Geometric mean speedup of our new setup over KCL, at 3072 cores, split as in Table 3. "Avg. time" denotes the average running time of KCL on the respective instances.

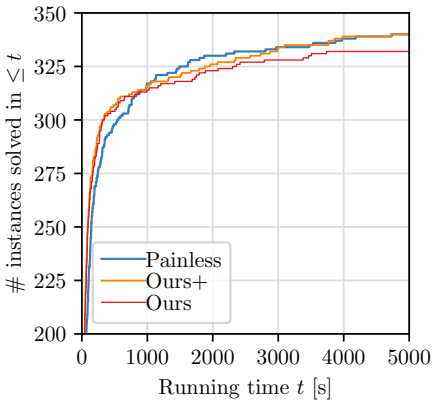
Family	#	Avg. time	Speedup
set-covering-sat	6	0.21	0.04
heule-folkman-sat	6	5.82	0.08
register-allocation-unsat	11	0.12	0.10
random-circuits-sat	10	23.27	0.60
rbsat-sat	5	13.93	0.68
maxsat-optimum-unsat	5	19.11	0.70
hamiltonian-unsat	9	5.19	0.78
brent-equations-sat	9	0.63	0.81
argumentation-unsat	16	12.00	0.84
profitable-robust-production-sat	8	70.91	0.84
cryptography-ascon-unsat	10	8.12	0.88
quantum-kochen-specker-unsat	6	9.92	0.90
hamiltonian-sat	8	1.46	1.03
scheduling-unsat	6	28.64	1.04
scheduling-sat	17	24.12	1.08
satcoin-unsat	10	13.73	1.16
cryptography-sat	8	24.14	1.18
school-timetabling-sat	9	11.98	1.25
miter-sat	9	87.61	1.25
miter-unsat	23	29.96	1.26
coloring-unsat	5	24.48	1.26
software-verification-unsat	5	34.61	1.38
hashtable-safety-unsat	7	100.41	1.48
cryptography-ascon-sat	8	3.00	1.57
grs-fp-comm-unsat	6	65.93	1.63

Table 5 Performance as in Tab. 1, limited to the 209 randomly chosen instances from SAT Competition 2023.

	K						KCL						Ours					
cores	#	+	−	PAR	S_g	S_t	#	+	−	PAR	S_g	S_t	#	+	−	PAR	S_g	S_t
48	122	57	65	429.9	7.6	20.0	134	55	79	411.5	12.8	16.7	142	61	81	400.1	8.7	25.0
96	128	62	66	419.7	10.9	40.9	136	55	81	406.6	16.7	23.0	146	62	84	394.0	12.3	39.1
192	129	62	67	418.0	13.1	43.5	141	57	84	398.8	20.4	39.6	148	62	86	388.7	15.5	56.9
384	132	62	70	412.3	16.5	72.5	147	58	89	391.6	23.5	60.2	151	63	88	384.0	16.8	91.3
768	137	64	73	406.5	17.0	70.5	150	61	89	386.5	28.0	74.5	159	67	92	373.2	21.5	116.9
1536	142	66	76	398.8	18.1	80.5	151	60	91	384.4	30.8	81.5	157	65	92	374.5	23.4	120.6
3072	146	67	79	393.8	18.9	83.5	160	68	92	372.2	33.2	97.8	159	65	94	371.3	25.2	158.9

Table 6 Performance as in Tab. 1, limited to the 191 randomly chosen instances from SAT Competition 2024.

	K						KCL						Ours					
cores	#	+	−	PAR	S_g	S_t	#	+	−	PAR	S_g	S_t	#	+	−	PAR	S_g	S_t
48	148	75	73	396.5	8.1	16.6	148	74	74	397.1	9.3	19.1	152	74	78	391.1	8.2	16.8
96	158	78	80	382.4	11.3	22.6	157	77	80	382.1	12.5	25.1	156	75	81	383.0	10.9	25.0
192	163	81	82	373.3	13.7	32.6	168	83	85	366.5	15.6	33.7	163	80	83	372.4	14.1	31.3
384	165	82	83	368.4	17.6	39.7	168	82	86	363.5	20.1	42.5	167	82	85	363.6	17.6	45.3
768	168	83	85	363.0	20.8	46.9	169	82	87	360.4	24.4	52.1	168	82	86	360.4	21.9	53.7
1536	172	85	87	357.0	24.1	55.8	172	84	88	354.6	29.6	64.4	172	84	88	352.9	25.8	76.8
3072	172	85	87	354.7	28.0	67.6	175	86	89	350.5	35.6	81.0	175	86	89	348.1	28.1	76.4



	#	+	−	PAR2
Ours	332	148	184	1857.0
Ours+	**340**	154	**186**	**1705.3**
Painless	**340**	**159**	181	1721.4

Figure 15 Performance of our approach from the paper ("Ours"), an enhanced version ("Ours+") with added Lingeling-based preprocessing and each 20th thread running YalSAT, and state-of-the-art shared-memory solver PL-PRS-BVA-KISSAT, at a single node (48 cores) and for up to 5000 s of running time, as in the SAT Competition Parallel tracks. Note the y axis offset.

CNOT-Optimal Clifford Synthesis as SAT

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

Clifford circuit optimization is an important step in the quantum compilation pipeline. Major compilers employ heuristic approaches. While they are fast, their results are often suboptimal. Minimization of noisy gates, like 2-qubit CNOT gates, is crucial for practical computing. Exact approaches have been proposed to fill the gap left by heuristic approaches. Among these are SAT based approaches that optimize gate count or depth, but they suffer from scalability issues. Further, they do not guarantee optimality on more important metrics like CNOT count or CNOT depth. A recent work proposed an exhaustive search only on Clifford circuits in a certain normal form to guarantee CNOT count optimality. But an exhaustive approach cannot scale beyond 6 qubits.

In this paper, we incorporate search restricted to Clifford normal forms in a SAT encoding to guarantee CNOT count optimality. By allowing parallel plans, we propose a second SAT encoding that optimizes CNOT depth. By taking advantage of flexibility in SAT based approaches, we also handle connectivity restrictions in hardware platforms, and allow for qubit relabeling. We have implemented the above encodings and variations in our open source tool Q-Synth.

In experiments, our encodings significantly outperform existing SAT approaches on random Clifford circuits. We consider practical VQE and Feynman benchmarks to compare with TKET and Qiskit compilers. In all-to-all connectivity, we observe reductions up to 32.1% in CNOT count and 48.1% in CNOT depth. Overall, we observe better results than TKET in the CNOT count and depth. We also experiment with connectivity restrictions of major quantum platforms. Compared to Qiskit, we observe up to 30.3% CNOT count and 35.9% CNOT depth further reduction.

2012 ACM Subject Classification Hardware → Quantum computation; Computing methodologies → Planning for deterministic actions

Keywords and phrases Circuit Synthesis, Circuit Optimization, Quantum Circuits, Propositional Satisfiability, Parallel Plans, Clifford Circuits, Encodings

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.28

Related Version *Extended Version*: `https://arxiv.org/abs/2504.00634` [26]

Supplementary Material *Software (Github)*: `https://github.com/irfansha/Q-Synth`
archived at `swh:1:dir:f934bdca528231c8f64a232248ae97bed6d26102`
Software (Zenodo Link): `https://doi.org/10.5281/zenodo.15575214`

Funding This research was partially funded by the Innovation Fund Denmark project "Automated Planning for Quantum Circuit Optimization". This research was also partially funded by the European Innovation Council through Accelerator grant no. 190124924.

Acknowledgements We thank L. Burgholzer for his generous help with settings of the QMAP software for Clifford Synthesis. The experiments were carried out on the Grendel cluster at the Centre for Scientific Computing, Aarhus (`http://www.cscaa.dk/grendel/`). We thank the anonymous reviewers of SAT 2025 for their useful suggestions.



© Irfansha Shaik and Jaco van de Pol;
licensed under Creative Commons License CC-BY 4.0



1 Introduction

Quantum Computing promises an alternative solution to some challenging computational problems that are out-of-reach for classical computers. While several competing quantum platforms exist in the current Noisy Intermediate Scale Quantum (NISQ) era, they all come with different strengths and weaknesses. Quantum programs must be compiled to low level quantum circuits satisfying target hardware requirements before execution. While most quantum platforms accept 1-qubit and 2-qubit gates, their native gate-sets can vary. Currently, no quantum platform has the best qubit count, fidelity, and latency all together. Circuit optimization can play a crucial role for practical quantum computing in both current NISQ or future fault-tolerant processors. For example, [10] observed that IonQ's Aria quantum platform had a limit of 950 single qubit gates. Only by using circuit optimization techniques they were able to run the required circuits. For satisfying hardware requirements, Layout Synthesis and Circuit Synthesis are two main steps. In Layout Synthesis, circuits are synthesized to handle hardware layout restrictions. Often, not all qubits are connected in a quantum platform thus 2-qubit quantum gates can only be scheduled on neighboring qubits. Circuit Synthesis involves either synthesizing to required gate-set or optimizing some circuit metric for practical quantum computing.

We consider circuit optimization in this paper. Optimal synthesis of an arbitrary n-qubit circuit requires considering $2^n \times 2^n$ unitary matrices of complex numbers. While optimal circuit synthesis is ideal, it is a challenging computational problem [16]. Instead, peephole optimization is often used where *easier* sub-circuits are optimized [24]. In this paper, we focus on an interesting subclass of circuits called *Clifford circuits*. Any circuit composed of 1-qubit Hamard (H) and Phase (S) gates, and 2-qubit Conditional-Not (CNOT) gates is a Clifford circuit. While polynomially simulatable, Clifford circuits capture important phenomena like entanglement and superposition, and have applications in teleportation and dense quantum encoding [1]. Clifford circuits are also key in quantum error correction, required for the future fault-tolerant hardware. Synthesis of Clifford circuits only requires $(2n) \times (2n + 1)$ Boolean matrices, using the *stabilizer* formalism [1], instead of full unitary matrices. Using peephole optimization, one can replace Clifford sub-circuits with their optimized counterparts.

There is no single optimization metric that can *predict* actual hardware performance perfectly. Quantitative metrics, like qubit fidelity, provide good predictions, but lead to numerical optimization problems. Simple metrics like gate count, circuit depth are preferred for optimization [15, 23, 28, 29]. In NISQ processors, 2-qubit CNOT gates are up to 10 times more error-prone than 1-qubit gates. Industrial compilers, like Qiskit [21] by IBM and TKET [28] by Quantinuum, apply Clifford optimization reducing CNOT gate-count (cx-count) and CNOT depth (cx-depth). Given a circuit with unary gates and CNOT gates, cx-depth is the maximum number of CNOT gates that are executed in series. In a recent survey paper [17], the authors extensively compared major compilers on cx-count/cx-depth reduction. While several approaches exist for Clifford optimization, industrial compilers mainly focus on heuristic approaches. Exact optimization approaches unfortunately suffer from scalability problems. Optimization of Clifford circuits is NP-hard [13]. Even approximation of optimal synthesis is NP-hard [12], thus there is no efficient algorithm unless P = NP. This undesirable gap between heuristic and exact approaches is well established in the literature [30, 20].

Thus, there is a need to improve the scalability of exact approaches. In recent years, several SAT based approaches were proposed for problems like Layout Synthesis [30, 25] and CNOT synthesis [24]. In [22], authors proposed a SAT encoding for cx-count optimization in Clifford circuits based on bounded reachability. Unfortunately, their encoding does not

guarantee optimality due to the use of asymptotic cx-count upper bound for termination criteria. Despite being a near-optimal encoding, their approach still suffers from scalability issues (even for $n > 3$ qubits). In [20], the authors proposed an improved SAT encoding instead optimizing circuit depth that can handle up to 5-qubit circuits. While this improved scalability, their synthesis is focused on circuit depth, rather than cx-count and cx-depth. Our experiments (Section 5) show that synthesized circuits with minimal circuit depth can have worse cx-count/cx-depth, even compared to heuristic tools.

Using some special properties of Clifford circuits, one can guarantee cx-count optimality. In [5], authors proposed circuit synthesis restricted to certain normal forms. They observed that by ignoring so-called phase updates one can guarantee cx-count optimality. Using brute force search, they successfully generated a 2.1TB database for all cx-count optimal 6-qubit circuits using up to 100 days of computation. While useful, generating such a database for beyond 6-qubits is not practical [5]. Further, such an approach is not flexible i.e., a new database needs to be generated for each optimization metric/criteria. For example, layout aware optimal Clifford circuits can vary depending on the platform layout restrictions. Alternatively, flexibility of SAT like approaches can be used for efficient on-demand computation.

Our Contribution

In this paper, we incorporate the ideas proposed in [5] to obtain efficient SAT encodings that guarantee optimal Clifford circuits. We provide the first *cx-count* optimal SAT encoding based on bounded reachability. Restricting the search to circuits in normal form reduces the search space (and consequently the makespan of the encoding) significantly. Further, we propose the first *cx-depth* optimal approach to Clifford circuit synthesis, by adapting the SAT encoding. We have extended our open source tool Q-Synth[1] with version 5 implementing the above encodings. For an experimental comparison, we propose three experiments. In Experiment 1, we compare existing SAT encodings with ours on random Clifford circuits of 3 to 7 qubits. For both cx-count/cx-depth optimization, we significantly outperform previous approach while solving optimally. We show that our approach can optimally solve 4 out of 5 7-qubit circuits for the first time for cx-depth optimization. In Experiment 2, we demonstrate our effectiveness on practical VQE and Feynman [2] benchmarks via peephole optimization. We generate 12 VQE benchmarks of 8 and 16 qubits, and 28 T-gate optimized Feynman benchmarks of 5 to 24 qubits. Given a 10-minute time limit, we consistently outperform TKET, and we observe the best results when TKET+Q-Synth are used together. In Experiment 3, we focus on layout aware re-synthesis of practical VQE and Feynman benchmarks. We first use Qiskit [21] to map VQE and Feynman benchmarks onto 54-qubit Sycamore [3], 80-qubit Rigetti [7], and 127-qubit Eagle [6] platforms. We then re-synthesize each benchmark with Q-Synth giving a 10-minute time limit. Overall, we observe significant reduction on all platforms in both cx-count and cx-depth optimizations. In VQE benchmarks, we observe a reduction of up to 19.3% cx-count and 27.4% cx-depth. In Feynman benchmarks, we observe a reduction of up to 30.3% cx-count and 35.9% cx-depth. Our experiments indicate that there is a place for SAT like approaches in the quantum compilation pipeline.

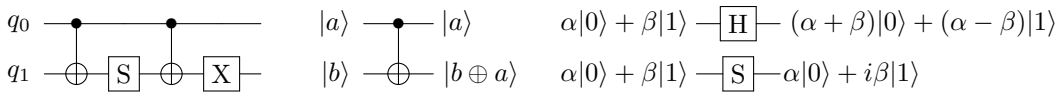
[1] Q-Synth v5 tool with source code, benchmarks, and scripts: `https://github.com/irfansha/Q-Synth`

2 Preliminaries

2.1 Clifford Circuits

For a textbook description of quantum states, gates and circuits we refer to [18]. We also provide a short summary of quantum notation in Section A.

In classical computing, the fundamental unit of information, a classical bit, has only two states 0 and 1. In Quantum computing, a quantum bit instead is a superposition of 0 and 1. One can represent a qubit state as a vector $\alpha|0\rangle + \beta|1\rangle$ where $\alpha, \beta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 = 1$. States over n-qubits live in the tensor product space on all vectors on n qubits. Quantum gates change the state by acting on one or more qubits. Due to practical difficulties of implementing muti-qubit quantum gates, quantum platforms typically only apply 1-qubit and 2-qubit gates. High level quantum programs are decomposed to low level circuits with 1- and 2-qubit gates before execution. In this paper, we focus on Clifford circuits made of {CNOT, H, S} gates (cf. Section A). Figure 1b shows how the Clifford gates change the state of qubits. A binary CNOT gate (Conditional NOT, also abbreviated as CX) negates the target qubit (b) when the control qubit (a) is on. This entangling gate can also be expressed as an exclusive OR. The H gate brings a qubit in a superposition, and the S gate applies a phase change to a qubit. Pauli gates {X, Y, Z} are composed of Clifford gates, where X = HSSH, Z = SS and Y = iXZ. We will use Pauli gates later in the paper for so-called relative phase recovery. Figure 1a shows an example Clifford circuit with {CNOT, S, X} gates.



(a) A 2-qubit Clifford circuit. **(b)** State updates by CNOT, H, and S gates.

Figure 1 Example Clifford circuit, and an illustration of state updates by Clifford gates.

Interestingly, Clifford gates are not universal and can be simulated in polynomial time and space. In [1], Aaronson and Gottesman proposed *stabilizer* formalism to efficiently represent Clifford circuits. For an n-qubit Clifford circuit, we only need a $(2n) \times (2n+1)$ Boolean matrix, called *tableau*. Two Clifford circuits are equivalent (up to global phase) if their corresponding tableaux are equal [5]. The following matrix shows the structure of a n-qubit tableau:

$$\left(\begin{array}{ccc|ccc|c} q_0 & \cdots & q_{n-1} & q_0 & \cdots & q_{n-1} & \\ x_{00} & \cdots & x_{0(n-1)} & z_{00} & \cdots & z_{0(n-1)} & r_0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{(2n-1)0} & \cdots & x_{(2n-1)(n-1)} & z_{(2n-1)0} & \cdots & z_{(2n-1)(n-1)} & r_{2n-1} \end{array}\right)$$

An n-qubit tableau is made of $x_{2n\times n}$, $z_{2n\times n}$, and $r_{2n\times 1}$ Boolean matrices. A tableau essentially represents so-called *stabilizer* and *destabilizer* generators. We refer to [1] for a detailed explanation on stabilizer generators, Clifford circuits, and relevant proofs. For the scope of this paper, it is sufficient to understand how a tableau can be computed given a Clifford circuit. The initial tableau, representing an empty circuit, is defined as $x_{ii} = z_{(n+i)i} = 1$ (for $0 \le i < n$) and every other cell is 0.

Table 1 Tableau update rules for every row i, when applying Clifford gates to qubits a and b.

	Base Gates			Pauli gates		
	$\mathrm{CNOT}_{a,b}$	H_a	S_a	X_a	Y_a	Z_a
x_{ia}	x_{ia}	z_{ia}	x_{ia}	x_{ia}	x_{ia}	x_{ia}
z_{ia}	$z_{ia} \oplus z_{ib}$	x_{ia}	$x_{ia} \oplus z_{ia}$	z_{ia}	z_{ia}	z_{ia}
x_{ib}	$x_{ia} \oplus x_{ib}$	-	-	-	-	-
z_{ib}	z_{ib}	-	-	-	-	-
r_i	$r_i \oplus x_{ia}z_{ib}(x_{ib} \oplus z_{ia} \oplus 1)$	$r_i \oplus x_{ia}z_{ia}$	$r_i \oplus x_{ia}z_{ia}$	$r_i \oplus z_{ia}$	$r_i \oplus x_{ia} \oplus x_{ib}$	$r_i \oplus x_{ia}$

For each gate in a given circuit, we update every row of the tableau according to the rules in Table 1; this corresponds to column additions modulo 2. For our example Clifford circuit from Figure 1a, the following sequence of matrices represent tableau updates:

$$\left(\begin{array}{cc|cc|c} 1&0&0&0&0\\ 0&1&0&0&0\\ \hline 0&0&1&0&0\\ 0&0&0&1&0 \end{array}\right) \xrightarrow[q_0,q_1]{\mathrm{CX}} \left(\begin{array}{cc|cc|c} 1&\mathbf{1}&0&0&0\\ 0&1&0&0&0\\ \hline 0&0&1&0&0\\ 0&0&\mathbf{1}&1&0 \end{array}\right) \xrightarrow[q_1]{\mathrm{S}} \left(\begin{array}{cc|cc|c} 1&1&0&\mathbf{1}&0\\ 0&1&0&\mathbf{1}&0\\ \hline 0&0&1&0&0\\ 0&0&1&1&0 \end{array}\right) \xrightarrow[q_0,q_1]{\mathrm{CX}} \left(\begin{array}{cc|cc|c} 1&\mathbf{0}&\mathbf{1}&1&0\\ 0&1&\mathbf{1}&1&0\\ \hline 0&0&1&0&0\\ 0&0&\mathbf{0}&1&0 \end{array}\right) \xrightarrow[q_1]{\mathrm{X}} \left(\begin{array}{cc|cc|c} 1&0&1&1&\mathbf{1}\\ 0&1&1&1&\mathbf{1}\\ \hline 0&0&1&0&0\\ 0&0&0&1&\mathbf{1} \end{array}\right)$$

The leftmost matrix corresponds to the initial tableau and the rightmost matrix is our target tableau. Optimal synthesis of Clifford circuits can now be transformed to a graph search problem. Each node of the graph is labelled with a tableau, and edges labeled with Clifford gates correspond to the update rules. Synthesizing the optimal number of gates boils down to finding the shortest path in this graph. On the other hand, optimizing the number of CNOT gates is not so straightforward. We will revisit this in Section 3.

2.2 Layout Restrictions and Qubit Relabeling

In high-level quantum circuits, binary CNOT gates can be applied to any pair of qubits. However, in many physical realizations (in particular on superconducting quantum computers), binary gates can only be applied to physical neighbour qubits. We assume that this is specified in some coupling graph (Q, E), where Q are the qubits and $q\,E\,q'$ if a CNOT is possible between qubits q and q'. This graph is usually quite sparse and mostly planar.

Most existing Clifford synthesis approaches [5, 28, 21, 20] assume all-to-all qubit connectivity. Under this assumption, local rewrite rules can be used effectively. While useful, such structure breaks down in case of restricted qubit connectivity. Thus, compilers like Qiskit and TKET apply Clifford synthesis before the layout synthesis phase, leading to suboptimal results overall. Some heuristic approaches [9] exist that handle connectivity restrictions in related problems like CNOT synthesis. In [24], we showed that connectivity restrictions can be encoded elegantly using SAT in CNOT synthesis. Given a platform connectivity graph, we will only allow 2-qubit gates on neighboring qubits. Thus, our approach can be used post layout synthesis, allowing further reduction.

In tableau representation, notice that the columns are labelled with qubits. Allowing permutation of columns essentially corresponds to relabeling qubits. In a relaxed notion of equivalence, two Clifford circuits are equivalent if their tableau are equivalent up to a column permutation. Allowing any column permutation of initial tableau can result in better circuits. Major compilers like TKET also allow qubit relabeling in the context of all-to-all connectivity. In [24], we showed that permutation of a similar matrix can be elegantly encode using Exactly-One constraints. We adapt the same idea for Clifford synthesis, we encode permutation of columns in initial tableau using cardinality constraints.

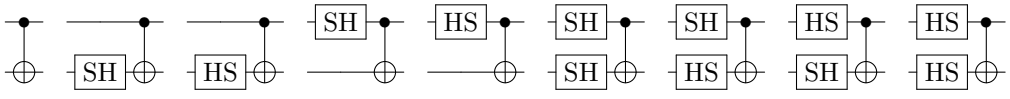
3 Revisiting Clifford Normal Forms for Optimal CNOT Synthesis

In earlier sections, we briefly discussed that a simple bounded reachability encoding is sufficient for gate count optimality. For cx-count optimization, existing SAT approach in [22] proposed a MAXSAT like formulation to optimize cx-count for the given gate count. While they can synthesize circuits with better cx-count, they do not ensure cx-count optimality. Consider our example circuit 1a with 4 gates and 2 CNOTs. If we run QMAP tool which implements SAT encoding in [22] on our circuit, it fails to produce a circuit with better cx-count. This makes sense as there does not exist a 4 gate circuit with better cx-count. However, if we consider circuits with higher gate count, there does exist a circuit with a single CNOT gate as shown in Figure 3a. So the given number of Clifford gates in the input circuit is not a valid upperbound. Also, the upperbound of $\Theta(n^2/log(n))$ gates for an n-qubit Clifford circuit [19, 1] is only asymptotically sharp, so it doesn't provide a reliable upper bound for concrete n.

To ensure cx-count optimality, we need to reformulate the search problem. Let q, q' be two qubits, we define the set of *entangling* sequences $E = \{\Psi\text{CNOT}_{q,q'} \mid \Psi \in \{\text{H}_q, \text{H}_{q'}, \text{S}_q, \text{S}_{q'}\}^*\}$. Each entangling sequence, by definition, adds a single CNOT gate to the circuit. A d-CNOT optimal circuit implies that there does not exist a Clifford circuit with $< d$ entangling sequences. This modification results in a graph search problem where nodes are still labelled with tableaux. Edges on the other hand are labelled with entangling sequences instead of individual gates. Finding an optimal CNOT circuit corresponds to finding the shortest path in the new graph. While this new formulation guarantees CNOT optimality, the problem of arbitrary single gate sequences still remains. Taking advantage of equivalences between Clifford circuits, [5] proposed a normal form that considers only 9 unique entangling sequences. In the following section, we will revisit observations made in [5] and adapt in the context of a SAT encoding.

3.1 Clifford Normal Forms

Remember that a tableau is made of x, z, r matrices. [5] observed that two tableau with same x and z matrices have same optimal cx-count. From Table 1, we can see that Pauli gates only update r column. Indeed, one can synthesize any circuit with differing r column by appending Pauli gates at the front. The authors in [5] consider all circuits with differing r column as an equivalence class. We will revisit r column updates in detail later in this Section. We ignore the r column updates aka relative phase updates for now. This makes the tableau update rules simpler, resulting in fewer unique 1-qubit gate sequences. Turns out there are not many sequences of H and S gates that result in unique tableau state. Clearly, HH and SS cancel out which is equivalent to an Identity gate I. Thus only alternating H and S sequences of gates are interesting. So the interesting sequences are {I, H, S, HS, SH, HSH, SHS, HSHS, SHSH, ...}. Further, HSH and SHS result in the same tableau state. Using the equivalence of HSH and SHS, we can simplify any non-trivial 1-q gate sequence to a sequence of length less than 4. For example, consider HSHS gate sequence and replace HSH with SHS resulting in SHSS. Since SS cancel out, we are left with SH sequence. Any such sequence can be replaced with one of the 6 {I, H, S, HS, SH, HSH} sequences, defined as 1-q unique sequences. Considering these 6 sequences on each qubit followed by a CNOT results in 36 sequences. [5] showed that only the following 9 sequences are unique entangling sequences.



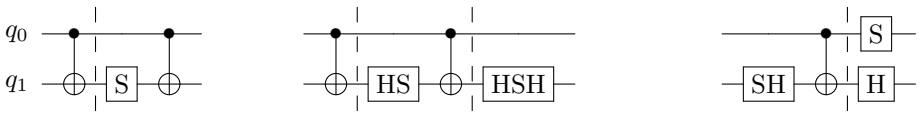
They showed that each of the 36 entangling sequences can be rewritten as one of the above 9 sequences followed by one of the 1-q unique sequences on each qubit. Note that, a CNOT gate $\mathrm{CNOT}_{j,i}$ can be rewritten as $\mathrm{H}_i\mathrm{H}_j\mathrm{CNOT}_{i,j}\mathrm{H}_i\mathrm{H}_j$. Thus, we can rewrite all CNOT gates in a given circuit such that the control qubit is less than its target qubit. Now, let us suppose, we have an entangling sequence in the form $\mathrm{A}_i\mathrm{B}_j\mathrm{CNOT}_{i,j}$ where A, B are elements of 1-q unique sequences. Such a sequence can be rewritten to the form $\mathrm{A'}_i\mathrm{B'}_j\mathrm{CNOT}_{i,j}\mathrm{C}_i\mathrm{D}_j$ where A', B' are in {I, HS, SH} and C, D are 1-q unique sequences. Using these rewrite rules, from start to end of the circuit, one can push 1-q sequences that are not {I, HS, SH} to the last layer. The final rewritten circuit will only have the above 9 unique entangling sequences followed by 1-q sequences on every qubit.

Consider our example circuit without X gate as in 2a. First CNOT is part of unique entangling sequence, however, S gate followed by second CNOT is not. One can replace $\mathrm{S}_{q_1}\mathrm{CNOT}_{q_0,q_1}$ by $\mathrm{HS}_{q_1}\mathrm{CNOT}_{q_0,q_1}\mathrm{HSH}_{q_1}$. The resulting circuit as shown in 2b has both CNOTs within unique sequences followed by 1-q unique sequences. Figure 2c shows an equivalent circuit with only 1 CNOT. We illustrate entangling sequences in Figure 2 with dotted lines. Authors in [5] consider all circuit with same entangling sequences as part of an equivalence class. However, for a SAT formulation we need to encode the final layer of 1-q sequences explicitly.

Using the above observations, synthesizing d-CNOT circuit corresponds to finding d entangling sequences followed by 1-q unique sequences. For synthesizing d-CNOT optimality, we need to show there does not exist a normal form circuit with $< d$ entangling sequences. We can easily adapt these observations to handle CNOT-depth optimal synthesis. Allowing parallel entangling sequences on independent qubits at each step allows CNOT-depth optimization. This notion is exactly same as $\forall$-step parallel plans [14] in the context of SAT based planning. In other words, sequential encoding corresponds to cx-count optimization whereas parallel encoding corresponds to cx-depth optimization. In Section 4, we present encodings for both cx-count and cx-depth optimization.

Phase Recovery for Peephole Setting

Earlier in this Section, we ignored phase updates when synthesizing Clifford gates. Thus, the optimized Clifford circuit (either by cx-count or cx-depth) will have a different r-column in the tableau matrix. While relative phase can be ignored for pure Clifford circuit optimization, for general quantum circuits (i.e., in particular for peephole-optimization), we need to reconstruct the relative phase (r-column). This can be achieved separately for each qubit, by appending single Pauli gates to the optimized circuit. Appending Pauli gates at the beginning of the circuit is same as so-called *Pauli left multiplication*. In plain words, adding either X or Z Pauli gates at the beginning of the circuit flips the target tableau r-column. Remember that Pauli gates do not change x, z matrices but only act on the r-column. Phase recovery is then figuring out which Pauli gates to be applied to recover our target r-column. We observe that applying a Z_i gate flips r_i whereas X_i gate flips r_{n+i} in the target tableau.



(a) Without X gate. **(b)** 2 CNOT Normal form circuit. **(c)** 1 CNOT Normal form circuit.

Figure 2 Equivalent Clifford circuits of Figure 1a, ignoring relative phase.

Let r, r' be r-columns of optimized and target tableau of an n-qubit circuit. Here, we present a simple algorithm that computes the sequence of X and Z gates that recovers the relative phase. Algorithm 1 clearly runs in linear time in the number of qubits. Consider the CNOT optimal example circuit in Figure 2c: the tableau deviates in the r-column. Appending X and Z gates as shown in Figure 3a recovers the r-column. This transformation is shown in Figure 3b. In our tool, we apply the recovery algorithm after synthesizing the optimal circuit. After every synthesis run, we compare the tableau of input and output circuits for correctness.

Algorithm 1 Relative Phase Recovery.

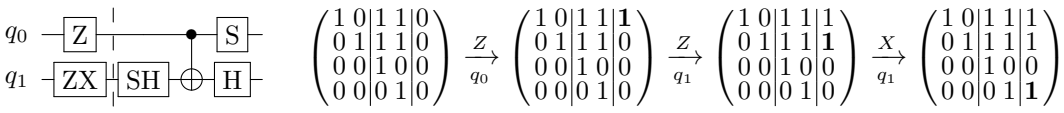
```
SEQ is empty
for all i ∈ [0 ... n−1] do
    if r_i ⊕ r'_i = 1 then
        APPEND Z_i to SEQ
    if r_{n+i} ⊕ r'_{n+i} = 1 then
        APPEND X_i to SEQ
```

4 CNOT Optimal SAT Encodings

We now present bounded reachability SAT encodings for cx-count and cx-depth optimization based on the ideas discussed above. For cx-count optimization, we need to reach the target tableau in d entangling steps from the initial tableau. Encoding tableau updates for the entangling sequences across two time steps is quite complex. Instead, we define a *layer* that acts across three time steps. To encode layer k constraints, we encode 1-qubit gate updates from time step $2k$ to $2k+1$ and CNOT updates from time step $2k+1$ to $2(k+1)$. In each layer, we choose a control and target qubit on which the entangling sequence is applied. At the end, we need to apply a last layer of all 6 1-q sequences. In total, we need $2d+2$ time steps i.e., from $t=0$ to $t=2d+1$ for a d-CNOT circuit encoding. Algorithm 2 presents the overall outline of the SAT encoding for a d-CNOT, n-qubit circuit. For cx-depth optimization, instead of exactly one control and target, we will allow any subset of independent (i.e., parallel) qubit pairs. Essentially, optimizing for the number of layers corresponds to optimizing for the chosen cx-count or cx-depth metric.

4.1 Gate optimal encoding

We define the variables as shown in Table 2. The variables in top half represent chosen control and target qubits, and applied gates. Variables in bottom half represent states of matrix elements. In addition, we define auxiliary variables to encode XOR constraints elegantly. For time steps $t=0$ to $t=2k$, we set $\mathrm{px}^t_{i,a}$ true iff $\mathrm{x}^t_{i,a}$ and $\mathrm{x}^{t+1}_{i,a}$ are equal. Similarly, we



(a) Optimal circuit. **(b)** Tableau transforms to correct phase.

Figure 3 Phase recovery applied to the Clifford circuit of Figure 2c.

Algorithm 2 SAT Encoding Outline for d-CNOT Circuit.

```
InitialStateConstraints          ▷Encode initial tableau at t = 0 as in 4.1.4
GoalStateConstraints             ▷Encode target tableau at t = 2d + 1 as in 4.1.4
for all k ∈ [0 . . . d − 1] do
    Encode1qConstraints(k)       ▷Encode I,HS,SH constraints in kth layer as in 4.1.1
    EncodeCNOTConstraints(k)     ▷Encode CNOT constraints in kth layer as in 4.1.2
Last1qConstraints                ▷Encode last 1q constraints at t = 2d as in 4.1.3
```

Table 2 Encoding variables and descriptions.

Variable	Description
id_a^k / hs_a^k / sh_a^k	apply I/HS/SH gate on ath qubit in the layer k
$\mathrm{cnot}_{a,b}^k$	apply CNOT gate on a, b qubits ($a < b$) in the layer k
ctrl_a^k / trg_a^k	choose a as a control/target qubit in the layer k
id_a^L / h_a^L / s_a^L / hs_a^L / sh_a^L / hsh_a^L	apply I/H/S/HS/SH/HSH gates on ath qubit in the last layer
$\mathrm{x}_{i,a}^t$ / $\mathrm{z}_{i,a}^t$	state of x_{ia}/z_{ia} matrix element at time step t
$\mathrm{px}_{i,a}^t$ / $\mathrm{pz}_{i,a}^t$	propagate x_{ia}/z_{ia} matrix element from time step t to $t+1$

set $\mathrm{pz}_{i,a}^t$ true iff $\mathrm{z}_{i,a}^t$ and $\mathrm{z}_{i,a}^{t+1}$ are equal. We essentially represent propagation of x, z matrix states from t to $t+1$ time steps. In the following Subsections, we present constraints for each macro in Algorithm 2.

4.1.1 Encoding 1-q I, HS, SH constraints

For a given layer number k, we define $t := 2k$. Encode1qConstraints(k) is then defined by the following constraints. Exactly one of the 3 single qubit gates is chosen on each qubit. We apply sequential counter ExactlyOne constraints from PySAT [11], referred as EO from here on. If a qubit is neither a control nor target, then we apply identity gate. Instead, if HS or SH gate is applied, the qubit must be either a control or target.

$$\bigwedge_{a=0}^{n-1} \mathrm{EO}(\mathrm{id}_a^k, \mathrm{hs}_a^k, \mathrm{sh}_a^k) \wedge \bigwedge_{a=0}^{n-1} (\neg\,\mathrm{ctrl}_a^k \wedge \neg\,\mathrm{trg}_a^k) \implies \mathrm{id}_a^k \wedge \bigwedge_{a=0}^{n-1} (\mathrm{hs}_a^k \vee \mathrm{sh}_a^k) \implies (\mathrm{ctrl}_a^k \vee \mathrm{trg}_a^k) \quad (1)$$

For each of the three gates I, HS, SH, we encode following constraints:

$$\bigwedge_{a=0}^{n-1} \bigwedge_{i=0}^{2n-1} (\quad (2)$$

$$\mathrm{id}_a^k \implies (\mathrm{px}_{i,a}^t \wedge \mathrm{pz}_{i,a}^t) \qquad \triangleright \mathrm{I}\colon\ x_{ia} := x_{ia};\ z_{ia} := z_{ia}$$

$$\mathrm{hs}_a^k \implies \big((\mathrm{z}_{i,a}^t \iff \mathrm{x}_{i,a}^{t+1}) \wedge (\mathrm{x}_{i,a}^t \iff \neg\,\mathrm{pz}_{i,a}^t)\big) \qquad \triangleright \mathrm{HS}\colon\ x_{ia} := z_{ia};\ z_{ia} := x_{ia} \oplus z_{ia}$$

$$\mathrm{sh}_a^k \implies \big((\mathrm{x}_{i,a}^t \iff \mathrm{z}_{i,a}^{t+1}) \wedge (\mathrm{z}_{i,a}^t \iff \neg\,\mathrm{px}_{i,a}^t)\big)\big) \qquad \triangleright \mathrm{SH}\colon\ z_{ia} := x_{ia};\ x_{ia} := z_{ia} \oplus x_{ia}$$

4.1.2 Encoding CNOT constraints

Let k be the layer number, then we define $t := 2k + 1$. EncodeCNOTConstraints(k) defines the CNOT constraints as follows. For cx-count optimization, we schedule exactly one CNOT gate in every layer and the chosen CNOT gate defines the control and target qubits.

$$\mathrm{EO}(\{\mathrm{cnot}^k_{a,b} \mid 0 \le a < n; a < b < n\}) \wedge \bigwedge_{a=0}^{n-1} \bigwedge_{b=a+1}^{n-1} \mathrm{cnot}^k_{a,b} \iff (\mathrm{ctrl}^k_a \wedge \mathrm{trg}^k_b) \tag{3}$$

Similar to 1q constraints, we define the tableau updates rules as below.

$$\bigwedge_{a=0}^{n-1} \bigwedge_{b=a+1}^{n-1} \bigwedge_{i=0}^{2n-1} \mathrm{cnot}^k_{a,b} \implies \big((\mathrm{x}^t_{i,a} \iff \neg\, \mathrm{px}^t_{i,b}) \wedge (\mathrm{z}^t_{i,b} \iff \neg\, \mathrm{pz}^t_{i,a})\big) \tag{4}$$

$$\triangleright\, x_{ib} := x_{ia} \oplus x_{ib};\ z_{ia} := z_{ib} \oplus z_{ia}$$

$$\bigwedge_{a=0}^{n-1} \bigwedge_{i=0}^{2n-1} (\neg\, \mathrm{trg}^k_a \implies \mathrm{px}^t_{i,a}) \wedge \bigwedge_{b=0}^{n-1} \bigwedge_{i=0}^{2n-1} (\neg\, \mathrm{ctrl}^k_b \implies \mathrm{pz}^t_{i,b}) \quad \triangleright\, x_{ia} := x_{ia};\ z_{ib} := z_{ib} \tag{5}$$

Given a coupling, for every non-neighbor (a, b) qubit pair we add $\neg\, \mathrm{cnot}^t_{a,b}$.

4.1.3 Final Layer 1q Constraints

Let $t = 2d$, Last1qConstraints is defined by the following constraints. We apply exactly one of the 6 1-q unique gate sequences for each qubit a as $\bigwedge_{a=0}^{n-1} \mathrm{EO}(\mathrm{id}^L_a, \mathrm{h}^L_a, \mathrm{s}^L_a, \mathrm{hs}^L_a, \mathrm{sh}^L_a, \mathrm{hsh}^L_a)$. The constraints for gate sequences {I, HS, SH} are exactly same as transition constraints presented earlier as in 4.1.1. We now give constraints for rest of the three gate sequences.

$$\bigwedge_{i=0}^{n-1} \bigwedge_{r=0}^{2n-1} (\tag{6}$$

$$\begin{aligned}
&\mathrm{h}^L_i \implies \big((\mathrm{z}^t_{i,a} \iff \mathrm{x}^{t+1}_{i,a}) \wedge (\mathrm{x}^t_{i,a} \iff \mathrm{z}^{t+1}_{i,a})\big) && \triangleright \mathrm{H}\text{: } x_{ia} := z_{ia};\ z_{ia} := x_{ia} \\
&\mathrm{s}^L_i \implies \big(\mathrm{px}^t_{i,a} \wedge (\mathrm{x}^t_{i,a} \iff \neg\, \mathrm{pz}^t_{i,a})\big) && \triangleright \mathrm{S}\text{: } x_{ia} := x_{ia};\ z_{ia} := x_{ia} \oplus z_{ia} \\
&\mathrm{hsh}^L_i \implies \big(\mathrm{pz}^t_{i,a} \wedge (\mathrm{z}^t_{i,a} \iff \neg\, \mathrm{px}^t_{i,a})\big)\big) && \triangleright \mathrm{HSH}\text{: } z_{ia} := z_{ia};\ x_{ia} := z_{ia} \oplus x_{ia}
\end{aligned}$$

4.1.4 Initial and Goal Constraints

Recall our discussion on tableau structure and initial tableau in Section 2.1. Let us divide the tableau without r-column into 4 equal quadrants. The left quadrants corresponds to the x matrix and the right quadrants correspond to the z matrix, where columns are labelled with qubits. Initially, the top-left and bottom-right quadrants correspond to the Identity matrix, whereas the other quadrants are zero matrices. Permuting columns of x and z matrices corresponds to qubit relabeling. Exactly-One constraints on rows and column of top-left quadrant represents column permutation.

$$\begin{aligned}
\bigwedge_{i=0}^{n-1} \big(\bigwedge_{a=0}^{n-1} \big((\mathrm{x}^0_{i,a} \iff \mathrm{z}^0_{n+i,a}) \wedge \neg\, \mathrm{x}^0_{n+i,a} \wedge \neg\, \mathrm{z}^0_{i,a}\big) \wedge \\
\mathrm{EO}(\{\mathrm{x}^0_{i,a} \mid 0 \le a < n\}) \wedge \mathrm{EO}(\{\mathrm{x}^0_{a,i} \mid 0 \le a < n\})\big)
\end{aligned} \tag{7}$$

We can disable column permutation (qubit relabeling) by forcing top-left quadrant as the Identity matrix i.e., adding clauses $\bigwedge_{i=0}^{n-1} \mathrm{x}^0_{i,i}$.

For goal constraints, we first define the target tableau T where $\mathrm{T}[x_{ia}]/\mathrm{T}[z_{ia}]$ returns the value of x/z matrix element in row i and column a. For a d-CNOT circuit, the following clauses encode the goal constraints.

$$\bigwedge_{i=0}^{2\,n-1}\bigwedge_{a=0}^{n-1}\begin{cases}\mathrm{x}_{i,a}^{2d+1} & \text{if } \mathrm{T}[x_{ia}]=1\\ \neg\,\mathrm{x}_{i,a}^{2d+1} & \text{if } \mathrm{T}[x_{ia}]=0\end{cases}\quad\wedge\quad\begin{cases}\mathrm{z}_{i,a}^{2d+1} & \text{if } \mathrm{T}[z_{ia}]=1\\ \neg\,\mathrm{z}_{i,a}^{2d+1} & \text{if } \mathrm{T}[z_{ia}]=0\end{cases}\tag{8}$$

4.1.5 Improvements due to Search Space Reduction

We observe that the order between two parallel entangling sequences does not matter. Let $\mathrm{cnot}_{a,b}^{k-1}$ and $\mathrm{cnot}_{a',b'}^{k}$ be two parallel gates, i.e., they do not share control or target qubits. We order them based on the control qubits, i.e., if $a > a'$, then we add $\neg\,\mathrm{cnot}_{a,b}^{k-1} \vee \neg\,\mathrm{cnot}_{a',b'}^{k}$. Note that gate ordering constraints increases the number of clauses from $O(dn^3)$ to $O(dn^4)$. Yet, these extra binary clauses reduce the search space and help the SAT solver.

In an optimal makespan, a valid solution can never repeat the state in any layer i.e., we need to consider simple paths only. Given layers k, k', we use indicator variables $\mathrm{dr}_a^{k,k'}, \mathrm{dx}_a^{k,k'}, \mathrm{dz}_a^{k,k'}$ representing which row and x-column or z-column is different, respectively. The following constraints encode that if the ith row and the ath column is chosen, then the corresponding matrix variable is different. We need to find exactly one such matrix variable which is different.

$$\bigwedge_{a=0}^{n-1}\bigwedge_{i=0}^{2n-1}((\mathrm{dr}_i^{k,k'}\wedge\mathrm{dx}_a^{k,k'})\implies(\mathrm{x}_{i,a}^{2k+2}\neq\mathrm{x}_{i,a}^{2k'+2}))\wedge((\mathrm{dr}_i^{k,k'}\wedge\mathrm{dz}_a^{k,k'})\implies(\mathrm{z}_{i,a}^{2k+2}\neq\mathrm{z}_{i,a}^{2k'+2}))$$

$$\wedge\,\mathrm{EO}(\{\mathrm{dr}_i^{k,k'}\mid 0\le i<2n\})\wedge\mathrm{EO}(\{\mathrm{dx}_i^{k,k'}\mid 0\le i<n\}\cup\{\mathrm{dz}_i^{k,k'}\mid 0\le i<n\})$$

To break any m-cycle, we instantiate the above constraints for every layer pair k, k' that differs in at most m. By default, we break all 3-cycles, since it doesn't increase the asymptotic variable and clause count.

4.1.6 Redundant Clauses and Auxiliary Variables for Propagation

Finally, we optionally allow some redundant clauses and auxiliary variables to help our backend SAT solver. In SAT encodings, there is often a trade-off between search and propagation. In [25], we observed that adding redundant clauses sometimes improve performance of SAT solving. Giving explicit constraints that trigger unit clause propagation can avoid some unnecessary search. For example, Equation 4 explicitly encodes that exactly-one CNOT variable can be true at time step t. It also implicitly encodes that exactly-one control or target variable can be true in the same time step. If a SAT solver sets a CNOT variable to true, then all control, target, and rest of the CNOT variables are propagated. However, if a SAT solver instead sets a control variable to true it does not learn any more information. By giving (redundant) exactly-one constraints on control and target variables can avoid this unnecessary search.

Similarly, using auxiliary variables can impact performance of SAT solving. In Equation 1, recall that propagation variable $\mathrm{px}_{i,a}^t$ is true then $\mathrm{x}_{i,a}^t$ is propagated else $\mathrm{x}_{i,a}^t$ is flipped. We experimented with using additional auxiliary variable $\mathrm{fx}_{i,a}^t$ to indicate the flipping instead of $\neg\,\mathrm{px}_{i,a}^t$. We add an option to use the new flipping variables for both x, z state variables as part of XOR constraints in our encoding. We provide these variations as an option, we turn them on default as they seem to help our backend SAT solver Cadical [4].

4.2 Depth optimal encoding

With minimal changes, we can encode optimal depth instead of optimal gate count. Instead of allowing a single CNOT gate, we allow multiple independent gates in each layer. In other words, we allow at most one CNOT gate at any control or target qubit. We use sequential counter AtMostOne (AMO) and AtLeastOne (ALO) constraints from Pysat [11]. We essentially replace Equation 4 with following Equation 9.

$$\bigwedge_{q=0}^{n-1} \mathrm{AMO}(\{\mathrm{cnot}_{a,b}^{k} \mid a = q \text{ or } b = q \text{ and } 0 \leq a, b < n\}) \wedge \mathrm{ALO}(\{\mathrm{cnot}_{a,b}^{k} \mid 0 \leq a, b < n\}) \wedge$$
$$\bigwedge_{q=0}^{n-1} ((\mathrm{ctrl}_{q}^{k} \implies \bigvee_{b=0}^{n-1} \mathrm{cnot}_{q,b}^{k}) \wedge (\mathrm{trg}_{q}^{k} \implies \bigvee_{a=0}^{n-1} \mathrm{cnot}_{a,q}^{k})) \tag{9}$$

Gate ordering restrictions are simpler for Depth optimal encoding. We eagerly schedule entangling gates without losing optimality. If a CNOT is applied on qubits i, j, we restrict that some entangling gate is applied on i or j qubits in the previous layer. Note that we only require $O(dn^3)$ clauses even with the additional gate ordering constraints.

$$\bigwedge_{i=0}^{n-1} \bigwedge_{j=i+1}^{n-1} \mathrm{cnot}_{i,j}^{k} \implies \left(\mathrm{ctrl}_{i}^{k-1} \vee \mathrm{ctrl}_{j}^{k-1} \vee \mathrm{trg}_{i}^{k-1} \vee \mathrm{trg}_{j}^{k-1} \right)$$

5 Experiments and Results

We implemented above encodings and their variations in version 5 of our open source tool Q-Synth[2] [27]. Given a quantum circuit in OPENQASM 2.0 [8] format, we return the optimized circuit in the same format. For *pure* Clifford circuits, we return CNOT-optimal circuits for a given metric. Given a mapped circuit and a coupling graph of a quantum platform, we optimize respecting the coupling restrictions. Similarly, we generate circuits with a possible qubit relabeling when enabled. On the other hand, we employ peephole synthesis for arbitrary circuits. From start to end of the circuit, we first group non-Clifford and Clifford gates greedily. Then we replace each obtained Clifford slice by its optimized counterpart. Note that for general circuits this does not guarantee global optimality either in cx-count or cx-depth optimization.

Search Strategies

For each Clifford optimization call, we provide two search strategies i.e., forward and backward. In forward search, we search for circuits with k CNOT gates, increasing from $k = 0$ until we find the optimal circuit. Forward search either produces an optimal circuit or a timeout. In backward search, we search for circuits with $\leq k$ CNOT gates, decreasing from the number of CNOT gates in the initial circuit until we reach UNSAT. Within the given time limit, we either synthesize an optimal circuit or report the best circuit found so far. In our initial experiments, we observed that search space reduction techniques help forward search. Thus, only in forward experiments, we add gate ordering constraints both for cx-count and cx-depth optimization. For forward cx-depth optimization, we additionally include simple path constraints as well. In Tables, we denote forward search with additional constraints as qsynth. Similarly, backward search is denoted as qsynth-b.

[2] Full code, data and logs are available at `https://doi.org/10.5281/zenodo.15575215`

Research Questions

We are interested in investigating the effectiveness and relevance of our SAT based approach. In particular, we are interested in answering the following three research questions:

- R1: Does restricting search to Clifford normal forms help with scalability?
- R2: What is the effectiveness of SAT encodings on unmapped practical benchmarks?
- R3: What is the effectiveness of SAT encodings on layout mapped practical benchmarks?

We propose the following 3 experiments to answer the above research questions.

5.1 Experiment 1: Optimal synthesis

The main difference from existing SAT based approaches is our use of normal forms. In this experiment, we investigate if normal form restricted search helps with scalability. For comparison, we consider QMAP [29] tool that implements previous exact sat-based synthesis of Clifford circuits. QMAP provides near-optimal synthesis for cx-count [22] and optimal synthesis for circuit depth [20]. In [22, 20], QMAP presented results mainly on synthesis of random Clifford circuits from given stabilizers. Random Clifford circuits are one of the standard benchmarks for comparing various approaches. Unfortunately, the random stabilizers used by QMAP in [22, 20] are not accessible. Thus, we generated new random benchmarks of 3 to 7 qubits, with 5 random instances for each qubit. For benchmark generation, we used the standard Qiskit random Clifford stabilizer function. For each such stabilizer, we further optimized with TKET [28] both with and without permutation. Our input benchmarks are essentially the best circuits synthesized by Qiskit and TKET. Any improvement on such benchmarks essentially shows the strength of exact approaches. All generated benchmarks and their scripts are available online (for reproducibility).

For cx-count optimization, we compared with near-optimal cx-count optimization by QMAP [22]. For cx-depth optimization, we compare with optimal circuit depth optimization by QMAP [20] and report cx-depth instead of circuit depth. We chose the best (default) options i.e., binary search, in QMAP v2.8 for a fair comparison. We present results for Q-Synth with forward and backward search, with- and without-permutations. For each instance, we give 3h time limit and 8GB memory limit. For Q-Synth, we use state-of-the-art SAT solver Cadical (v2.1) [4] as a backend. We use Pysat [11] for sequential counter cardinality constraints.

Results and Discussion

Table 3 presents the results of Experiment 1 with cx-count and cx-depth optimization. For cx-count optimization, we can see that the near-optimal cx-count synthesis by QMAP does not scale well and only solves 3-qubit instances. Q-Synth, on the other hand, with forward search can solve all instances up to 5-qubits and 3 out of 5 6-qubit instances. With backward search, Q-Synth even improves cx-count of 7-qubit instances but uses full 3 hour time limit to do so. In the presence of output permutation, the results are similar to without output permutation while solving fewer 6-qubit instances. For cx-depth optimization, QMAP (without output permutation) can now solve up to 5-qubit instances with optimal depth. Note that QMAP often reports suboptimal cx-depth when optimizing depth. For the hard 6-qubit instances QMAP produces intermediate results, but they are far from optimal. Q-Synth on the other hand, solves all but one 7-qubit instance with forward search. Interestingly with backward search we can improve the unsolved 7-qubit instance but not solve optimally. With output permutation, Q-Synth can optimally solve up to 6-qubit instances but not 7-qubit instances. With backward search, Q-Synth can improve 1 extra 7-qubit instance. We provide the full data in Tables 6 and 7 in Section B.

Table 3 Experiment 1: Optimizing of random Clifford circuits, n: qubits, m: optimization metric (average cx-count/cx-depth of 5 instances), t: average time if all 5 instances are completely solved; else the number of instances solved within the time limit.

	without-permutation							with-permutation				
	tket	qmap		qsynth		qsynth-b		tket	qsynth		qsynth-b	
n	m	m	t	m	t	m	t	m	m	t	m	t
(CNOT-count optimization, m = cx-count)												
3	3.6	3.6	1002.8	3.6	0.1	3.6	0.1	2.8	2.8	0.1	2.8	0.1
4	6.8	–	0/5	6.0	0.3	6.0	0.2	6.0	4.8	0.3	4.8	0.4
5	13.2	–	0/5	8.2	7.1	8.2	21.9	10.2	7.2	19.5	7.2	70.3
6	17.0	–	0/5	12.6	3/5	11.6	2/5	14.4	13.6	1/5	11.0	0/5
7	22.8	–	0/5	–	0/5	21.0	0/5	20.4	–	0/5	18.4	0/5
(CNOT-depth optimization, m = cx-depth)												
3	3.6	3.6	0.4	3.6	0.1	3.6	0.1	2.8	2.8	0.1	2.8	0.1
4	5.6	4.8	5.1	3.6	0.1	3.6	0.2	4.8	3.0	0.1	3.0	0.2
5	11.2	7.2	434.3	4.8	1.2	4.8	16.2	8.2	4.0	2.5	4.0	5.8
6	13.6	13.2	0/5	5.0	10.5	5.0	839.4	11.2	5.0	82.3	5.0	583.0
7	18.0	–	0/5	8.6	4/5	17.8	0/5	15.4	–	0/5	15.0	0/5

From the results, it is clear that restricting search to normal form indeed helps with scalability. We outperform previous SAT approaches by several orders of magnitude in both optimization metrics. Notice that restricting to normal forms achieves two goals. First, the makespan in our encodings correspond to cx-count or cx-depth. On the other hand, the makespan in previous encodings correspond to gate count and circuit depth. Note that gate count or circuit depth can be several times higher than their counterparts. Second, we avoid search through many equivalent circuits thus reducing search space. We conjecture that good performance is due to lower makespan and reduced search space.

Coming to weaknesses of our approach, we could not solve all 6-qubit instances for cx-count optimization. Existing exhaustive based approach [5] could generate a database of all 6-qubit instances. Essentially, an exhaustive approach allows more freedom in avoiding search of equivalent (or symmetric) circuits. It is unclear to us how to include such symmetry breaking in a SAT encoding. On the other hand, our cx-depth optimization can go beyond 6-qubits which is not feasible in exhaustive approach.

5.2 Experiment 2: Effect on All-to-All Practical Benchmarks

In practical benchmarks, the Clifford sub-circuits are often shallow. We are interested in investigating the effect of our SAT based optimization on such benchmarks. Here we optimize benchmarks assuming all-to-all qubit connectivity. For practical benchmarks, we consider standard variational quantum eigensolver (VQE) benchmarks as the first benchmark set. Similar to [25], we generated 12 random VQE benchmarks of 8 and 16 qubits with up to 536 cx-depth and 860 cx-count. For the second benchmark set, we used 28 instances from the standard benchmark collection of Feynman tool [2] with up to 24 qubits, 2149 cx-count, and 1878 cx-depth. We consider best cx-count reduction compiler TKET [28, 17] for a comparison. We also experiment with combined optimization of TKET and Q-Synth, we

denote it by TK+QS. In the TK+QS configuration, we first optimize each instance with TKET and then optimize with Q-Synth. In the experiment with qubit permutation, we also enable permutation in TKET. Waiting 3 hours for a single benchmark is not practical. In Experiments 2 and 3, we run Q-Synth only with forward search, with a 600s time limit.

Table 4 Experiment 2: Optimizing practical benchmarks; org: original optimization metric, ch%: change% of optimization metric (lower is better), t: time.

				without-permutation					with-permutation				
				tket	qsynth		TK+QS		tket	qsynth		TK+QS	
			org	ch%	ch%	t	ch%	t	ch%	ch%	t	ch%	t
VQE	cx-count	avg	575.8	-3.5	-3.3	321.9	-3.9	318.4	-5.8	-4.4	371.4	-6.1	364.0
		min	298	-7.7	-8.2	8.3	-8.5	8.2	-8.5	-8.9	16.4	-9.8	16.2
		max	860	-0.6	-0.6	604	-0.6	604.4	-0.7	-0.7	603.2	-0.7	603.3
VQE	cx-depth	avg	367.6	-2.9	-9.1	20.1	-9.6	19.2	-4	-10.8	72.2	-11.2	58.8
		min	233	-8.2	-16	3.1	-16.7	3.4	-8.6	-16.7	4.9	-17.1	4.8
		max	536	0	-3	60.5	-3.8	51.9	0	-3.8	363.5	-4.7	230.6
Feynman	cx-count	avg	222	-8.7	-8.4	207.2	-11.2	208.7	-9.8	-8.8	233.3	-12.4	218.4
		min	18	-28.6	-32.1	0.2	-32.1	0.2	-28.6	-32.1	0.2	-32.6	0.2
		max	2149	0	0	612.3	0	613.5	0	0	610.3	0	609.5
Feynman	cx-depth	avg	161	-7.8	-12.3	89.7	-14.4	89.5	-8.3	-12.6	104.5	-14.7	104.2
		min	16	-29.6	-48.1	0.2	-48.1	0.2	-29.6	-48.1	0.2	-48.1	0.2
		max	1878	0	1.9	604.3	0	604.2	0	4.9	603.8	1.9	603.7

Results and Discussion

Table 4 presents the results of Experiment 2. We present average, minimum, and maximum change in optimization metric and time taken. For cx-count optimization, Q-Synth produces similar reductions compared to TKET in both benchmark sets. We observed up to 8.9% and 32.1% reduction in VQE and Feynman benchmarks respectively by Q-Synth. There does exist some hard slices where Q-Synth timeouts resulting in better results by TKET. For cx-depth reduction, Q-Synth overall outperforms TKET for both benchmarks sets. We observed up to 16.7% and 48.1% reduction in VQE and Feynman benchmarks respectively by Q-Synth. We usually observe the best results with TKET+Q-Synth combination. Full tables are available in [26, Appendix B] for all variations.

From the results, we can see that SAT based approaches perform well also on practical benchmarks. Remember that our benchmarks can have up to 24 qubits, 2149 cx-count, and 1878 cx-depth. Q-Synth produces better cx-depth results up to 48.1% compared to 29.1% by TKET. Such reductions are feasible since TKET focuses on cx-count. Indeed, TKET on an average produces better cx-count reduction than Q-Synth on large instances. Since cx-count and cx-depth are both important metrics, optimizing both is helpful in practice. We see the best results by combining the strengths of TKET and Q-Synth in TK+QS combination. Our SAT based cx-depth optimization appears to nicely complement heuristic cx-count optimization. We conclude that using the combination TK+QS for reduction in both cx-count and cx-depth is feasible in a practical compilation setting.

5.3 Experiment 3: Effect on Mapped Practical Benchmarks

Layout synthesis routines in industrial compilers are fast but sub-optimal. Many Clifford optimization techniques do not support layout-aware synthesis, for example TKET. Intuitively, connectivity restrictions break assumptions made in all-to-all connectivity. SAT based approaches can elegantly encode connectivity constraints. In this experiment, we investigate if our approach can further reduce mapped practical circuits by industrial compilers. Our goal is to investigate the reduction in best synthesized benchmarks from industrial compilers. We consider TKET optimized benchmarks with qubit permutations from Experiment 2. We then mapped the benchmarks onto 54-qubit Sycamore [3], 80-qubit Rigetti [7] and 127-qubit Eagle [6] platforms using Qiskit. During Qiskit transpiling, we used the highest optimization "-O3" level. For each benchmark, we optimize with Q-Synth for both cx-count and cx-depth.

Table 5 Experiment 3: Optimizing mapped practical benchmarks; org: original optimization metric, ch%: change% of optimization metric (lower the better), t: time.

		sycamore-54			rigetti-80			eagle-127		
		org	ch%	t	org	ch%	t	org	ch%	t
VQE cx-count	avg	1253.7	-7.7	604.1	1550.2	-8.8	603.8	1842.9	-9.4	568
	min	515	-15.9	602.2	644	-19.3	602.7	731	-18.3	158.1
	max	2179	-2	607.4	2698	-2.3	605.1	3172	-2.5	607.2
VQE cx-depth	avg	818.1	-15.8	277.9	981.3	-17.2	354.2	1150.4	-18.9	344.5
	min	419	-27.4	24	514	-25.9	28.5	597	-24.8	33
	max	1377	-9.1	607.8	1579	-9.3	607.7	1886	-12.3	611.9
Feynman cx-count	avg	433.4	-16.2	264.5	497.4	-18.5	224.3	559.3	-21	219
	min	33	-24.8	0.7	38	-27.3	0.5	39	-30.3	0.6
	max	3564	-7.9	632	4173	-5.9	611.9	4809	-9	622.6
Feynman cx-depth	avg	308.5	-22.1	97.1	349.7	-22.7	119.4	386.2	-23	134.7
	min	30	-32.6	0.7	37	-32.6	0.5	37	-35.9	0.6
	max	2849	-10.7	610.8	3322	-7.9	610.7	3790	-6.6	620.9

Results and Discussion

Table 5 presents the results for all three platforms. On all platforms, Q-Synth reports good reduction for both cx-count and cx-depth metrics. For VQE benchmarks, we observed reduction of up to 19.3% cx-count and 27.4% cx-depth. For Feynman benchmarks, we observed reduction of up to 30.3% cx-count and 35.9% cx-depth. Full tables are available in [26, Appendix B] for all variations.

The separation of layout and circuit synthesis in current compilers results in suboptimal circuits. We see that SAT based approaches are effective in post layout optimization. From the full tables, we also observe that mapping and optimizing to Google's Sycamore produces the best results. We conjecture that the denser array structure of the coupling graph is the main factor. The coupling graphs of Rigetti with octagons and Eagle with heavy-hex require many extra CNOT gates to route qubits, at least on our benchmarks. Despite our good reductions, the final cx-count/cx-depth on these platforms are higher than on Sycamore. Overall, the results indicate that SAT based optimization is useful in the compilation pipeline.

6 Conclusion

In this paper, we proposed two SAT encodings that synthesize Clifford circuits with optimal CNOT count or depth. We also handle connectivity restrictions and allow qubit permutations. We implemented all encoding variations in the open source tool Q-Synth. The experiments demonstrate scalability compared to existing SAT encodings. For the first time, we are able to synthesize cx-depth optimal 7-qubit random Clifford circuits. On practical VQE and Feynman benchmarks, we overall perform better than TKET in all-to-all connectivity. We also investigated layout-aware optimization on major quantum platforms. We first optimize benchmarks with TKET, then map them using Qiskit and finally optimize the results using Q-Synth. This achieves reductions of up to 30.3% in cx-count and 35.9% in cx-depth. These results show that our SAT based approach nicely complements existing heuristic approaches.

References

1 Scott Aaronson and Daniel Gottesman. Improved simulation of stabilizer circuits. *Physical Review A*, 70(5), November 2004. doi:10.1103/physreva.70.052328.

2 Matthew Amy. Quantum circuit analysis toolkit, 2016. URL: https://github.com/meamy/feynman.

3 Frank Arute et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019. doi:10.1038/s41586-019-1666-5.

4 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.

5 Sergey Bravyi, Joseph A. Latone, and Dmitri Maslov. 6-qubit optimal clifford circuits. *npj Quantum Information*, 8(1), July 2022. doi:10.1038/s41534-022-00583-7.

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 Rigetti Computing. Rigetti computing. URL: https://www.rigetti.com.

8 Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language, 2017. arXiv:1707.03429.

9 Arianne Meijer-van de Griend and Sarah Meng Li. Dynamic qubit routing with CNOT circuit synthesis for quantum compilation. In *Proceedings 19th International Conference on Quantum Physics and Logic, QPL 2022, Wolfson College, Oxford, UK, 27 June - 1 July 2022*, volume 394 of *EPTCS*, pages 363–399, 2022. doi:10.4204/EPTCS.394.18.

10 Patrick Ettenhuber, Mads Bøttger Hansen, Pier Paolo Poier, Irfansha Shaik, Stig Elkjaer Rasmussen, Niels Kristian Madsen, Marco Majland, Frank Jensen, Lars Olsen, and Nikolaj Thomas Zinner. Calculating the energy profile of an enzymatic reaction on a quantum computer. *Journal of Chemical Theory and Computation*, 21(7):3493–3503, 2025. PMID: 40162965. doi:10.1021/acs.jctc.5c00022.

11 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.

12 Kazuo Iwama, Yahiko Kambayashi, and Shigeru Yamashita. Transformation rules for designing cnot-based quantum circuits. In *Proceedings of the 39th Design Automation Conference, DAC 2002, New Orleans, LA, USA, June 10-14, 2002*, pages 419–424. ACM, 2002. doi:10.1145/513918.514026.

13 Jiaqing Jiang, Xiaoming Sun, Shang-Hua Teng, Bujiao Wu, Kewen Wu, and Jialin Zhang. Optimal space-depth trade-off of CNOT circuits in quantum logic synthesis. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 213–229. SIAM, 2020. doi:10.1137/1.9781611975994.13.

14 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.

15 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.

16 Harsha Nagarajan, Owen Lockwood, and Carleton Coffrin. QuantumCircuitOpt: An open-source framework for provably optimal quantum circuit design. *2021 IEEE/ACM Second International Workshop on Quantum Computing Software (QCS)*, pages 55–63, 2021. doi:10.1109/QCS54837.2021.00010.

17 Paul D. Nation, Abdullah Ash Saki, Sebastian Brandhofer, Luciano Bello, Shelly Garion, Matthew Treinish, and Ali Javadi-Abhari. Benchmarking the performance of quantum computing software. *CoRR*, abs/2409.08844, 2025. arXiv:2409.08844.

18 Michael A. Nielsen and Isaac L. Chuang. *Quantum circuits*, pages 171–215. Cambridge University Press, 2010. doi:10.1017/CBO9780511976667.008.

19 K. N. Patel, I. L. Markov, and J. P. Hayes. Efficient synthesis of linear reversible circuits, 2003. arXiv:quant-ph/0302002.

20 Tom Peham, Nina Brandl, Richard Kueng, Robert Wille, and Lukas Burgholzer. Depth-optimal synthesis of Clifford circuits with SAT solvers. *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, 01:802–813, 2023. doi:10.1109/QCE57702.2023.00095.

21 Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023. doi:10.5281/zenodo.2573505.

22 Sarah Schneider, Lukas Burgholzer, and Robert Wille. A SAT encoding for optimal Clifford circuit synthesis. *2023 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 190–195, 2022. doi:10.1145/3566097.3567929.

23 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.

24 Irfansha Shaik and Jaco van de Pol. Optimal layout-aware CNOT circuit synthesis with qubit permutation. In *ECAI 2024*, volume 392 of *Frontiers in Artificial Intelligence and Applications*, pages 4207–4215. IOS Press, 2024. doi:10.3233/FAIA240993.

25 Irfansha Shaik and Jaco van de Pol. Optimal layout synthesis for deep quantum circuits on NISQ processors with 100+ qubits. In *27th International Conference on Theory and Applications of Satisfiability Testing SAT*, volume 305 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SAT.2024.26.

26 Irfansha Shaik and Jaco van de Pol. CNOT-Optimal Clifford Synthesis as SAT (full version). *CoRR*, abs/2504.00634, 2025. arXiv:2504.00634.

27 Irfansha Shaik and Jaco van de Pol. Code, benchmarks, scripts and data for "CNOT-optimal Clifford synthesis as SAT" paper at SAT25, June 2025. doi:10.5281/zenodo.15575215.

28 Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket⟩: a retargetable compiler for NISQ devices. *Quantum Science and Technology*, 6(1):014003, November 2020. doi:10.1088/2058-9565/ab8e92.

29 Robert Wille and Lukas Burgholzer. MQT QMAP. In *Proceedings of ISPD-23*. ACM, March 2023. doi:10.1145/3569052.3578928.

30 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.

A Quantum Computing Notation and Clifford Circuits

We provide a short introduction to the notations used in quantum computing. For a full textbook explanation we refer to [18]. We also shortly summarize the stabilizer formalism for Clifford circuits introduced in [1].

A qubit can be viewed as a 2-dimensional complex vector in the basis $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$. That is, an arbitrary single qubit is a pair $\binom{\alpha}{\beta} = \alpha|0\rangle + \beta|1\rangle$, for some $\alpha, \beta \in \mathbb{C}$ such that $|\alpha|^2 + |\beta|^2 = 1$. The equal superposition $|+\rangle$ is defined as $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. A system of n qubits forms a 2^n-dimensional space. For instance, the base vector $|010\rangle$ is the tensor product $|0\rangle \otimes |1\rangle \otimes |0\rangle$ of dimension 8.

A quantum operation on n qubits is a unitary transformation (i.e., linear and reversible) and can be represented by a $2^n \times 2^n$ matrix U such that $UU^\dagger = I$. Quantum circuits are used as a concise notation for quantum operations. A quantum circuit is a sequence of quantum gates, where each operation is carried out on a few qubits only.

An important class of 1-qubit operations are the Pauli matrices, which are defined as:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

One can check that $X|0\rangle = 1$ and $X|1\rangle = 0$, so X negates the basis vectors. The Pauli matrices, together with their $\pm 1, \pm i$ multiples, form the Pauli group P_1 that satisfies many nice equations, e.g. $XX = I$, $XZ = -ZX$ and $XZ = -iY$ (this can be easily checked by matrix multiplication). It can also be checked that $Z|0\rangle = |0\rangle$, i.e., Z stabilizes $|0\rangle$. Similarly, $-Z|1\rangle = |1\rangle$ and $X|+\rangle = |+\rangle$. The Pauli group P_n on n qubits simply consists of Pauli strings, i.e., a tensor product of n Pauli matrices multiplied by a constant: $i^c P_1 \otimes \cdots \otimes P_1$ (for $c \in \{0, 1, 2, 3\}$). For instance $-X \otimes Z \otimes X \in P_3$ stabilizes the 3-qubit state $|+1+\rangle$.

The Pauli group forms a small but useful fragment of quantum computing, since it represents the stabilizers of some interesting quantum states. A larger fragment can be generated by the Clifford gates, which consist of the Hadamard gate (H), the Phase gate (S) and the binary (entangling) gate $CNOT$ or CX (conditional NOT).

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \qquad CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

On basis states a and b, $CNOT(a, b)$ returns $(a, a \oplus b)$, i.e., it negates b if $a = |1\rangle$. The Hadamard gate can be used to create a superposition, i.e. $H|0\rangle = |+\rangle$, and $HH = I$. One can check that $Z = SS$, $X = HZH$ and $Y = iXZ$, so the Pauli gates can be expressed in terms of the Clifford gates. As a matter of fact the Clifford gates correspond exactly to the normalizers of P_n, that is, those unitary transformations U such that $UP_nU^\dagger = P_n$. For instance, $SXS^\dagger = Y$ and $HXH^\dagger = Z$.

The core of the stabilizer formalism [1] is that quantum circuits consisting of Clifford gates can be simulated in polynomial time by tracking the effect of the gates on the states efficiently. Instead of representing a state as a 2^n-dimensional complex vector, it is represented by n Pauli strings that together generate all stabilizers of that state. And the effect of each Clifford gate can be defined directly on those stabilizers. This information can be concisely stored in a tableau, using 2 bits for each Pauli matrix, $00 = I$, $10 = X$, $01 = Z$ and $11 = Y$. A row of $2n+1$ bits represents a single Pauli string, where the one extra bit is used to encode the sign ± 1. For technical reasons, we also keep track of n so-called destabilizers, explaining the $2n \times (2n + 1)$ Boolean matrices that are used in the SAT encoding.

B Experiment 1: Full Tables

Table 6 Experiment 1: Random Clifford cnot-count optimization, m: cx-count t: time, TO: timeout.

	without-permutation							with-permutation				
Tool:	tket	qmap		qsynth-f		qsynth-b		tket	qsynth-f		qsynth-b	
instance	m	m	t	m	t	m	t	m	m	t	m	t
3q05306	4	4	1452.5	4	0.13	4	0.06	2	2	0.1	2	0.09
3q33936	3	3	29.3	3	0.07	3	0.05	3	3	0.07	3	0.06
3q50494	4	4	1804.7	4	0.08	4	0.05	3	3	0.06	3	0.05
3q55125	3	3	77.2	3	0.06	3	0.04	3	3	0.06	3	0.05
3q99346	4	4	1650.2	4	0.08	4	0.05	3	3	0.06	3	0.05
4q05306	6	–	TO	6	0.2	6	0.12	6	5	0.44	5	0.5
4q33936	8	–	TO	6	0.38	6	0.31	8	5	0.35	5	0.71
4q50494	7	–	TO	7	0.49	7	0.38	4	4	0.1	4	0.07
4q55125	6	–	TO	6	0.18	6	0.11	6	5	0.43	5	0.33
4q99346	7	–	TO	5	0.19	5	0.25	6	5	0.43	5	0.35
5q05306	12	–	TO	8	4.74	8	14.1	11	8	34.93	8	151.2
5q33936	13	–	TO	9	11.7	9	33.44	10	8	34.35	8	126.5
5q50494	17	–	TO	8	4.03	8	36.89	11	7	22.35	7	56.6
5q55125	13	–	TO	9	13.89	9	22.23	9	7	4.49	7	9.25
5q99346	11	–	TO	7	1.05	7	2.86	10	6	1.52	6	7.88
6q05306	15	–	TO	–	TO	12	TO	15	15	TO	11	TO
6q33936	20	–	TO	11	2017.7	12	TO	14	14	TO	11	TO
6q50494	17	–	TO	11	964.3	11	4525.9	14	14	TO	12	TO
6q55125	18	–	TO	11	1060.2	11	5059.7	14	10	6452.9	10	TO
6q99346	15	–	TO	–	TO	12	TO	15	–	TO	11	TO
7q05306	25	–	TO	–	TO	–	TO	24	–	TO	19	TO
7q33936	24	–	TO	–	TO	–	TO	20	–	TO	17	TO
7q50494	22	–	TO	–	TO	18	TO	20	–	TO	–	TO
7q55125	22	–	TO	–	TO	17	TO	18	–	TO	17	TO
7q99346	21	–	TO	–	TO	–	TO	20	–	TO	19	TO

Table 7 Experiment 1: Random Clifford cnot-depth optimization, m: cx-depth, t: time, TO: timeout.

	without-permutation							with-permutation				
Tool:	tket	qmap		qsynth-f		qsynth-b		tket	qsynth-f		qsynth-b	
instance	m	m	t	m	t	m	t	m	m	t	m	t
3q05306	4	4	0.4	4	0.13	4	0.06	2	2	0.1	2	0.09
3q33936	3	3	0.39	3	0.07	3	0.05	3	3	0.07	3	0.05
3q50494	4	4	0.25	4	0.08	4	0.05	3	3	0.06	3	0.04
3q55125	3	3	0.42	3	0.06	3	0.04	3	3	0.06	3	0.05
3q99346	4	4	0.34	4	0.08	4	0.05	3	3	0.06	3	0.05
4q05306	4	5	4.0	4	0.11	4	0.07	4	3	0.12	3	0.1
4q33936	7	5	4.27	4	0.16	4	0.34	7	3	0.14	3	0.57
4q50494	6	6	6.85	4	0.17	4	0.17	3	3	0.07	3	0.05
4q55125	4	4	2.53	3	0.1	3	0.09	4	3	0.12	3	0.1
4q99346	7	4	8.08	3	0.11	3	0.44	6	3	0.21	3	0.36
5q05306	10	8	437.1	5	1.64	5	9.68	8	4	2.57	4	4.56
5q33936	10	7	533.8	5	0.91	5	6.3	8	4	2.81	4	7.0
5q50494	15	7	626.2	5	2.25	5	40.49	9	4	5.71	4	6.98
5q55125	12	8	347.7	5	0.91	5	21.45	8	4	1.27	4	4.98
5q99346	9	6	226.7	4	0.39	4	3.25	8	4	0.37	4	5.41
6q05306	11	–	TO	5	23.59	5	299.27	11	5	168.09	5	607.01
6q33936	16	11	TO	5	6.39	5	994.98	10	5	60.75	5	379.15
6q50494	14	14	TO	5	1.86	5	741.38	11	5	64.86	5	357.08
6q55125	15	17	TO	5	3.03	5	638.7	12	5	35.44	5	587.76
6q99346	12	11	TO	5	17.66	5	1522.43	12	5	82.35	5	984.03
7q05306	19	–	TO	–	TO	18	TO	18	–	TO	–	TO
7q33936	19	–	TO	6	4151.7	–	TO	16	–	TO	–	TO
7q50494	18	–	TO	6	7347.0	–	TO	15	–	TO	–	TO
7q55125	19	–	TO	6	1485.5	–	TO	14	–	TO	12	TO
7q99346	15	–	TO	6	241.0	–	TO	14	–	TO	–	TO

Reencoding Unique Literal Clauses

Aeacus Sheng
Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA, USA

Joseph E. Reeves
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Marijn J. H. Heule
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

We present a lightweight reencoding technique that augments propositional formulas containing implicit or explicit exactly-one constraints with auxiliary variables derived from the order encoding. Our approach is based on the observation that many formulas contain clauses where each literal appears only in that clause, and that these *unique literal clauses* can be replaced by the corresponding sequential counter encoding of exactly-one constraints, which introduces the same variables as the order encoding. We implemented the reencoding in the state-of-the-art SAT solver CaDiCaL with support for proof logging and solution reconstruction. Experiments on SAT Competition benchmarks demonstrate that our technique enables solving dozens of additional formulas. We found that shuffling a formula before reencoding harms performance. To mitigate this issue, we introduce a method that sorts literals within clauses based on the formula structure before applying our reencoding. The same technique also predicts whether reencoding is likely to yield improvements.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Satisfiability solving, auxiliary variables, graph coloring

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.29

Supplementary Material *Software*: `https://github.com/jreeves3/ulc-cadical` [28]
archived at `swh:1:dir:615c90061cfad33b59249832fb0043defef84419`

Funding Supported by the National Science Foundation (NSF) under grant CCF-2415773.
Joseph E. Reeves: Supported in part by a fellowship award under contract FA9550-21-F-0003 through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR) and the Army Research Office (ARO).

Acknowledgements We thank Sonya Simkin for her contributions during the early stages of this research.

1 Introduction

When encoding combinatorial problems into propositional logic, it is often necessary to encode the selection of an object from a set. The *direct encoding* handles this with one propositional variable per object, which is `true` if that object is selected, along with a clause enforcing that one of these variables must be `true`. A potential disadvantage of the direct encoding is that it does not contain variables to facilitate reasoning about multiple objects simultaneously. For example, it could be beneficial to have a variable that indicates whether an object in the first half of the set is selected (using some ordering). If this variable is assigned to `false`, then the solver knows that the selected object needs to be in the second half. Other encodings, such as the *sequential counter encoding* [29] and the *order encoding* [31], include such variables. This paper presents a lightweight reencoding method that incorporates these variables by replacing specific clauses to enhance solver performance across various benchmarks.


© Aeacus Sheng, Joseph E. Reeves, and Marijn J. H. Heule;
licensed under Creative Commons License CC-BY 4.0



It is well-known that auxiliary variables in proofs could reduce their size exponentially [32]. The impact of auxiliary variables in formulas has been less studied. However, the variables that we introduce are used in the order encoding, which improves solver performance on a range of problems [1,8,10,24,30]. Moreover, the award-winning CSP solver Sugar employs the order encoding [31]. Introducing new variables tends to reduce propagation speed, thereby offsetting potential gains. Hence, ideally, the order variables should only be introduced when they provide a tangible benefit.

While automatically reencoding problems can be challenging, direct encodings for object selection constraints are typically easy to detect: the literal that expresses that an object is selected occurs only once in the formula. We exploit this property by searching for unique literal clauses (ULCs), that is, clauses containing only literals that do not appear elsewhere in the formula. We will show later that ULCs express either implicit or explicit ExactlyOne constraints. Whenever we find a ULC, we can reencode the involved part of the formula to turn the direct encoding into the sequential counter or order encoding.

To make this transformation effective, two additional steps are required. First, many formulas contain pairs of ULCs with complementary literals. We resolve these pairs to obtain longer ULCs and to simplify the reencoding process. Second, it is critical to *sort* the literals in ULCs in a way that respects the structure of the formula. We introduce the concept of *aligning* ULCs so that the newly introduced order variables have a meaningful interpretation. We observed that when ULCs cannot be aligned, the reencoding can be harmful.

Existing work on reencoding propositional formulas focuses on reducing the size of the formula by introducing new variables [13, 20, 26]. In contrast, our reencoding increases the size of the formula, although it may be reduced because certain clauses become redundant. The closest related work is by Manthey and Steinke [21], who also propose transforming the direct encoding into the sequential counter encoding. They detect direct encodings of explicit ExactlyOne constraints rather than ULCs to guarantee a reduction in formula size. Due to a small-scale evaluation that showed limited impact on performance, it is unclear whether their approach is effective.

Experiments show that our reencoding strategy, implemented in the state-of-the-art solver CaDiCaL [5], solves substantially more formulas from the 2022 SAT Competition Anniversary Track. The introduction of meaningful order variables is the key component of our technique.

2 Encodings

In this section, we describe the three encodings of the ExactlyOne constraint that are relevant to this paper. The constraint ExactlyOne $(\ell_1, \ldots, \ell_k)$ requires that exactly one of the literals $\ell_1, \ldots, \ell_k$ is assigned to true.

2.1 Direct Encoding

The direct encoding of ExactlyOne $(\ell_1, \ldots, \ell_k)$ splits the constraint into AtLeastOne $(\ell_1, \ldots, \ell_k)$ and AtMostOne $(\ell_1, \ldots, \ell_k)$, where the former is just a clause and the latter blocks any pair of these literals from both being true.:

$$(\ell_1 \vee \cdots \vee \ell_k) \wedge \bigwedge_{1 \leq i < j \leq k} (\overline{\ell}_i \vee \overline{\ell}_j)$$

Note that the number of binary clauses is quadratic in k.

If the literals $\ell_1, \ldots, \ell_k$ do not occur in other clauses, the binary clauses can be omitted while preserving satisfiability. In that case, we call it an implicit ExactlyOne constraint. If all binary clauses are present, we call it an explicit ExactlyOne constraint.

2.2 Sequential Counter Encoding

The sequential counter encoding by Sinz [29] facilitates a compact representation of cardinality constraints. The method works for arbitrary cardinality constraints and uses auxiliary variables $s_{i,j}$ expressing that out of the first i literals, exactly j are true. Since we focus on the ExactlyOne constraint, we only use variables with $j = 1$ and therefore drop the second index. For this restricted case, the encoding uses the following definitions: $s_i \leftrightarrow (s_{i-1} \vee \ell_i)$ for $1 < i < k$ and $s_1 \leftrightarrow \ell_1$. Additionally, to enforce that at most one can be true, the clauses $\overline{s}_{i-1} \vee \overline{\ell}_i$ are included for $1 < i < k$. Finally, at least one of them must be true, which is encoded as follows: $\overline{s}_{k-1} \to \ell_k$.

▶ **Example 1.** Consider the constraint ExactlyOne (ℓ_1, ℓ_2, ℓ_3). The sequential counter encoding uses the following clauses:

$$\underbrace{(s_1 \vee \overline{\ell}_1) \wedge (\overline{s}_1 \vee \ell_1)}_{s_1 \leftrightarrow \ell_1} \wedge \underbrace{(\overline{s}_2 \vee s_1 \vee \ell_2) \wedge (s_2 \vee \overline{s}_1) \wedge (s_2 \vee \overline{\ell}_2)}_{s_i \leftrightarrow (s_{i-1} \vee \ell_i)} \wedge \underbrace{(\overline{s}_1 \vee \overline{\ell}_2) \wedge (\overline{s}_2 \vee \overline{\ell}_3)}_{\overline{s}_{i-1} \vee \overline{\ell}_i} \wedge \underbrace{(s_2 \vee \ell_3)}_{\overline{s}_{k-1} \to \ell_k}$$

For our reencoding technique, we will replace a clause $(\ell_1 \vee \cdots \vee \ell_k)$ if it represents an implicit or explicit ExactlyOne constraint with the sequential counter encoding shown above. Note that the reencoding replaces one clause by roughly $3k$ clauses. However, clauses of the form $(\overline{\ell}_i \vee \overline{\ell}_j)$ become redundant and will therefore be removed if present.

2.3 Order Encoding

The order encoding [31] uses order variables $o_{\leq i}$ with $1 < i < k$ to express if one of $\ell_1, \ldots, \ell_i$ is `true`. To state that ℓ_i is `true` using only order variables, $o_{\leq i}$ must be `true`, while $o_{\leq i-1}$ must be `false`. The case for ℓ_1 is special: ℓ_1 is `true` if and only if $o_{\leq 1}$ is `true`. When encoding ExactlyOne, the case ℓ_k is also special: ℓ_k is `true` if and only if $o_{\leq k+1}$ is `false`. The order encoding uses the following clauses $\overline{o}_{\leq i} \vee o_{\leq i+1}$ stating that if one of the first i literals is `true`, then one of the first $i + 1$ is `true`.

The order encoding eliminates the x_i variables by replacing all their occurrences with $o_{\leq i}$ variables. More specifically,

- all literals ℓ_1, $\overline{\ell}_1$, ℓ_k, and $\overline{\ell}_k$ are replaced by $o_{\leq 1}$, $\overline{o}_{\leq 1}$, $\overline{o}_{\leq k-1}$, and $o_{\leq k-1}$, respectively.
- all literals $\overline{\ell}_i$ with $1 < i < k$ are replaced by $o_{\leq i-1} \vee \overline{o}_{\leq i}$.
- all clauses with literals ℓ_i with $1 < i < k$ are duplicated with one copy using $\overline{o}_{\leq i-1}$ and the other copy using $o_{\leq i}$ instead of ℓ_i.

▶ **Example 2.** For the order encoding of a graph-coloring problem with graph $G = (V, E)$ and k colors, let $v_{\leq i}$ denote whether vertex v has color at most i. The clauses are:

- For each $v \in V$: $(\overline{v}_{\leq 1} \vee v_{\leq 2}) \wedge (\overline{v}_{\leq 2} \vee v_{\leq 3}) \wedge \cdots \wedge (\overline{v}_{\leq k-2} \vee v_{\leq k-1})$
- For each $(u, v) \in E$: $(\overline{u}_{\leq 1} \vee \overline{v}_{\leq 1}) \wedge (u_{\leq 1} \vee \overline{u}_{\leq 2} \vee v_{\leq 1} \vee \overline{v}_{\leq 2}) \wedge \cdots \wedge (u_{\leq k-1} \vee v_{\leq k-1})$

Note that for larger k, the clauses of length four will outnumber the binary clauses.

Transforming the direct encoding of ExactlyOne into the sequential counter encoding is simple: replace the clauses that encode the constraint. The remaining part of the formula is unaltered. Now, observe that transforming from the sequential counter encoding into the order encoding is also easy: eliminate the original variables using variable elimination (also known as DP resolution). To see this, note that the s_i variables of the sequential counter encoding map to the $o_{\leq i}$ variables of the order encoding.

3 Reencoding

In this section, we describe the theoretical foundation of our reencoding method. Our method focuses on introducing meaningful auxiliary variables that help the solver. We aim to achieve this by allowing the solver to reason about multiple objects simultaneously. We first describe the pattern that we use to detect a reencoding opportunity. Afterward, we merge these patterns when possible to increase effectiveness. Finally, we sort the literals to provide more meaningful variables.

3.1 Unique Literal Clauses

The key building block of our reencoding method is the notion of unique literal clauses, which is defined below.

▶ **Definition 3.** *A clause C is a* unique literal clause (ULC) *with respect to a formula Γ if none of the literals $\ell \in C$ occurs in $\Gamma \setminus \{C\}$.*

Given a formula Γ, the set of ULCs can be computed in linear time in the size of Γ by first computing which literals occur only once in Γ and then determining which clauses only contain such literals.

▶ **Lemma 4.** *Given a formula Γ and a ULC $C \in \Gamma$, if Γ is satisfiable, then it can be satisfied with exactly one literal in C assigned to true.*

Proof. Let τ be a satisfying assignment of Γ, then τ must satisfy at least one literal in C. Then, we flip assignments to variables in C until there is only one satisfied literal in it. Since each literal in a ULC is unique in Γ, each flip cannot falsify any satisfied clause. ◀

An alternative argument showing Lemma 4 is that for a ULC $(\ell_1 \vee \cdots \vee \ell_k)$, the formula can be extended with all binary clauses $(\overline{\ell}_i \vee \overline{\ell}_j)$ for $1 \leq i < j \leq k$ using blocked clause addition [19]. These binary clauses enforce that at most one literal of the ULC can be `true`.

To transform the direct encoding into the sequential counter encoding, ULC $(\ell_1 \vee \cdots \vee \ell_k)$ and all present binary clauses $(\overline{\ell}_i \vee \overline{\ell}_j)$ for $1 \leq i < j \leq k$ are replaced by the definitions of the auxiliary variables, $s_1 \leftrightarrow \ell_1$ and $s_i \leftrightarrow (s_{i-1} \vee \ell_i)$ with $1 < i < k$, together with the at-most-one clauses $(\overline{s}_{i-1} \vee \overline{\ell}_i)$ with $1 < i < k$ and the at-least-one clause $(s_{k-1} \vee \ell_k)$.

Alternatively, one could reencode the direct encoding into the order encoding. This would eliminate the ULC $(\ell_1 \vee \cdots \vee \ell_k)$ and replace the literals $\overline{\ell}_1, \ldots, \overline{\ell}_k$ by literals of new variables $o_{\leq 1}, \ldots, o_{\leq k-1}$ that are inspired by the order encoding:

- $\overline{\ell}_1 \equiv \overline{o}_{\leq 1}$
- $\overline{\ell}_i \equiv o_{\leq i-1} \vee \overline{o}_{\leq i}$ for $1 < i < k$
- $\overline{\ell}_k \equiv o_{\leq k-1}$

Again, binary clauses $(\overline{\ell}_i \vee \overline{\ell}_j)$ in Γ would become redundant after the reencoding and should be discarded. An optional extension of the reencoding is the addition of the order clauses $(\overline{o}_{\leq i-1} \vee o_{\leq i})$ for $1 < i < k$. Typically, adding these order clauses improves performance.

3.2 Clashing ULCs

To maximize the effectiveness of the reencoding, the ULCs should be as large as possible. Also, we want to reencode *all* ULCs above a size threshold. Since original variables occur positively and negatively in the definition clauses of the sequential counter encoding, reencoding one ULC may break the ULC property of another clause. We achieve both objectives by resolving "clashing" ULCs.

▶ **Definition 5.** *Two unique literal clauses C and D are said to* clash *(on a literal ℓ) if there exist $\ell \in C$ and $\overline{\ell} \in D$. We also say that they clash on the variable v underlying ℓ.*

▶ **Definition 6.** *A formula Γ is* clash-free *if none of its ULCs clash.*

We can extend Lemma 4 to show that if a satisfiable formula is clash-free, then every ULC can be satisfied on a single literal.

▶ **Lemma 7.** *Let Γ be a clash-free formula. If Γ is satisfiable, then it can be satisfied with exactly one literal in each ULC assigned to true.*

Proof. Pick a satisfying assignment τ of Γ. For each ULC, augment τ as in the proof of Lemma 4. Since the ULCs do not clash, flipping assignments to literals in them does not falsify other ULCs. ◀

Note that resolving two clashing ULCs either results in a tautology (if there are multiple clashing pairs) or a new ULC. So, if a formula has a pair of clashing ULCs, we can either just eliminate the pair (if the resolvent is a tautology) or replace the pair with a new one while eliminating a variable.

▶ **Lemma 8.** *Eliminating all clashing variables among the ULCs in a formula Γ is confluent, and the unique fixpoint is a clash-free formula.*

Proof sketch. There exists a unique partition of the clashing ULCs such that for each partition, it holds that i) each ULC does not clash with any ULC in another partition, ii) clashing ULCs are in the same partition, and iii) each ULC clashes with at least one other ULC in its partition. If a partition with n ULCs has more than $n-1$ clashing pairs, then variable elimination will remove all ULCs in it. On the other hand, if a partition with n ULCs has $n-1$ clashing pairs, then variable elimination (which in this case is just resolution) will result in a unique ULC consisting of all the literals that are not clashing. Because clashes do not occur between partitions, the resulting formula is clash-free. ◀

Our reencoding method begins by making the input formula clash-free through variable elimination, as described in the proof. Apart from eliminating clashes, variable elimination increases the size of ULCs. The larger the ULC, the more likely the auxiliary variables will be beneficial, as they allow the solvers to reason about more objects at the same time.

3.3 Aligning ULCs

In this section, we assume that a formula is clash-free. Any formula that is not clash-free can be easily turned into one by applying the method described in Section 3.2. While all literals are symmetric in the direct encoding, their order matters when performing our reencoding. Since ULCs are clauses, the literals can be permuted in any way to optimize performance. We consider ULCs as sequences of literals in this section. The following example shows that the order of literals in ULCs could impact the effectiveness of our reencoding.

▶ **Example 9.** Consider a graph-coloring problem with k colors. Let the graph contain a 4-clique among the vertices u, v, w, x. The direct encoding contains the ULCs $(u_1 \vee \ldots \vee u_k)$, $(v_1 \vee \ldots \vee v_k)$, $(w_1 \vee \ldots \vee w_k)$, $(x_1 \vee \ldots \vee x_k)$. The variables u_i denote that vertex u has color i. Reencoding these ULCs using the natural ordering of the literals creates auxiliary variables that can be interpreted as $u_{\leq i}$, $v_{\leq i}$, $w_{\leq i}$, $x_{\leq i}$ with $i \in \{1, \ldots, k-1\}$.

Now, let's consider the assignment that makes $u_{\leq 3}$, $v_{\leq 3}$, $w_{\leq 3}$, $x_{\leq 3}$ false, which makes variables $u_1, u_2, u_3, v_1, v_2, v_3, w_1, w_2, w_3, x_1, x_2, x_3$ false by unit propagation. Using $k = 6$, a solver can quickly determine that this assignment cannot be extended to a satisfying

assignment as at least four distinct colors are required for the vertices u, v, w, x. Hence, the solver can learn the clause $(u_{\leq 3} \vee v_{\leq 3} \vee w_{\leq 3} \vee x_{\leq 3})$. Note that a similar clause with original variables would be much longer: $(u_1 \vee u_2 \vee u_3 \vee v_1 \vee v_2 \vee v_3 \vee w_1 \vee w_2 \vee w_3 \vee x_1 \vee x_2 \vee x_3)$.

Alternatively, consider the case of a different order of literals in ULCs. For example, using $k = 6$, they could be $(u_4 \vee u_2 \vee u_5 \vee u_1 \vee u_6 \vee u_3)$, $(v_2 \vee v_1 \vee v_4 \vee v_3 \vee v_6 \vee v_5)$, $(w_1 \vee w_3 \vee w_2 \vee w_6 \vee w_5 \vee w_4)$, $(x_6 \vee x_2 \vee x_4 \vee x_3 \vee x_1 \vee x_5)$. After reencoding, the auxiliary variables have no meaning. Moreover, assigning the corresponding four auxiliary variables to false will falsify the first three literals in each clause by unit propagation. Now it is still possible to assign the vertices u, v, w, x to four distinct colors, so no (short) clause can be learned. Therefore, order of literals in ULCs is crucial for learning the short clause.

To allow the solver to make good use of the auxiliary variables, we want to order the literals in the ULCs so that the solver can learn more useful short clauses like the one above. This motivates our definition of *aligned* ULCs.

▶ **Definition 10.** *Let Γ be a formula whose unique literal clauses are $\{C_1, \ldots, C_m\}$. We say that Γ is aligned if for every pair of binary clauses $(\overline{a} \vee \overline{b}), (\overline{c} \vee \overline{d}) \in \Gamma$ with $a, c \in C_i$ and $b, d \in C_j$, it holds that literal a occurs before c in C_i if and only if b occurs before d in C_j.*

As we will show in the experimental evaluation, aligning ULCs increases the effectiveness of our reencoding method. Moreover, when formulas are not aligned or unalignable, the reencoding can become ineffective or even harmful to performance.

▶ **Definition 11.** *A formula is* independent *if it is trivially aligned, that is, it does not contain any binary clauses between its ULCs. Suppose Γ is not independent. Then it is* alignable *if there exists a permutation of the literals in its ULCs such that it becomes aligned, and* unalignable *otherwise.*

Whether a formula is alignable is computable in linear time in the size of the formula. We will discuss this in more detail in Section 4. Since we can efficiently determine whether a formula is alignable, we can run this check first and only reencode alignable formulas.

3.4 Exclusive Literal Clauses

Manthey and Steinke discuss transforming the direct encoding of ExactlyOne constraints [21]. They look for the following pattern: a clause $(\ell_1 \vee \cdots \vee \ell_k)$ together with all binary clauses $(\overline{\ell}_i \vee \overline{\ell}_j)$ with $1 \leq i < j \leq k$. Note that they do not require literals $\ell_1, \ldots, \ell_k$ to occur uniquely in the formula, in contrast to ULCs. On the other hand, ULCs do not rely on the presence of binary clauses $(\overline{\ell}_i \vee \overline{\ell}_j)$. Thus, each method targets a different pattern. The concept of exclusive literal clauses generalizes both methods.

▶ **Definition 12.** *Given a formula Γ and a clause C. Let $U(C, \Gamma)$ be the set of literals in C that do not occur in $\Gamma \setminus \{C\}$ and $X(C, \Gamma) = C \setminus U(C, \Gamma)$. A clause C is an* exclusive literal clause (XLC) *with respect to Γ if $(\overline{\ell} \vee \overline{\ell'}) \in \Gamma$ for all $\ell, \ell' \in X(C, \Gamma)$.*

So, a ULC is an XLC in which all literals are unique, while Manthey and Steinke detect XLCs with no unique literals. For the purpose of experiments, it is convenient to consider XLCs that are not ULCs separately, and we call them *proper* XLCs.

Similar to a ULC, an XLC can be considered an implicit ExactlyOne constraint as all missing binary clauses can be added via blocked clause addition. The notion of a pair of clashing XLCs is the same as for ULCs: the pair contains complementary literals. In contrast to ULCs, it is not the case that resolving two XLCs always results in a new XLC.

▶ **Example 13.** Consider the formula $\Gamma = (x \vee z) \wedge (x \vee u) \wedge (y \vee \overline{u}) \wedge (y \vee z) \wedge (\overline{z})$. The clauses $(x \vee u)$ and $(y \vee \overline{u})$ are clashing XLCs with u and $\overline{u}$ being unique literals. Resolving these clauses results in $(x \vee y)$, which is not an XLC. Moreover, it cannot be turned into XLC, because adding the missing binary clause $(\overline{x} \vee \overline{y})$ does not preserve satisfiability.

The above issue arises when resolving on unique literals. When resolution on a pair of clashing XLCs is restricted to non-unique literals, then the resulting clause will be XLC. However, doing so can be very costly in practice, as many clauses may contain the resolution variable. We observed that there are many formulas from SAT Competitions for which resolving away all pairs of clashing XLCs resulted in an enormous blowup of the formula. For these reasons, we will not resolve clashing XLCs in our experiments, and we apply only the sequential counter encoding in our experiments, without trying to eliminate into the order encoding. Definitions regarding alignment extend straightforwardly to formulas with XLCs.

3.5 Proof Production

Modern SAT solvers support proof logging, so the results of each run can be verified [14]. All techniques used in our reencoding technique can be compactly expressed in the DRAT proof system [33]. The first step of our method, described in Section 3.2, makes the formula clash-free. Each step either resolves two ULCs or eliminates a pure literal. Since clauses are sets of literals, one can shuffle the literals around without any proof logging, so aligning is not explicit in the proof.

The reencoding consists of four stages for the sequential counter encoding and an additional fifth stage for the order encoding. We will explain the proof logging procedure for reencoding ULCs. The procedure for XLC is similar but somewhat more complicated.

1. Make the ExactlyOne constraint explicit. Before reencoding a ULC $(\ell_1 \vee \cdots \vee \ell_k)$, we need all the binary clauses $(\overline{\ell}_i \vee \overline{\ell}_j)$ with $1 \leq i < j \leq k$ to be present in the formula. Each of the missing ones can be added using blocked clause addition.
2. Add the definitions. Next, we add the definitions of the sequential counter encoding: $s_1 \leftrightarrow \ell_1$ and $s_i \leftrightarrow (s_{i-1} \vee \ell_i)$ for $1 < i < k$. The clauses corresponding to these definitions can be added using blocked clause addition by adding them in increasing order of i.
3. Express AtMostOne and AtLeastOne. Now, we need to express that exactly one of the $\ell_1, \ldots, \ell_k$ literals can be `true` using the new definitions. For the at-most-one part, the clauses $(\overline{s}_{i-1} \vee \overline{\ell}_i)$ are included for $1 < i < k$, while for the at-least-one part we add the clause $(s_{k-1} \vee \ell_k)$. All these clauses have the reverse unit propagation (RUP) property w.r.t. the formula and can be added in arbitrary order [11].
4. Remove the original clauses. At this point, we no longer need the original clauses, including the added binary clauses, and we remove them. After the prior step, these clauses become RUP, so deleting them will not introduce additional satisfying assignments.
5. Eliminate the original variables. In case of a transformation to the order encoding, we need to apply variable elimination on all the original variables in the direct encoding. When reencoding ULCs, this step will decrease the number of clauses. However, when reencoding XLCs, this step could substantially increase the number of clauses.

The main difference between reencoding ULCs and proper XLCs is the need for careful accounting in the latter case. It may not be allowed to remove some of the original binary clauses.

4 Implementation

We implement XLC reencoding inside the state-of-the-art solver CaDiCaL [5]. Reencoding is performed as preprocessing immediately before entering the CDCL loop. By default, no preprocessing is performed in CaDiCaL; however, the order of literals within clauses may be changed by unit propagation during parsing and by propagation during lucky search [27]. We perform reencoding after lucky search so that trivial problems are still solved quickly.

1. Classify XLCs
2. Perform variable elimination with clashing ULCs
3. Optionally, align all XLCs
4. Add sequential counter encoding
5. Remove tautologies and clauses that became redundant
6. Optionally, perform variable elimination to create the order encoding

First, we classify clauses as XLCs. Each clause in the formula is considered during this stage, and the clause size cutoff is only enforced during reencoding in step 4. We compute occurrence counts for each literal in the formula Γ, then classify clauses as an ULC, XLC, or neither. In the first check, the clause is considered a ULC if $\mathsf{occ}(\ell, \Gamma) = 1$ for all $\ell \in C$. If this check fails, then in the second check, the clause is classified as an XLC if all of the binary clauses between the negations of all pairs of non-unique literals in C exist in the formula. To perform this check efficiently, we use a lookup table that stores the binary clauses for each literal. Finally, if neither check succeeds, the clause is neither a ULC or XLC and therefore is not a candidate for reencoding.

Second, we perform resolution on ULC clashes. For each literal ℓ in a ULC with $\mathsf{occ}(\ell, \Gamma) = 1$ and $\mathsf{occ}(\overline{\ell}, \Gamma) = 1$ we mark the literal as clashing only if $\overline{\ell}$ occurs in a ULC, and store a clause lookup table $L(\ell)$ and $L(\overline{\ell})$ pointing to the clauses ℓ and $\overline{\ell}$ occur in. We then process each ULC with a clashing literal ℓ, resolving both $L(\ell)$ and $L(\overline{\ell})$. We delete $L(\ell)$ and $L(\overline{\ell})$, eliminating the variable $\mathsf{var}(\ell)$ from the formula. The resolvent may take two forms: (1) the resolvent is a tautology in which case we delete it; (2) the resolvent is a clause with at least two literals in which case we add it to the formula and mark it as a ULC. The resolvent cannot be empty because we propagate unit clauses in a separate procedure, and the resolvent cannot be unit because we only resolve two ULCs. We must also update the lookup table for any clashing literals contained in the resolvent because it may also clash with an existing ULC. We do not perform variable elimination on clashing XLCs that are not ULCs to avoid a blowup in the formula size.

Third, we optionally align XLCs. The alignment algorithm is described in more detail below in Section 4.1. Alternatively, the user may specify the natural or a shuffled ordering for literals within XLCs. The natural ordering is the ordering of literals by variable names (ids in the DIMACS format). This ordering of literals within each XLC will determine which clauses are generated in the sequential counter encoding. This step is crucial, even without alignment enabled because propagations during the parsing and lucky search can change the ordering of literals within a clause since newly watched literals are swapped to the front.

Before the fourth step of reencoding we perform several checks. We encode all XLCs of size 5 and up, preventing the reencoding of small XLCs that will have little impact on the formula. If the clause is an XLC and not a ULC, we must check if the XLC is clashing on a unique literal with another XLC. If so, we can only encode one of the XLCs, and choose to encode the first one that is seen. Finally, before adding the sequential counter encoding, we may first need to add clauses to the proof to ensure that the sequential encounter encoding

can be derived. This is only necessary if the binary clauses described in Step 1 of Section 3.5 do not exist in the formula, in which case we can add them or add them directly or can add a more compact set of clauses [12].

Fourth, we reencode each XLC by adding the sequential counter encoding to the formula.

We generate new variables for the encoding starting from one plus the maximum variable in the formula. This strategy of adding new variables is not viable in the incremental setting when new variables may be added through the API between solver calls, but this limitation can be resolved and is not the focus of this work.

Fifth, we remove tautologies and redundant binary clauses. We use a time stamp to make this procedure efficient by marking the negation of literals occurring together in sequential counter encodings. For example, binary clauses formed of two negated literals from the same XLC can be removed after encoding the sequential counter, and this can be done by checking if the time stamp of both literals in the binary clause is the same. However, if a literal appears in multiple XLCs, it will have an updated time stamp. In this case, we must delete the binary clauses using a lookup table. Further, any binary clause added to the proof before adding the sequential counter encoding is deleted. Finally, we delete all reencoded XLCs from the formula.

Sixth, we optionally perform variable elimination on the literals occurring in reencoded ULCs. This will transform the sequential counter encoding into the order encoding. Variable elimination for all ULCs is done in a single pass over the clauses in the formula by using a variable substitution mapping. For each clause containing a negated literal from a ULC, literals in that clause are mapped to their replacements in a new clause, and the old clause is deleted. Note that some negated literals from ULCs will be replaced by two literals from the sequential counter encoding, and this could turn many binary clauses into clauses of length three or four, potentially slowing down solver propagations. Then, clauses from the sequential counter encodings that contain ULC literals are deleted. While we perform variable elimination by hand, it is possible that the solver would have eliminated some or all of these literals in the bounded variable elimination (BVE) inprocessing steps. On the other hand, the solver may eliminate variables from the sequential counter encoding.

All of our clause additions and deletions use the DRAT proof API inside CaDiCaL. This uses the extension stack for RAT deletions to reconstruct solutions for the original set of variables upon solve completion. CaDiCaL supports proof logging in LRAT, and in future work, we plan to support LRAT proof production.

The reencoding preprocessor is called once before solving, and checks every clause in the formulas for the XLC criteria. In addition, this could be performed as inprocessing [17]. The solver cannot learn a ULC directly through conflict analysis, because prior to clause learning, the literals in the learned clause must have been pure and therefore fixed. However, techniques like subsumption, probing, or variable elimination that remove literals from clauses and delete clauses may produce ULCs. To detect ULCs in inprocessing, it would suffice to check the occurrence counts for irredundant clauses. If an irredundant clause meets the ULC criteria, we delete all redundant clauses containing the literals from the ULC. Then, this clause could be reencoded as a ULC. Also, the solver could learn binary clauses that allow some other clause to meet the XLC criteria. We do not implement these inprocessing procedures because the preprocessing approach already performs well on a wide range of problems containing ULCs, and does not improve performance much on problems containing XLCs.

4.1 Alignment of XLCs

This section presents a procedure for aligning the XLCs within a formula. This amounts to assigning each literal occurring in an XLC an integer alignment value and then using these values to sort the literals within the XLCs.

To perform alignment, we first construct a graph G where nodes are literals and the graph contains an undirected edge between two literals ℓ and ℓ' iff $(\overline{\ell}' \vee \overline{\ell}) \in \Gamma$ and both ℓ and ℓ' occur in distinct XLCs. Next, we sort the XLCs from largest to smallest, sort the literals within each individual XLC by the natural ordering, and initialize the alignment of all literals to 0. Then, for each $C \in XLCs$ and for each $\ell \in C$: if ℓ is not aligned, ℓ is assigned $alignment + 1$ and all unaligned literals in the connected component of G containing ℓ are aligned with $alignment + 1$. We increment $alignment$, then continue to the next literal. After aligning all literals in C, the clause is sorted based on the alignment values. If any pair of literals in C shares the same alignment value, the problem is said to be unalignable. This will happen if two literals from an XLC occur in the same connected component in G. In this case, the literals are sorted lexicographically by alignment and then by natural ordering. If no alignment occurs and the formula is independent (i.e., G is empty), then each clause retains its natural literal ordering.

▶ **Example 14.** Consider the following formula:

$$C_1 : (\ell_1 \vee \ell_2 \vee \ell_3) \wedge C_2 : (\ell_4 \vee \ell_5 \vee \ell_6) \wedge (\overline{\ell}_1 \vee \overline{\ell}_6) \wedge (\overline{\ell}_2 \vee \overline{\ell}_4) \wedge \Gamma$$

Assume Γ does not contain a binary clause with literals from C_1 and C_2, so they are both ULCs. We first consider the literals from C_1 in natural order. After ℓ_1 is given alignment 1, ℓ_6 also receives this alignment because of the binary clause $(\overline{\ell}_1 \vee \overline{\ell}_6)$. ℓ_2 is given alignment 2, along with ℓ_4. ℓ_3 is given alignment 3. Now, C_1 is sorted by alignment to $\ell_1 \vee \ell_2 \vee \ell_3$. When C_2 is processed, both ℓ_4 and ℓ_6 are skipped because they already have alignments. ℓ_5 is given alignment 4, and the clause is sorted as $\ell_6 \vee \ell_4 \vee \ell_5$. If the binary clause $\overline{\ell}_2 \vee \overline{\ell}_5$ would have been in Γ, the formula becomes unalignable because ℓ_4 and ℓ_5 would receive alignment 2.

The alignment procedure uses a linear number of steps to assign alignment values, processing each literal within an XLC once and processing all binary clauses within the formula at most once. There are multiple possible alignments for any set of alignable XLCs. For instance, in Example 14, ℓ_5 can be placed at the beginning of C_2 and the XLCs would still be aligned. Specifically, after assigning each literal an alignment value, those values can then be ordered in some way, and that ordering can be used to sort the clauses. In addition, the alignment procedure will always produce an aligned formula if the formula is alignable. This is the case because our procedure aligns all variables within a connected component before moving on to the next connected component, ensuring that the property in Definition 10 is not violated.

The independent and unalignable problems may also benefit from ordering. For example, the `shift1add` family benchmarks each contain only a single ULC (after resolving clashing ULCs) and are thus independent, but in experiments we found by accident that different orderings can lead to 10× difference in solving time. Recent work sorts literals in a single cardinality constraint [25], and a similar approach may be beneficial here. Furthermore, for unalignable problems, partial alignment may still be possible. We focus on the alignable case as these benchmarks tend to yield the most positive results, and leave ordering of independent and unalignable formulas for future work.

Table 1 Number of formulas reencoded, along with their formula types, average preprocessing time, number of formulas that take longer than 20 seconds, the medium number of encoded XLCs, and the medium of the maximum sizes of encoded XLCs. ULC-Clash is ULC detection without resolution on clashing literals. XLC shows results for formulas where at least one proper XLC was detected and reencoded. XLC-full is full detection, combining ULC and proper XLC. SBVA is only run on formulas for which an XLC was detected, and the reencoded count shows the number of those formulas for which SBVA performed any reencoding.

	Reencoded	Alignable	Ind.	Unalign.	Avg. (s)	$\geq$ 20s	Med. #	Med. Max
ULC-clash	891	415	145	331	4	29	760	45
ULC	1014	459	216	339	4	43	760	40
XLC-proper	825	305	187	333	1	9	1001	70
XLC-full	1739	791	294	654	3	49	867	52
SBVA	1616	679	314	623	30	356	–	–

5 Results

We ran our implementation to scan and reencode formulas in the Anniversary track of SAT Competition 2022 [4] for ULCs and XLCs, and the data are in Table 1. Of the 5,355 benchmarks, our method reencoded 1,739 ($\sim$ 32%) formulas with at least one XLC, with 1,014 ($\sim$ 19%) of them having ULCs, of which 459 ($\sim$ 9%) are alignable. These formulas are encoded after resolving clashing ULCs and filtering out clauses of size 4 or less, as explained in Section 4. Notice that by resolving clashing ULCs, we obtain more formulas containing ULCs above the set size threshold. This suggests that our method can sometimes find implicit large ExactlyOne constraints that the user is not aware of when encoding a problem. We also tested if the state-of-the-art reencoding technique SBVA (structured bounded variable addition) [13] would reencode these formulas, and found that it does affect most of them, so our experiments include a performance comparison between our method and SBVA, which will be discussed later in this section. An observation to be made now is that our method is very efficient, taking less than 5 seconds on average to reencode these formulas, while SBVA took 30 seconds on average.

All experiments were carried out at the Pittsburgh Supercomputing Center on nodes with 128 cores and 256 GB RAM [7], running solvers on the 1,014 benchmarks with ULCs. We used a 5,000 second timeout for solving, and 64 experiments were run in parallel per node, so each process held approximately 4GB of memory. SBVA was allowed to run for 200 seconds (if it does not terminate by then) before solving the reencoded formula with CaDiCaL as in the original paper [13]. All runtimes include reencoding time.

We used the following solving configurations for reencoding ULCs: original encoding, SBVA, sequential counter encoding with aligned, natural, or shuffled sorting, and order encoding with aligned sorting. Resolution was performed on clashing unique literal clauses before applying sequential counter and order encodings. Table 2 presents the performance of these configurations on all formulas that contain ULCs, divided into three classes: aligned, independent, and unalignable. Each class is further divided into satisfiable (SAT) and unsatisfiable (UNSAT) instances. Both the number of solved instances and the average solve time (where unsolved formulas contribute twice the time limit, denoted by Par2) are reported. Best performing configurations in each column are highlighted in bold, and a VBS (virtual best solver, which picks the best solver from available configurations) configuration is included for reference.

Table 2 Performance of ULC reencoding across different configurations. *SBVA uses original encoding for the 123 formulas it does not reencode.

	Alignable (459)				Independent (216)				Unalignable (339)			
	# Solved		Avg. PAR2		# Solved		Avg. PAR2		# Solved		Avg. PAR2	
	SAT	UNS	SAT	UNS	SAT	UNS	SAT	UNS	SAT	UNS	SAT	UNS
Original	118	123	1652	3081	**62**	90	**593**	823	**122**	103	1534	1534
SeqShuff	124	122	1237	3066	58	90	1173	899	121	102	1811	1596
SeqNat	125	137	1109	2088	56	**91**	1364	825	**122**	102	**1526**	1560
SeqAli	**127**	152	**951**	1159	56	**91**	1364	831	**122**	103	1544	1470
OrdAli	125	**155**	1149	**993**	58	89	1225	1011	115	90	2019	2595
SBVA*	122	137	1395	2068	61	**91**	616	**713**	120	**107**	1687	**1108**
VBS	137	163	256	513	63	93	301	471	137	112	293	672

Reencoding ULCs with our method provides a significant speedup on solving alignable formulas, even when compared with SBVA. In particular, the sequential counter encoding with alignment yields the best overall results. We visualized the impact of our aligned sequential counter ULC reencoding on solver performance using scatter plots, comparing it against configurations on different formula classes. In each plot, every point represents one benchmark formula, comparing the runtime or conflicts of two solver configurations. Points below the diagonal indicate instances where the tested configuration yields improvement against the baseline configuration (lower values on the y-axis), while points above the line indicate slower performance. We distinguish satisfiable (SAT) and unsatisfiable (UNSAT) instances by marker style in each plot to analyze performance trends across these classes. We now discuss each comparison in detail, highlighting the key patterns and what they reveal about the reencoding method.

We also ran experiments for reencoding XLCs with the aligned sequential counter encoding on formulas where at least one proper XLC was detected and reencoded. On these formulas, reencoding had limited impact, and data will be provided in Section 5.5.

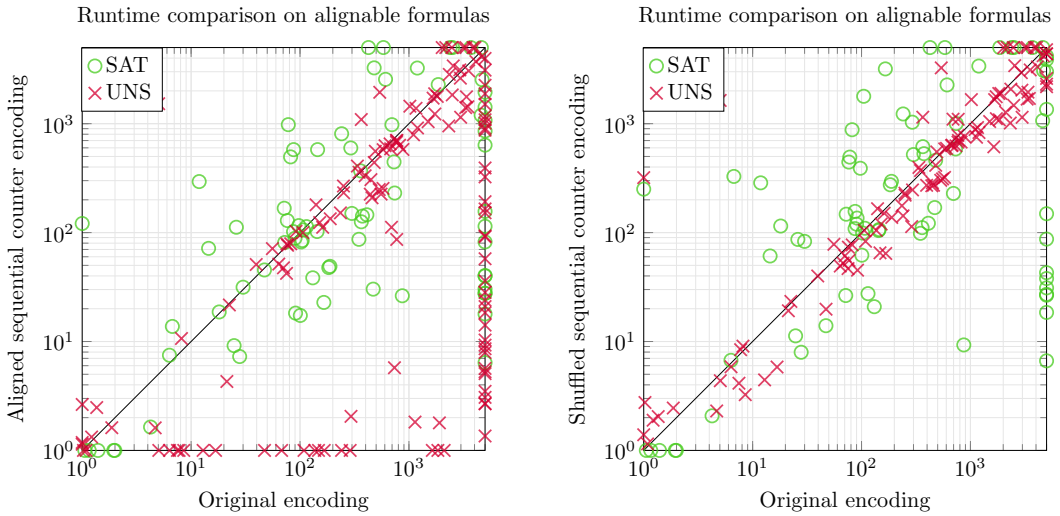


Figure 1 Sequential counter encoding on alignable instances: Aligned vs. original (left) and shuffled vs. original (right).

5.1 Alignable Formulas

The left plot of Figure 1 compares solver runtimes on alignable formulas, contrasting our method's aligned sequential counter encoding against the original encoding (baseline). For alignable formulas, we observe a clear performance improvement with the aligned reencoding, particularly on UNSAT instances. Most UNSAT points lie well below the diagonal, often on the axes, indicating that adopting our method can solve more UNSAT formulas using much less time. The SAT instances exhibit mixed results, but still, our method solves more SAT formulas and has an overall advantage. We have verified all the proofs and satisfying assignments produced by our method on the original formulas in this plot to make sure these speedups are not due to conceptual or implementation errors.

To understand the importance of aligning variables in ULCs, we also evaluated a shuffled reencoding configuration, where the literals in each ULC are randomized (and not aligned) before reencoding. The right plot of Figure 1 compares this configuration against the original encoding on the same set of alignable benchmarks. In stark contrast to the left one, shuffling largely negates the performance gains of our reencoding method. Most points in the plot lie near the diagonal, indicating no change over the baseline. While some formulas experience degraded performance, several satisfiable instances still benefit substantially from the reencoding, even after shuffling. However, the overall outcome is that shuffling offsets the advantage of our reencoding method.

Table 3 shows solver performance on the subset of benchmarks explicitly labeled with "Shuffled" or "Random". Sequential counter and order encodings with aligned sorting provide significant benefits, notably improving the number of solved UNSAT instances (from 55 solved by the original encoding to 73 with the order encoding). These results confirm the importance and effectiveness of alignment.

5.2 Independent and Unalignable Formulas

Experiments on shuffled alignable formulas demonstrate that when the formulas are not aligned, our method does not consistently provide benefits. We therefore expect the same when our method is applied to independent and unalignable formulas. Figures 2 examine the solver's performance on these instances, comparing the reencoded formula to the original. In both classes, most UNSAT instances lie close to the diagonal, and SAT instances are spread out, with no clear performance gains observed. The results suggest that reencoding ULCs tends to provide no benefit, occasionally even hurting performance. As mentioned in Section 4, future work on ordering literals in independent and unalignable formulas may yield better results with our reencoding.

Table 3 Results on aligned with Shuff or Rand in benchmark name (SAT: 63, UNS: 76, UNK: 74).

	# Solved		Avg. PAR2	
	SAT	UNS	SAT	UNS
Original	62	55	159	3341
SeqShuff	**63**	57	**21**	3016
SeqNat	62	58	158	2891
SeqAli	**63**	72	22	1018
OrdAli	**63**	**73**	30	**906**
SBVA	62	60	158	2458
VBS	63	76	21	566

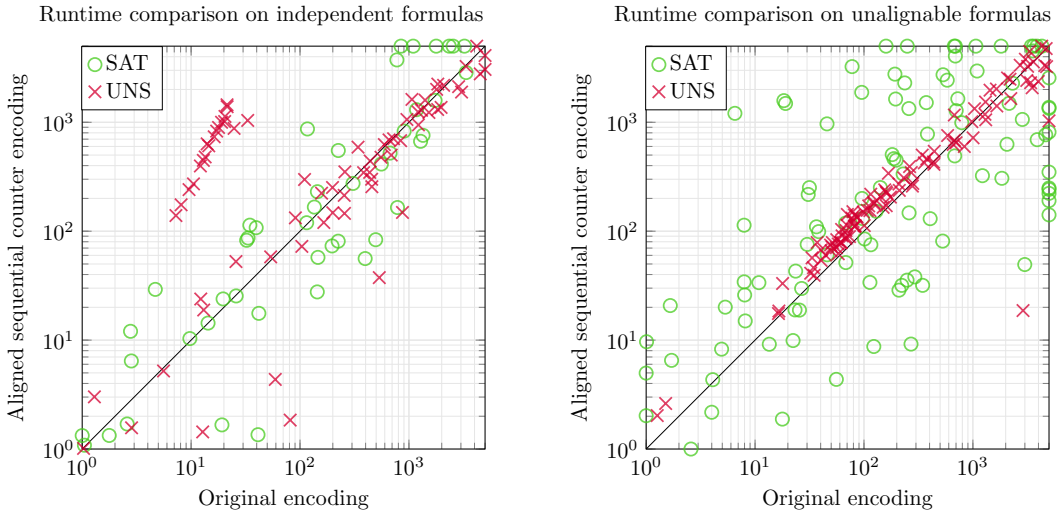


Figure 2 Runtime comparison between the aligned sequential counter encoding and the original encoding on independent formulas (left) and unalignable formulas (right). The cluster of UNSAT formulas in the top left of the independent plot are the shift1add formulas mentioned in Section 4.

5.3 Order Encoding

Our reencoding method can transform the direct encoding of ExactlyOne constraints to the sequential counter or the order encoding. The only difference between the two is that for the latter, variables from the direct encoding are eliminated using variable elimination. Bounded variable elimination [9], a technique used in all state-of-the-art SAT solvers, eliminates variables if that results in a reduction in the number of clauses. After reencoding ULCs, this is the case for all variables from the direct encoding, so solvers could eliminate them. However, they may not be eliminated because the solver can choose to eliminate some of the order variables instead. After eliminating some order variables, the variables from the direct encoding can no longer be eliminated without increasing the size of the formula.

Figure 3 shows the comparison between the sequential counter encoding and the order encoding on aligned formulas. When we compare the runtime (left), both encodings perform quite similarly. This is even more apparent when comparing the number of conflicts (right). If we only focus on the number of solved benchmarks, then the order encoding solves a couple of extra UNSAT formulas, while the sequential encoding solves a couple of extra SAT formulas. The results suggest that the solver does not always eliminate the original variables in ULCs, and sometimes it is better to keep them.

Figure 4 shows the runtime and conflict comparison between the sequential counter encoding and the order encoding on unalignable and independent formulas. The left plot shows that the sequential counter encoding is noticeably faster than the order encoding. This contrasts with the aligned formulas, where the two methods performed similarly. The right plot suggests that the worse performance of the order encoding is due to slower propagation rather than increased conflicts.

5.4 Bounded Variable Addition

Bounded variable addition (BVA) is a reencoding technique that searches for sets of clauses that can be reencoded to a smaller set of clauses by introducing a new variable [20]. In most cases, BVA will more compactly encode the set of binary clauses. BVA was recently

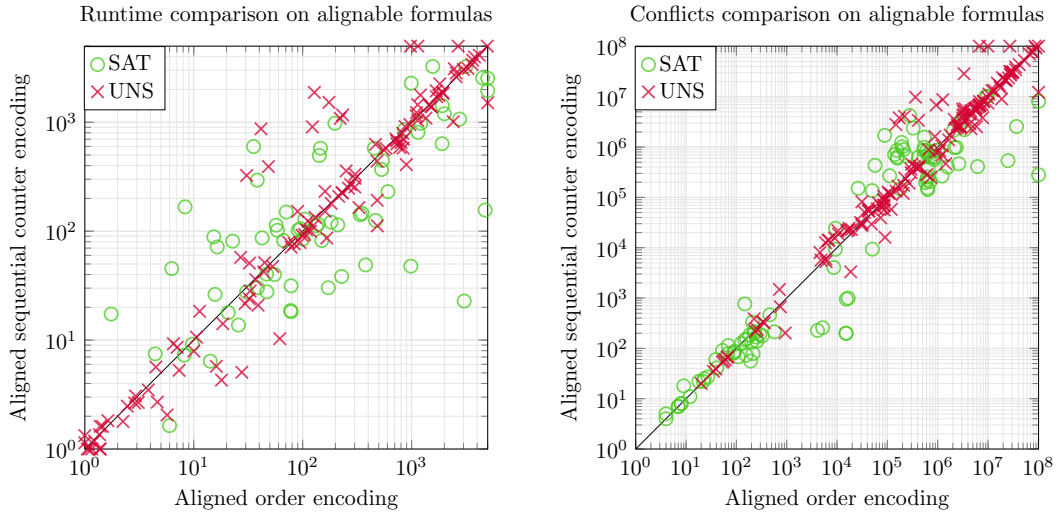


Figure 3 A comparison in runtime (left) and the number of conflicts (right) between the sequential counter encoding and the order encoding on alignable formulas with ULCs.

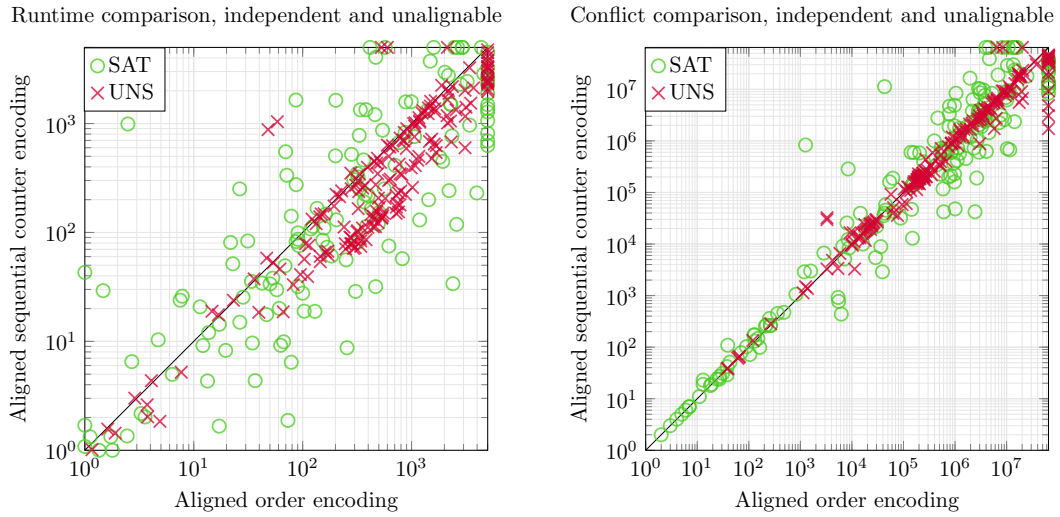


Figure 4 Comparison between runtime (left) and conflicts (right) on unalignable and independent formulas. The reduced runtime on unsatisfiable formulas is due to slower propagation.

improved by taking the structure of the formula into account during reencoding. This version is known as structured BVA (SBVA) [13]. The combination of SBVA and CaDiCaL won the main track of SAT Competition 2023.

We compared our reencoding method with SBVA as preprocessor for CaDiCaL. The results are shown in Figure 5. Our sequential counter encoding with alignment can solve dozens of alignable formulas that cannot be solved when using SBVA, suggesting that they reencode the formulas differently. Note that a large majority of SAT formulas are below the diagonal, showing that our reencoding works much better on both SAT and UNSAT formulas.

On unalignable and independent formulas, the picture is more mixed. Although most points are below the diagonal, there are more formulas that time out after applying our reencoding while they can be solved when using SBVA.

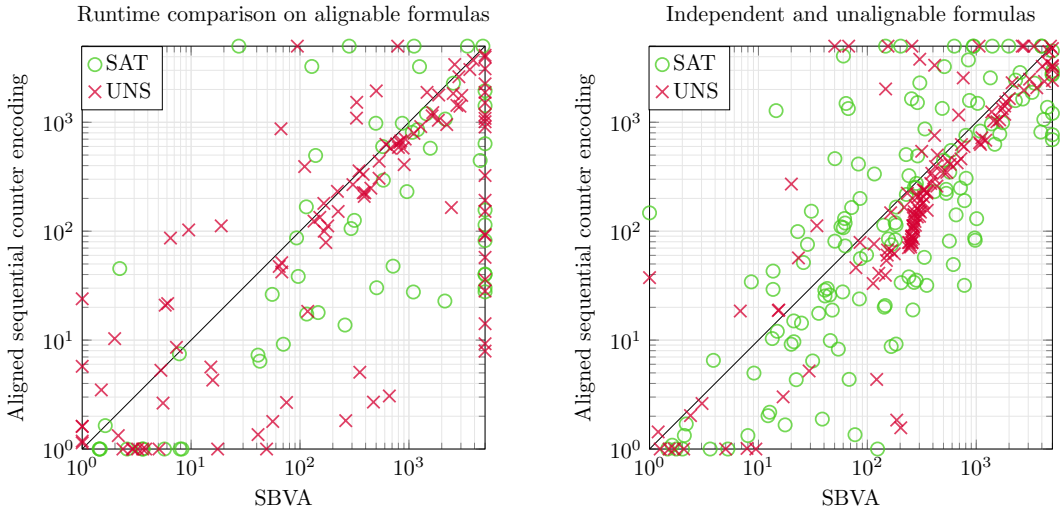


Figure 5 A runtime comparison between aligned sequential counter encoding and SBVA on alignable formulas (left) and unalignable and independent formulas (right). The runtime includes the reencoding time. Formulas for which SBVA does not perform reencoding are filtered out.

Table 4 Performance of XLC reencoding.

	Alignable (305)				Independent (187)				Unalignable (333)			
	# Solved		Avg. PAR2		# Solved		Avg. PAR2		# Solved		Avg. PAR2	
	SAT	UNS	SAT	UNS	SAT	UNS	SAT	UNS	SAT	UNS	SAT	UNS
Original	**119**	132	**318**	723	**102**	47	**644**	1461	**144**	135	**263**	471
XLC	118	**134**	397	**525**	100	**48**	762	**936**	143	**138**	343	**229**
VBS	120	135	207	433	105	49	270	783	145	138	155	206

5.5 Reencoding XLCs

Table 4 and Figure 6 show the performance of sequential counter encoding on formulas containing proper XLCs. Our reencoding consistently helps solve UNSAT formulas while making SAT formulas slower to solve. The impact of the reencoding is not as significant as on formulas containing only ULCs, especially when looking at alignable formulas. XLC reencoding solved two more UNSAT instances (134 vs. 132) but one fewer SAT instance (118 vs. 119), with a modest reduction in average UNSAT solving time (PAR2 reduced from 723 to 525), and a slight increase in average SAT solving time (PAR2 increased from 318 to 397).

Although the improvement is small compared with reencoding the ULCs in alignable formulas, reencoding XLCs still provides an overall performance gain. Moreover, the performance is consistent across alignable, independent, and unalignable formulas. Reencoding XLCs always helps solve a couple more UNSAT formulas, even on independent and unalignable formulas. This was not the case when reencoding the formulas with only ULCs, as our reencoding generally slightly hurts performance on non-alignable formulas.

5.6 Benefited Benchmark Families

We briefly describe some specific families of alignable formulas that our method for reencoding ULCs drastically improves solver performance on.

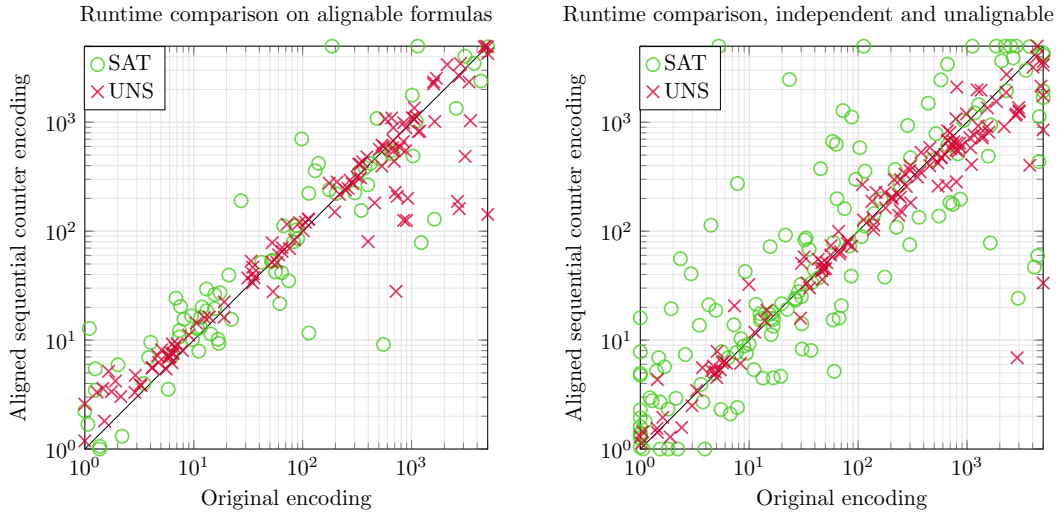


Figure 6 Performance comparison between the sequential counter and the original encoding on formulas with at least one proper XLC. Left, the alignable formulas and right, the unalignable and independent formulas.

- **Pigeonhole** Pigeonhole formulas try to put n pigeons into m holes, consisting of many ULCs (every pigeon must be in one hole). It is one of the famous problems that is easy for humans and hard for SAT solvers, and reencoding ULCs really helps. For example, both the original encoding and SBVA timed out on the formula `php-018-014.shuffled`, which is a shuffled formula that tries to put 18 pigeons in 14 holes, while after reencoding it was solved in 192 seconds. Similar speed boosts were found in related families, such as relativized pigeonhole formulas.
- **Graph Coloring** Another family is graph coloring, also having many ULCs (every vertex has one color). Our method works remarkably well on them. For example, our method solved `queen14_14.col.14` [15] in 9 seconds and `le450_15b.col.15` in 28 seconds, while SBVA and original encoding timed out on both of them.
- **FPGA-routing** These formulas are Field-Programmable Gate Arrays routing constraints encoded as large SAT problems, in which satisfying assignments correspond to feasible routing solutions. The ULCs in these formulas encode connectivity constraints to ensure the existence of a conductive path for each two-pin connection [22]. Reencoding ULCs shows significant performance improvement on almost all such formulas. For example, `homer19.shuffled` timed out with original encoding and took SBVA 75 seconds, but with our method, it was solved in 2 seconds.
- **Petri Net Concurrency** Petri nets are a model for concurrent computation that uses places and transitions [6]. Each formula represents whether there exists a valid partitioning of the Petri net's places into distinct subsets (units) respecting a given concurrency relation. The ULCs in these formulas encode that every place belongs to a unit. Our method drastically improved solver performance on these formulas. For example, `vlsat2_30744_3925645.dimacs` [3], a formula that timed out with both the original encoding and SBVA, was solved by our method in 177 seconds.

We also noticed that the majority of unalignable formulas belong to the Graph/Subgraph Isomorphism family. These formulas consider two random graphs and the question of whether one is a subgraph of the other. This is encoded by searching for a permutation of one of the graphs such that they overlap [2]. The formula contains many unalignable ULCs encoding the permutations.

5.7 Graph coloring

Based on the strong results on graph-coloring problems, we tested whether our version of CaDiCaL enhanced with reencoding boosts SAT-based graph-coloring approaches. For this purpose, we used the CliColCom framework [16], which represents the state-of-the-art in graph coloring and includes several techniques to improve performance for this application, such as symmetry breaking. We ran CliColCom on the DIMACS graph coloring suite [18]. Using CaDiCaL version 2.1.0 (unmodified), the framework could solve 90 benchmarks within an hour, while using the modified CaDiCaL (same version, but with our reencoding technique) allowed solving two more benchmarks. The additional benchmarks are: `queen9_9` (solved in 116 seconds) and `queen10_10` (solved in 2103 seconds). The runtime on most graphs was similar. An exception is `abb313GPIA`, which was solved in 2.12 seconds using the modified version and in 150.11 seconds using the unmodified CaDiCaL. Note that this graph was not solved with CaDiCaL version 1.5.2 used in the CliColCom paper [16].

6 Related Work

Encodings of ExactlyOne constraints have been studied for over 20 years. A seminal early work by Ansótegui and Manyá [1] already uses order variables. Our direct, sequential counter, and order encodings correspond to their standard mapping, regular mapping, and full regular mapping, respectively. They evaluate different encodings on mostly random problems, including graph coloring using the Chaff and Siege solvers. The encodings with order variables generally result in stronger performance, although Chaff is faster using the direct encoding on sparse graphs. A later study didn't observe significant differences in solver performance using various encodings [23]. Our evaluation is the first to involve a massive benchmark suite and concludes that the order variables can boost performance.

A more recent study zooms in on the effectiveness of a range of encodings on graph-coloring problems [10]. That study shows that variants of the order encoding are the most effective on the classical suite of graph-coloring problems [18]. Most instances of that suite are relatively easy, as the chromatic number equals the clique number, so if you find the largest clique, the lower bound call becomes trivial after symmetry breaking. The order encoding is especially effective on the `queen` graphs and `abb313GPIA`, which are exactly the graphs that can be solved using CliColCom when using our reencoding technique.

Other related works deal with reencoding formulas [13, 20, 21]. These works focus on reducing the number of clauses by introducing new variables. In practice, this comes mostly down to constructing a more compact representation of the binary clauses. In contrast, our reencoding may increase the number of clauses, especially in case of formulas with implicit ExactlyOne constraints. The work by Manthey and Steinke [21] is the closest to ours as it also targets ExactlyOne constraints, although they limit their technique to explicit ExactlyOne constraints. Their evaluation showed little benefit of performing reencoding, which is likely due to the small scale of the experiments and ignoring the implicit ExactlyOne constraints.

7 Conclusions

We presented a new reencoding technique for propositional formulas. While all existing work on reencoding focuses on reducing the size of formulas, our technique increases the number of variables and may also increase the number of clauses. The experimental results demonstrate that our method increases the number of solved formulas from SAT Competitions, providing concrete evidence that our reencoding method can significantly enhance solver performance

and even outperform state-of-the-art reencoding techniques on the targeted class of formulas. Importantly, our implementation scans and recovers the target structure in a formula before reencoding, allowing us to predict the performance of our method and determine whether to apply it. Future work on formula structures could leverage our method to a broader range of formulas.

We conjecture that the new variables help guide the solver. However, we could not show that the new variables could exponentially reduce the size of the smallest resolution proof. Answering this question is part of future work.

Our implementation of reencoding logs DRAT proofs, allowing us to formally verify the results presented in the experimental evaluation. In some cases, DRAT proof validation was expensive. To address this issue, CaDiCaL provides LRAT proof logging for all implemented reasoning techniques to create efficiently checkable proofs. In future work, we plan to support LRAT proof production.

References

1 Carlos Ansótegui and Felip Manyà. Mapping problems with finite-domain variables to problems with boolean variables. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing*, pages 1–15, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

2 Calin Anton and Lane Olson. Generating satisfiable SAT instances using random subgraph isomorphism. In Yong Gao and Nathalie Japkowicz, editors, *Advances in Artificial Intelligence, 22nd Canadian Conference on Artificial Intelligence, Canadian AI 2009, Kelowna, Canada, May 25-27, 2009, Proceedings*, volume 5549 of *Lecture Notes in Computer Science*, pages 16–26. Springer, 2009. doi:10.1007/978-3-642-01818-3_5.

3 Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, Finland, 2021.

4 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 Series of Publications B. Department of Computer Science, University of Helsinki, Finland, 2022. Accessed: 2025-03-13. URL: http://hdl.handle.net/10138/359079.

5 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

6 Pierre Bouvier and Hubert Garavel. Efficient algorithms for three reachability problems in safe petri nets. In *Application and Theory of Petri Nets and Concurrency: 42nd International Conference, PETRI NETS 2021, Virtual Event, June 23–25, 2021, Proceedings*, pages 339–359, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-76983-3_17.

7 Shawn T. Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A. Nystrom. *Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research*, pages 1–4. Association for Computing Machinery, New York, NY, USA, 2021. doi:10.1145/3437359.3465593.

8 James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*, AAAI'94, pages 1092–1097. AAAI Press, 1994. URL: http://www.aaai.org/Library/AAAI/1994/aaai94-168.php.

9 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005. doi:10.1007/11499107_5.

10 Daniel Faber, Adalat Jabrayilov, and Petra Mutzel. SAT Encoding of Partial Ordering Models for Graph Coloring Problems. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:20, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.SAT.2024.12.

11 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.

12 Isaac Grosof, Naifeng Zhang, and Marijn J. H. Heule. Towards the shortest drat proof of the pigeonhole principle. In *Workshop on Pragmatics of SAT (PoS')*, 2022. URL: https://api.semanticscholar.org/CorpusID:251040741.

13 Andrew Haberlandt, Harrison Green, and Marijn J. H. Heule. Effective auxiliary variables via structured reencoding. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 271, pages 11:1–11:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl. doi:10.4230/LIPICS.SAT.2023.11.

14 Marijn J. H. Heule. *Proofs of Unsatisfiability*, chapter 15, pages 635–668. IOS Press, 2021. doi:10.3233/FAIA200998.

15 Marijn J. H. Heule, Matti Juhani Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*, volume B-2018-1 of *Department of Computer Science Series of Publications B*. Department of Computer Science, University of Helsinki, Finland, 2018.

16 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.

17 Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 355–370, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-31365-3_28.

18 David S Johnson and Michael A Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

19 O. Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96–97:149–176, 1999. doi:10.1016/S0166-218X(99)00037-2.

20 Norbert Manthey, Marijn Heule, and Armin Biere. Automated reencoding of Boolean formulas. In Armin Biere, Amir Nahir, and Tanja E. J. Vos, editors, *HVC 2012*, volume 7857 of *LNCS*, pages 102–117. Springer, 2012. doi:10.1007/978-3-642-39611-3_14.

21 Norbert Manthey and Peter Steinke. Quadratic direct encoding vs. linear order encoding a one-out-of-n transformation on cnf. Technical Report KRR Report 11-03, Technische Universität Dresden, 2011.

22 Gi-Joon Nam, Fadi Aloul, Karem Sakallah, and Rob Rutenbar. A comparative study of two boolean formulations of fpga detailed routing constraints. In *Proceedings of the 2001 International Symposium on Physical Design*, ISPD '01, pages 222–227, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/369691.369777.

23 Van-Hau Nguyen, Van-Quyet Nguyen, Kyungbaek Kim, and Pedro Barahona. Empirical study on sat-encodings of the at-most-one constraint. In *The 9th International Conference on Smart Media and Applications*, SMA 2020, pages 470–475, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3426020.3426170.

24 Justyna Petke and Peter Jeavons. The order encoding: from tractable csp to tractable sat. Technical Report RR-11-04, DCS, University of Oxford, 2011.

25 Joseph E. Reeves, Joao Filipe, Min-Chien Hsu, Ruben Martins, and Marijn J. H. Heule. The impact of literal sorting on cardinality constraint encodings. In *Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2025.

26 Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. From clauses to klauses [inline-graphic not available: see fulltext]. In *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I*, pages 110–132, Berlin, Heidelberg, 2024. Springer-Verlag. doi:10.1007/978-3-031-65627-9_6.

27 John S. Schlipf, Fred S. Annexstein, John V. Franco, and R. P. Swaminathan. On finding solutions for extended horn formulas. *Inf. Process. Lett.*, 54(3):133–137, May 1995. doi:10.1016/0020-0190(95)00019-9.

28 Aeacus Sheng, Joseph E. Reeves, and Marijn J. H. Heule. jreeves3/ulc-cadical. Software, swhId: swh:1:dir:615c90061cfad33b59249832fb0043defef84419 (visited on 2025-07-29). URL: https://github.com/jreeves3/ulc-cadical, doi:10.4230/artifacts.24212.

29 Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 827–831, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11564751_73.

30 Naoyuki Tamura, Mutsunori Banbara, and Takehide Soh. Compiling pseudo-boolean constraints to sat with order encoding. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 1020–1027, 2013. doi:10.1109/ICTAI.2013.153.

31 Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints An Int. J.*, 14(2):254–272, 2009. doi:10.1007/S10601-008-9061-0.

32 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, Berlin, Heidelberg, 1983. doi:10.1007/978-3-642-81955-1_28.

33 Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. 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*, pages 422–429, Cham, 2014. Springer International Publishing.

Bridging Language Models and Symbolic Solvers via the Model Context Protocol

Stefan Szeider
Algorithms and Complexity Group, TU Wien, Austria

Abstract

This paper presents the *MCP Solver*, a system that bridges large language models with symbolic solvers through the Model Context Protocol (MCP). The system includes a *server* and a *client* component. The server provides an interface to constraint programming (via MiniZinc Python), propositional satisfiability and maximum satisfiability (both via PySAT), and SAT modulo Theories (via Python Z3). The client contains an agent that connects to the server via MCP and uses a language model to autonomously translate problem statements (given in English) into encodings through an incremental editing process and runs the solver. Our experiments demonstrate that this neurosymbolic integration effectively combines the natural language understanding of language models with robust solving capabilities across multiple solving paradigms.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Knowledge representation and reasoning; Computing methodologies → Natural language processing; Theory of computation → Logic and verification; Software and its engineering → Software notations and tools; Computing methodologies → Artificial intelligence

Keywords and phrases Large Language Models, Agents, Constraint Programming, Satisfiability Solvers, Maximum Satisfiability, SAT Modulo Theories, Model Context Protocol

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.30

Supplementary Material *Software*: `https://github.com/szeider/mcp-solver`

Funding This work was supported by Austrian Science Fund (FWF) grants 10.55776/P36420 and 10.55776/COE12.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities across diverse natural language tasks yet exhibit fundamental limitations in formal logical reasoning [6, 11, 12, 15, 18, 19, 22, 24]. In this work, we leverage the robust logical deduction capabilities of symbolic solvers to overcome these limitations. This enables language models to perform complex reasoning tasks with greater reliability. We present the *MCP Solver*, which uses the recently introduced *Model Context Protocol (MCP)* [2] for bridging language models with four complementary solving paradigms.

1. *MiniZinc* [14, 17]: a high-level constraint modeling language that supports global constraints, optimization, and diverse problem domains.
2. *PySAT* [8]: a Python interface to SAT solvers that enables propositional constraint modeling using CNF (Conjunctive Normal Form). The system supports various SAT solvers (including Glucose, Cadical, and Lingeling), with helpers for cardinality constraints.
3. *MaxSAT*: the same Python interface as for SAT enables to model and solve optimization optimization problems via various Maximum Satisfiability (MaxSAT) solvers (including R2) with helpers for cardinality constraints.
4. *Z3* [4, 5]: a SAT Modulo Theories (SMT) solver with Python bindings that supports rich type systems including booleans, integers, reals, bitvectors, and arrays, along with quantifiers for more expressive constraints.

© Stefan Szeider;
licensed under Creative Commons License CC-BY 4.0

The MCP Solver has several use cases. One use caseis its integration into an *AI chatbot interface* (like the Claude Desktop or the Cursor application). During a chat session, the user can state a problem (e.g., a scheduling, combinatorics, or verification problem) in plain English, and the language model will connect to the MCP solver via the provided tools and build an encoding, possibly with interactions from the user, solve the encoding with the backend solver, and report back and interpret the solution. The user can then modify the problem statement. Through this approach, the MCP solver offers an enhanced and highly dynamic user interface for the backend solver where encodings can be developed in a dialog with the language model based on immediate feedback from the solver. Once an encoding has been established, the encoding can be extracted and used in other contexts. This setup also provides educational benefits, as a user can observe how informally stated constraints are formalized for the backend solver, make adjustments, and receive explanations from the language model.

Another use case for the MCP Solver is to provide formal solving capabilities to an *autonomous multi-agent system*. To achieve this, one can connect the MCP Solver via the MCP interface to a *Reason and Act (ReAct)* agent [26], which itself is part of a multi-agent system. To exemplify this use case, we added a test client to the software package. The test client implements a basic two-agent system that automatically encodes problem descriptions provided in plain English. It consists of a ReAct agent that communicates with the server and a reviewer agent that checks the result. Our experiments show that this setup is sufficient for the autonomous encoding of problems with small or medium complexity. For more complex problems, we envisage a multi-agent system with a more refined division of work among agents, for instance, with an orchestrator-workers workflow [1].

The MCP Solver with MiniZinc mode was first released on GitHub in December 2024; PySAT and Z3 modes and the build-in client were added in March, the MaxSAT mode was added in June 2025.

2 Related Work

Several prototype systems for connecting language models with formal solvers have been proposed in recent years. PRoC3S [3] employs a two-stage architecture for robotics planning, generating parameterized skill sequences that undergo continuous constraint satisfaction. In a different direction, a counterexample-guided framework [9] merges an LLM synthesizer with an SMT verifier to enhance correctness guarantees for program synthesis. Several systems focus on translating natural language into solver-friendly formats. SATLM [27] converts natural language into logical formulas suitable for SAT solving, while LOGIC-LM [18] implements a comprehensive pipeline from LLM through symbolic solvers to interpreters. For program synthesis specifically, Lemur [25] offers a task-agnostic language model framework.

The integration between language models and verification tools appears in multiple configurations. The LLM-Modulo framework [10] pairs language models with external verifiers, while GenCP [20] incorporates language models into the domain generation of constraint solvers for text tasks. More specialized approaches include StreamLLM [23], which concentrates on language model-based generation of streamlining constraints to accelerate constraint solving and an LLM-based evolutionary search for SAT preprocessing algorithms [21]. Finally, LLMS4CP [13] shows how pre-trained language models can transform textual problem descriptions into executable constraint programming specifications through retrieval-augmented in-context learning.

While these approaches demonstrate the benefits of combining language models with formal solvers, they typically implement fixed pipelines or tight integration for specific use cases. In contrast, our MCP Solver provides a protocol-based architecture that supports iterative interaction within a range of use cases.

3 Tool Calling and the Model Context Protocol

At the heart of state-of-the-art language model integration is the concept of *tool calling*, where a language model, instead of generating only text, can produce structured data (typically JSON) that corresponds to a request to execute a specific function or tool. This allows the language model to interact with external systems, access live data, or delegate complex computations. The *ReAct (Reason and Act)* framework [26] is an agentic architecture that leverages tool calling in an iterative loop. In each iteration, the agent (which wraps a language model) generates a"thought" (its reasoning), an "action" (a tool call), and then receives an "observation" (the result from the tool). This loop continues until the task is complete. This incremental approach is particularly significant for generating a complex encoding because doing that in one attempt is error-prone and unlikely to succeed. Tool calling, however, enables the agent to build the encoding step-by-step, receiving immediate validation feedback after each addition.

The *Model Context Protocol (MCP)* is designed to provide (among other things) a standardized interface for tool calling. MCP operates between a *client*, hosting a language model, and a *server*, hosting various tools and resources. This interface is particularly valuable because it eliminates the need for each tool developer to create custom integrations with every model provider, and vice versa. The MCP creates a single integration point: tools implement the MCP server interface once, and any MCP-compatible client can immediately use them. At its core, MCP defines a stateful server-client communication that enables language models to maintain context across multiple interactions and invoke external tools through a standardized interface. This statefulness means the server remembers previous interactions within a session.

Since its introduction in late November 2024 [2, 16], the MCP has quickly become a foundational open standard for integrating language models with external tools and data sources. Officially released by Anthropic, it offered SDKs, reference implementations, and integration into the Claude Desktop app—alongside ready-to-use servers for systems like Google Drive, GitHub, Slack, Postgres, and Puppeteer. Open-source and enterprise ecosystems embraced MCP early: platforms such as Zed, Replit, Codeium/Cursor, and Sourcegraph implemented MCP support, reflecting widespread developer engagement. By early 2025, MCP had gained institutional recognition: OpenAI integrated support into its Agents SDK and Responses API, and confirmed upcoming MCP functionality in ChatGPT desktop and enterprise endpoints. Microsoft added MCP to Copilot Studio and Azure OpenAI workflows, providing a first-party C# SDK for tool-connected agents. All these data points underscore MCP's widespread production adoption across leading AI platforms.

4 System Architecture

The MCP Solver offers a server component and a client component. The server connects to any MCP compatible client application; our built-in client (described in Section 5) is just one possibility. The server supports four complementary solver backends: MiniZinc for constraint

programming, PySAT for propositional satisfiability, PySAT-MaxSAT for optimization, and Python Z3 for satisfiability modulo theories. Figure 1 gives a high-level architectural diagram of the system.

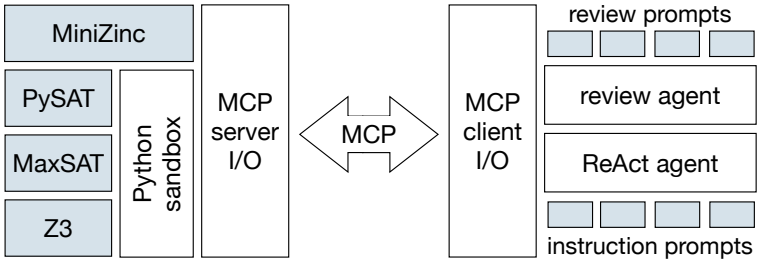


Figure 1 Overview of the MCP Solver architecture showing the main components and their relationships. The system supports four complementary solver backends accessible through a unified MCP interface.

Figure 2 shows the sequence diagram of the MCP Solver when used with an AI chat application as the client, illustrating the dynamic interaction flow between components.

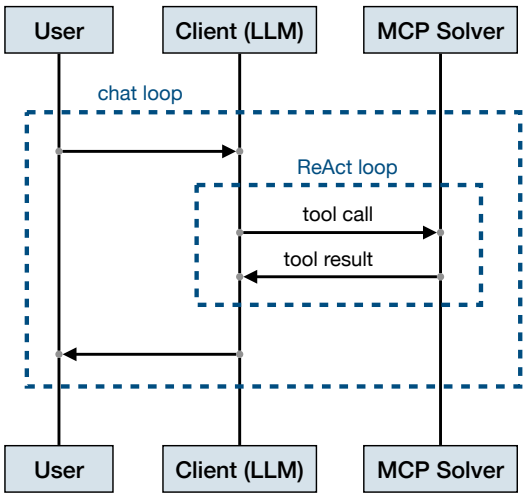


Figure 2 Sequence diagram of MCP Solver's interaction within an AI chat application.

For a unified terminology across all solver backends, we refer to PySAT code, Z3 Python code, or a MiniZinc model as "model", and to a small code entity, like a variable declaration, a MiniZinc constraint, or a Python function definition, as " item ."

4.1 Tools

The MCP Solver provides the following tools, whose item-based operations are demonstrated in Figure 3:

- `clear_model`: reset the solver model;
- `add_item`: add a new item at a specific index;
- `replace_item`: replace an item at a specific index;
- `delete_item`: delete an item at a specific index;
- `get_model`: view the current model with numbered items;
- `solve_model`: solve the model with a specified timeout and receive the solution.

The first version of the MCP solver offered several more tools, but it turned out that fewer tools perform better, as they put a smaller cognitive load on the language model.

In principle, one could run all four solving backends in parallel, with the client selecting on demand the appropriate backend for each problem. However, this burdens the language model with considerable complexity, as it needs to be instructed for all four solving backends. This increases the context size and token use and makes the entire operation potentially confusing for the language model. The current setup assumes that for each session, only one of the four solver backends is used. A command line flag chooses whether the MCP Solver is run in MiniZinc, PySAT, MaxSAT, or Z3 mode.

4.2 Prompt Structure and Design

For each backend (MiniZinc, PySAT, MaxSAT, Z3), a detailed *instruction prompt* is provided to the main ReAct agent via MCP. This prompt acts as its system guide, explaining the available tools and the incremental model-building paradigm. The prompt provides syntax examples and best practices tailored to the specific backend (e.g., MiniZinc syntax vs. Python with PySAT). This focused instruction ensures the agent generates valid code for the chosen solver.

The instruction prompts vary considerably in size across the four modes: MiniZinc requires less than 1000 words, while the other modes use between 3000 and 4000 words. The MiniZinc prompt's brevity stems from the language's declarative nature and standardized features. For instance, solution export is handled automatically by the MiniZinc runtime, whereas the Python-based backends require explicit instructions for formatting and exporting solutions.

The prompts in their current form have emerged via a meta-prompting process. We started with a basic, manually written prompt. Then we carried out an iterative refinement process: observing the performance of the agent, which mistakes or errors are made, followed by adjusting and extending the prompt accordingly.

4.3 Workflow

Next, we describe the full workflow from a problem statement in English to a solution computed by a solver.

1. *Problem Statement.* The process starts with a problem statement in natural language, which can be as brief as "Find a solution to the 10-Queens Problem," and can be entered as a prompt within an AI chatbot or provided as a text file for a command-line client application.
2. *Encoding.* The ReAct agent, guided by its backend-specific instructions prompt, receives the query. It begins to build an encoding by calling `add_item` and other tools.
3. *Incremental Validation.* After each `add_item` call, the MCP Solver's backend validates the new code fragment. It uses MiniZinc methods to evaluate the code (or Python's AST module to parse the code for Python-based backends). If the agent makes a mistake, the server returns a precise error, which the agent observes and corrects in the next step, often by using `replace_item`; if the item is correct, the server returns the encoding generated so far, with items labeled with indices to ensure consistent indexing between client and server. See Figure 3 for an example of this item-based editing approach.
4. *Solving.* Once the problem is fully encoded, the agent calls `solve_model` with a certain timeout. The server executes the backend solver in a secure, isolated process.

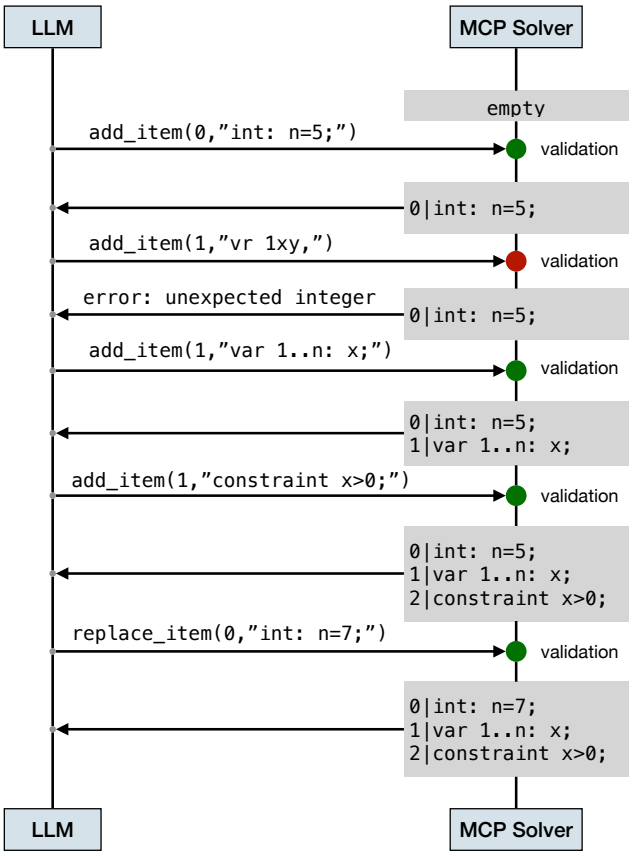


Figure 3 Example for MCP Solver's item-based model editing with validation. Each modification is validated before being applied, maintaining model consistency. Numbers indicate item indices.

5. *Review*. Once the solver has terminated or the timeout has been reached, the instructions tell the agent to critically review the result: in case of "satisfable," whether the solution provided by the solver satisfies all constraints of the problem statement, or, in the case of "unsatisfiable," whether all constraints in the encoding are justified by constraints in the problem statement. In a multi-agent setup, modeling and reviewing are split between two different agents (see Section 5).
6. *Output*. The result is then delivered to the user. In an interactive setup within an AI chatbot, the user can now adjust the problem, add or remove constraints, so that the workflow is repeated from Step 1. This way, the user and the language model can engage in a dynamic dialogue with real-time feedback from the solver.

4.4 Incremental Validation

Each backend performs multi-stage validation before solving, catching errors early, and providing precise diagnostics. In MiniZinc mode, the system reconstructs the complete model after each edit—automatically injecting necessary global definitions—and submits it to the MiniZinc analyzer. This single operation combines parsing, type checking, and model-level consistency checking. The system intercepts syntax, typing, or semantic errors and reports them with exact line and column details.

The Python-based SAT and MaxSAT modes execute user scripts in sandboxed subprocesses, isolating validation from the main application. The isolated environment parses code to detect syntax errors, then applies a restrictive import policy to block unauthorized modules. An Abstract Syntax Tree (AST) analysis pass identifies common logical errors, such as reassigning collections to scalars. Before executing the solving logic, the system verifies that scripts contain the required solver invocation and solution export calls, providing immediate warnings for missing calls.

Z3 mode validation balances safety with responsiveness. A pre-execution pass blocks disallowed operations—including arbitrary file I/O and dynamic code execution—while catching syntax errors through parsing. Unlike SAT backends, Z3 does not enforce solver-invocation patterns at parse time. A runtime wrapper monitors for missing model export or satisfiability check calls, implementing default extraction mechanisms when necessary.

4.5 Execution and Solution Processing

After validation, each backend executes the solving phase with isolation and timeout controls while maintaining a uniform synchronous interface. MiniZinc mode passes solve requests to the MiniZinc library within an asynchronous worker thread bounded by a hard timeout. The library manages solver processes with inherited isolation and cleanly cancels jobs on timeout to prevent resource leakage.

Python-based SAT and MaxSAT backends execute user scripts in dedicated subprocesses. The system captures standard output and errors for reporting, with CPU-time alarms enforcing configured time limits. After completion, subprocesses package satisfiability status, variable assignments, and objective values, returning them to the main process through secure channels.

Z3 mode runs in the main process within a restricted namespace, exposing only approved operations and the Z3 API. Run-time alarms prevent overruns, and the runtime wrapper that handles missing export calls captures valid models even when users omit explicit solution-export steps. All backends normalize errors from validation, sandbox enforcement, or solving into structured codes and messages, providing consistent feedback to MCP clients.

Currently, the solving is performed synchronously; when the `solve_model` tool is called, the entire system waits for the solver to return a solution or for the given timeout to be reached. The timeout is provided as a parameter for the tool call. For our tests, a maximum timeout of 30 seconds was sufficient and worked well in conjunction with the internal timeouts of the AI chatbot application. For problems with longer solving times, we consider adding an asynchronous solving tool that starts the solving in the background and another tool that queries the solving status.

4.6 Modularity

The MCP Solver is architected for modularity, enabling extension with new solver backends. The core design is built on abstract solver interfaces and a shared validation/security scaffolding. The system defines a solver manager interface, which mandates a set of methods that any backend solver must implement (the tools). A base class provides a default implementation that backends can inherit. To add a new solver, one needs to create a new module and implement a class that adheres to this interface. For Python-based solvers, the system provides the security and validation infrastructure described above, including the execution environment and AST-based analysis. This modularity extends to the client, which we will discuss next. Adapting the client to a new solver backend requires only providing a new review prompt.

5 MCP Client

Our package includes an MCP client that provides a command-line interface to the MCP Solver. The client implements a ReAct agent, which utilizes a language model; our model-agnostic interface allows the use of frontier models from all major providers. Although the client does not include a dynamic dialog, as is the case with an AI chatbot, the looping between the client and the server is unlimited. The instruction prompt for the ReAct client includes the request to verify the solution. This is an effective way of self-control, and we have observed that often, the agent identifies a wrong solution and modifies the encoding.

To enhance reliability, the client includes a dedicated *review agent* that independently reviews each solution, following the "LLM-as-a-judge" technique [7]. The reviewer's task is to independently scrutinize the solution: If the result is "satisfiable," does the provided solution correctly satisfy the model? If the result is "unsatisfiable," does the generated model accurately encode all constraints from the original problem statement, and no further spurious constraints?

We do not provide the review agent with the entire message history on purpose, only the problem description, the model, and the solution. This way, the review agent can focus only on this task and is not distracted. By default, the review and the ReAct agent use the same language model, but, in principle, they could use different ones.

Figure 4 shows a sequence diagram of the entire workflow. If the reviewer found an error, one could loop back to the ReAct agent to try again, equipped with feedback.

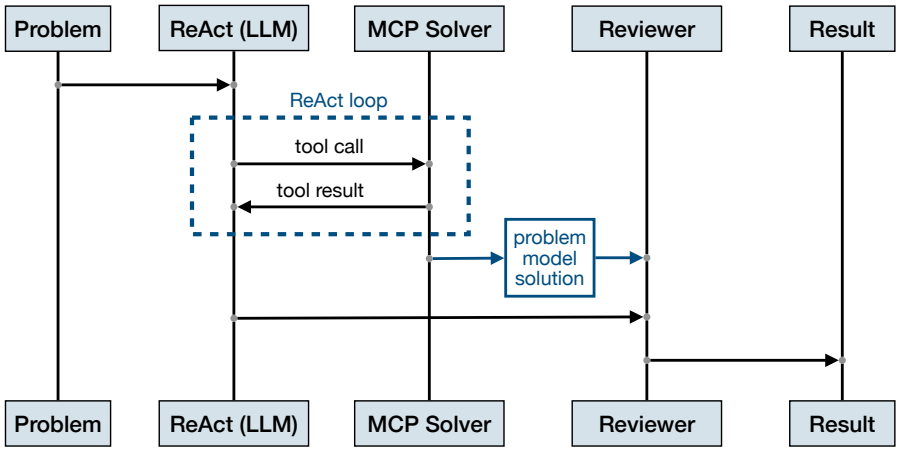


Figure 4 Sequence diagram of client-server interaction of the MCP solver.

The client has proven useful for developing and debugging the solver integration as it has all components (server, client, problems, prompts) at the same location, and hence can adjust the seamless communication between these components. However, the client lacks the interactive aspect provided by an AI chatbot and works as a one-shot encoder.

6 Experiments

We tested the MCP Solver on benchmark problems stated in natural language. All results refer to version 3.3.0 of the MCP Solver. It uses the Python packages mcp (1.5.0), minizinc (0.10.0), z3-solver (4.14.1.0), and python-sat (1.8.dev16). For the client, we use langgraph (0.3.21). As the language model, we use Claude Sonnet 4 version 20250514. Our client can

easily be adapted to other language models because of our model-agnostic architecture. In non-rigorous tests with other language models (GPT-4.1, O3, O4-mini, and Gemini-2.5-pro-preview) only GPT-4.1 performed comparably with Claude Sonnet 4.

We considered twenty problem descriptions, five for each of the four solver backends. The problems can be found as markdown files on the MCP Solver's GitHub repository. These problems were chosen to represent a wide range of problem types from various domains, including scheduling, optimization, verification, combinatorics, and logical puzzles. The selection demonstrates the diversity of possible applications and the individual strengths of the solver backends. For each solver backend, one of the five selected problems is unsatisfiable.

We ran all twenty problem descriptions five times. Table 1 shows the results, including statistics on the number of tool calls and token usage.

Table 1 The table shows average token usage in thousands (k) for *ReAct* (ReAct Agent Total) and *Rev* (Reviewer Total), followed by the average number of tool calls: C (`clear_model`), A (`add_item`), R (`replace_item`), D (`delete_item`), G (`get_model`), and S (`solve_model`). (5 runs per problem).

Mode	*Problem*	C	A	R	D	G	S	*ReAct*	*Rev*
MiniZinc	tsp	2.4	16.2	0.0	0.2	3.2	2.4	143.5k	2.8k
	carpet_cutting	1.0	9.8	0.0	0.0	1.0	1.0	58.6k	3.6k
	zebra	1.0	12.6	0.0	0.0	1.0	1.0	86.7k	3.0k
	university_scheduling	1.0	14.2	0.0	0.0	1.0	1.0	106.5k	3.2k
	university_scheduling_unsat	2.2	24.2	1.4	0.0	4.6	4.2	426.7k	2.9k
PySAT	furniture_arrangement	1.0	7.8	0.0	0.0	1.0	1.0	127.4k	4.1k
	sudoku_16x16	1.0	8.6	0.0	0.0	1.2	1.2	186.3k	8.8k
	equitable_coloring_hajos	1.0	6.4	0.2	0.0	1.2	1.2	120.0k	4.6k
	no_three_in_line_5x5	2.6	11.4	5.6	0.2	7.4	7.4	825.5k	5.2k
	petersen_12_coloring_unsat	1.6	8.2	0.4	0.2	2.4	2.2	269.4k	4.0k
MaxSAT	task_assignment	1.0	5.0	0.0	0.0	1.0	1.0	85.8k	3.4k
	equipment_purchase	1.0	5.6	0.0	0.0	1.2	1.0	98.5k	3.5k
	package_selection	1.0	5.6	0.6	0.0	1.4	1.2	114.5k	3.5k
	network_monitoring	1.2	6.0	0.6	0.0	2.0	1.8	152.6k	4.1k
	workshop_scheduling_unsat	1.8	9.0	3.0	0.0	5.2	4.8	396.9k	3.0k
Z3	cryptarithmetic	1.0	6.8	0.0	0.0	1.0	1.0	100.9k	2.6k
	sos_induction	1.0	8.0	0.2	0.0	1.4	1.4	155.6k	4.7k
	array_property_verifier	1.0	7.8	0.0	0.0	1.0	1.0	110.2k	2.0k
	processor_verification	1.8	11.8	0.4	0.0	2.0	2.0	284.4k	3.8k
	bounded_sum_unsat	1.2	10.6	0.0	0.0	2.2	2.2	173.5k	3.2k

All hundred runs produced solutions and were confirmed by the reviewer. By manual inspection, we could verify that indeed all results were correct.

Tool usage patterns show the difficulty a problem poses to the agent. Most problems required single solve attempts, with `solve_model` calls averaging 1.0–1.2 per problem. The `add_item` tool usage ranges from 5.0 calls for simple MaxSAT problems to 24.0 calls for the MiniZinc zebra puzzle. Increased use of `delete_item`, `replace_item`, and `clear_model` indicates that the agent requires several attempts (one call to `clear_model` is obligatory at start). The difficulty of a problem can also be seen from the token usage, which varies by more than a factor of 10 between the easiest and the hardest problem.

7 Conclusion

The MCP Solver provides language models access to formal solving and reasoning capabilities via the standardized MCP interface. By supporting multiple solving paradigms, the MCP Solver addresses a broad range of problems while maintaining a consistent interface. The flexible architecture enables various use cases, from dynamic problem refinement through natural language interaction when integrated into an AI chatbot to the integration into a multi-agent system for autonomous modeling and solving. The MCP Solver, although stable and application-ready, is still under development. Presently planned additions are Minimal Unsatisfiable Subset support for PySAT, Answer Set Programming, and Model Counting modules, as well as an asynchronous solving interface for longer timeouts. The support of encodings that process instance data (such as graph or tabular data or MiniZinc data files) would also be an interesting addition that enhances the system's versatility. The MCP solver can be integrated into a multi-agent system that uses an orchestrator-workers workflow to autonomously develop more complex encodings, where the encoding task is split into independent components. Such a system could include several solver backends with a routing agent deciding which one to use. Such an approach can optimize solving time by autonomously generating and testing alternative encodings for components.

References

1 Anthropic. Building effective agents, 2024. URL: `https://www.anthropic.com/engineering/building-effective-agents`.

2 Anthropic. Model context protocol: A standard for AI system integration, October 2024. URL: `https://modelcontextprotocol.io`.

3 Aidan Curtis, Nishanth Kumar, Jing Cao, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Trust the PRoC3S: Solving long-horizon robotics problems with llms and constraint satisfaction. In Pulkit Agrawal, Oliver Kroemer, and Wolfram Burgard, editors, *Conference on Robot Learning, 6-9 November 2024, Munich, Germany*, volume 270 of *Proceedings of Machine Learning Research*, pages 1362–1383. PMLR, 2024. URL: `https://proceedings.mlr.press/v270/curtis25a.html`.

4 Leonardo de Moura and Nikolaj Bjørner. Z3 API in Python. `https://ericpony.github.io/z3py-tutorial/guide-examples.htm`. Accessed: 2025-03-20.

5 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. `doi:10.1007/978-3-540-78800-3_24`.

6 Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: program-aided language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR, 2023. URL: `https://proceedings.mlr.press/v202/gao23f.html`.

7 Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Yuanzhuo Wang, and Jian Guo. A survey on LLM-as-a-Judge. *CoRR*, abs/2411.15594, 2024. `doi:10.48550/arXiv.2411.15594`.

8 Alexey Ignatiev, Zi Li Tan, and Christos Karamanos. Towards universally accessible SAT technology. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*, volume 305 of *LIPIcs*, pages 16:1–16:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. `doi:10.4230/LIPICS.SAT.2024.16`.

9 Sumit Kumar Jha, Susmit Jha, Patrick Lincoln, Nathaniel D. Bastian, Alvaro Velasquez, Rickard Ewetz, and Sandeep Neema. Counterexample guided inductive synthesis using large language models and satisfiability solving. In *IEEE Military Communications Conference, MILCOM 2023, Boston, MA, USA, October 30 - Nov. 3, 2023*, pages 944–949. IEEE, 2023. doi:10.1109/MILCOM58377.2023.10356332.

10 Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Saldyt, and Anil Murthy. Position: Llms can't plan, but can help planning in LLM-modulo frameworks. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL: https://openreview.net/forum?id=Th8JPEmH4z.

11 Bill Yuchen Lin, Ronan Le Bras, Kyle Richardson, Ashish Sabharwal, Radha Poovendran, Peter Clark, and Yejin Choi. Zebralogic: On the scaling limits of llms for logical reasoning. *CoRR*, abs/2502.01100, 2025. doi:10.48550/arXiv.2502.01100.

12 Zhan Ling, Yunhao Fang, Xuanlin Li, Zhiao Huang, Mingu Lee, Roland Memisevic, and Hao Su. Deductive verification of chain-of-thought reasoning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL: http://papers.nips.cc/paper_files/paper/2023/hash/72393bd47a35f5b3bee4c609e7bba733-Abstract-Conference.html.

13 Kostis Michailidis, Dimos Tsouros, and Tias Guns. Constraint modelling with llms using in-context learning. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain*, volume 307 of *LIPIcs*, pages 20:1–20:27. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CP.2024.20.

14 MiniZinc Development Team. *MiniZinc Python: Native Python Interface for the MiniZinc Toolchain*, 2025. Accessed: 2025-03-20. URL: https://python.minizinc.dev/.

15 Seyed-Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. GSM-Symbolic: understanding the limitations of mathematical reasoning in large language models. *CoRR*, abs/2410.05229, 2024. doi:10.48550/arXiv.2410.05229.

16 Model Context Protocol Development Team. Model context protocol: Seamless integration between llm applications and external data sources, 2025. Accessed: 2025-03-20. URL: https://github.com/modelcontextprotocol.

17 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

18 Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. Logic-LM: empowering large language models with symbolic solvers for faithful logical reasoning. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 3806–3824. Association for Computational Linguistics, 2023. doi:10.18653/V1/2023.FINDINGS-EMNLP.248.

19 Chengwen Qi, Ren Ma, Bowen Li, He Du, Binyuan Hui, Jinwang Wu, Yuanjun Laili, and Conghui He. Large language models meet symbolic provers for logical reasoning evaluation. *CoRR*, abs/2502.06563, 2025. doi:10.48550/arXiv.2502.06563.

20 Florian Régin, Elisabetta De Maria, and Alexandre Bonlarron. Combining constraint programming reasoning with large language model predictions. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain*, volume 307 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CP.2024.25.

21 André Schidler and Stefan Szeider. Extracting problem structure with LLMs for optimized SAT local search. In *The 18th International Symposium on Combinatorial Search (SoCS 2025)*, 2025. To Appear.

22 Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity, 2025. arXiv:2506.06941.

23 Florentina Voboril, Vaidyanathan Peruvemba Ramaswamy, and Stefan Szeider. StreamLLM: Enhancing constraint programming with large language model-generated streamliners. In *2025 IEEE/ACM 1st International Workshop on Neuro-Symbolic Software Engineering (NSE)*, pages 17–22, Los Alamitos, CA, USA, 5 2025. IEEE Computer Soc. URL: https://doi.ieeecomputersociety.org/10.1109/NSE66660.2025.00010, doi:10.1109/NSE66660.2025.00010.

24 Yuxuan Wan, Wenxuan Wang, Yiliu Yang, Youliang Yuan, Jen-tse Huang, Pinjia He, Wenxiang Jiao, and Michael R. Lyu. Logicasker: Evaluating and improving the logical reasoning ability of large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 2124–2155. Association for Computational Linguistics, 2024. doi:10.18653/V1/2024.EMNLP-MAIN.128.

25 Haoze Wu, Clark W. Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL: https://openreview.net/forum?id=Q3YaCghZNt.

26 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL: https://openreview.net/forum?id=WE_vluYUL-X.

27 Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Satlm: Satisfiability-aided language models using declarative prompting. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL: http://papers.nips.cc/paper_files/paper/2023/hash/8e9c7d4a48bdac81a58f983a64aaf42b-Abstract-Conference.html.

On Top-Down Pseudo-Boolean Model Counting

Suwei Yang
Grabtaxi Holdings, Singapore
National University of Singapore, Singapore

Yong Lai
Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, Changchun, China

Kuldeep S. Meel
Georgia Institute of Technology, Atlanta, GA, USA
University of Toronto, Canada

Abstract

Pseudo-Boolean model counting involves computing the number of satisfying assignments of a given pseudo-Boolean (PB) formula. In recent years, PB model counting has seen increased interest partly owing to the succinctness of PB formulas over typical propositional Boolean formulas in conjunctive normal form (CNF) at describing problem constraints. In particular, the research community has developed tools to tackle exact PB model counting. These recently developed counters follow one of the two existing major designs for model counters, namely the bottom-up model counter design. A natural question would be whether the other major design, the top-down model counter paradigm, would be effective at PB model counting, especially when the top-down design offered superior performance in CNF model counting literature.

In this work, we investigate the aforementioned top-down design for PB model counting and introduce the first exact top-down PB model counter, PBMC. PBMC is a top-down search-based counter for PB formulas, with a new variable decision heuristic that considers variable coefficients. Through our evaluations, we highlight the superior performance of PBMC at PB model counting compared to the existing state-of-the-art counters PBCount, PBCounter, and Ganak. In particular, PBMC could count for 1849 instances while the next-best competing method, PBCount, could only count for 1773 instances, demonstrating the potential of a top-down PB counter design.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Pseudo-Boolean, Model Counting, Constraint Satisfiability

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.31

Related Version *Full Version*: https://arxiv.org/abs/2506.05232

Funding This work was supported in part by the Grab-NUS AI Lab, a joint collaboration between GrabTaxi Holdings Pte. Ltd. and National University of Singapore, and the Industrial Postgraduate Program (Grant: S18-1198-IPP-II), funded by the Economic Development Board of Singapore. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) [RGPIN-2024-05956], Jilin Provincial Natural Science Foundation [20240101378JC], and Jilin Provincial Education Department Research Project [JJKH20241286KJ].

Acknowledgements The authors thank the reviewers for their feedback. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore.

1 Introduction

Model counting refers to the task of computing the number of satisfying assignments of a given logical formula, typically in the form of a propositional Boolean formula in conjunctive normal form (CNF). Model counting has witnessed sustained attention from the research community in the past decades, owing to its relevance as a technique for tackling problems in


© Suwei Yang, Yong Lai, and Kuldeep S. Meel;
licensed under Creative Commons License CC-BY 4.0

various domains such as circuit vulnerability analysis, network reliability, and probabilistic inference [8, 6, 16]. While model counting techniques are applied across various domains, a potential barrier to their wider adoption is the requirement for users to express the domain problem as a CNF formula before passing it to model counters. As a result, there has been recent interest in the research community in developing model counting tools for alternative input logic formats, such as PBCount [19, 20], PBCounter [13], and ApproxMCPB [18] for pseudo-Boolean formulas and sharpASP [9] and ApproxASP [10] for answer set programming. The motivation for exploring alternative input formats is in part due to the fact that these inputs are more expressive and might be natural for certain application domains.

One notable emerging alternative input format for the model counting task is pseudo-Boolean (PB) formulas. PB formulas and related tools have seen an increase in attention from the research community in recent years, partly owing to the succinctness of PB formulas over typical CNF formulas [14]. There are typically two major paradigms when designing model counters – bottom-up and top-down designs. Whilst both designs have been used in exact CNF model counters, the existing exact PB model counters, namely PBCount and PBCounter, are of the bottom-up design. In particular, both counters make use of algebraic decision diagrams (ADDs) [1] as building blocks to perform counting in a bottom-up manner, starting with ADDs which represent individual constraints of the input PB formula and combining the individual ADDs in the computation process. The bottom-up design enables PBCount and PBCounter to reuse some of the decision diagram techniques found in existing ADD-based CNF model counters, namely ADDMC and DPMC [3, 4]. A natural question one would wonder is *"How would a top-down counter design perform in comparison to existing counters for PB model counting?"*.

In this work, we address the aforementioned question via a prototype top-down exact PB model counter, which we term PBMC. PBMC uses a top-down search-based approach along with techniques similar to conflict-driven clause learning (CDCL), adapted for PB formulas, to perform model counting. In particular, PBMC builds on top of methodologies adapted from RoundingSat PB solver [7], GPMC CNF model counter [15], and PBCount PB counter [19]. In addition, we developed a new variable decision ordering heuristic, VCIS, and a new component caching scheme tailored for PB formulas. In our evaluations, PBMC demonstrates superior performance to state-of-the-art PB counters PBCount and PBCounter, and CNF counter Ganak [19, 13, 17]. More specifically PBMC is able to return counts for 1849 instances while PBCount could only return for 1773 instances, PBCounter for 1508 instances, and Ganak for 1164 instances. In addition, we show that our VCIS variable decision heuristic brings performance benefits, enabling PBMC to improve from 1772 instances using prior heuristics to 1849 instances with VCIS.

2 Background and Preliminaries

2.1 Pseudo-Boolean Formula

A pseudo-Boolean (PB) formula F consists of a set Ω of one or more pseudo-Boolean constraints. Each PB constraint $\omega \in \Omega$ is of the form $\sum_{i=1}^{n} a_i x_i \triangleright k$, where $x_1, ..., x_n$ are Boolean literals, $a_1, ..., a_n$, and k are integers, and $\triangleright$ is one of $\{\geq, =, \leq\}$. $a_1, ..., a_n$ are referred to as term coefficients in the PB constraint, where each term is of the form $a_i x_i$ and k is known as the degree of a PB constraint. An assignment σ is a satisfying assignment if it assigns values to all variables of F such that all the PB constraints in Ω evaluate to *true*. PB model counting refers to the task of determining the number of satisfying assignments of F.

Without loss of generality, the PB constraints in this work are all of the "$\geq$" type unless otherwise stated as "$\leq$" type constraints can be converted by multiplying -1 on both sides, and "$=$" type constraints can be replaced with a pair of "$\geq$" and "$\leq$" constraints. In addition, all coefficients are positive unless stated otherwise as the signs of coefficients can be changed using $\bar{x} = 1 - x$ and updating the degree accordingly. We use the term *gap s* to denote the satisfiability gap of a PB constraint ω under assignment σ. A gap value of 0 or less indicates ω is always satisfied under σ, otherwise ω is yet to be satisfied.

▶ **Definition 1.** *Let ω be a PB constraint $\sum a_i \ell_i \geq k$ and σ be a partial assignment. The gap s of ω under σ is given by $k - (\sum_{j:\sigma(\ell_j)=1} a_j)$, where $j : \sigma(\ell_j) = 1$ indicates all literals ℓ that evaluates to true under σ.*

2.2 Bottom-Up Model Counting

Model counters that are of the bottom-up design typically make use of decision diagrams, which are directed acyclic graph (DAG) representations of functions, in the computation process. In particular, the counters make use of the Apply operation [2] to build up the representation of the input logical formula in a bottom-up manner. Notable bottom-up counters including ADDMC, DPMC, PBCounter, and PBCount [3, 4, 5, 13, 19, 20] employ algebraic decision diagrams (ADDs) [1] to perform model counting tasks. These counters use early projection of variables to reduce the size of the intermediate decision diagrams in practice, as the complexity of the Apply operation is on the order of the product of sizes of the two operand diagrams. The aforementioned counters first construct individual ADDs of the input formula components i.e. clauses in the case of CNF and constraints in the case of PB formula. The individual ADDs are then merged via the Apply operation with the "$\times$" operator to produce intermediate ADDs. Variables are projected out using the Apply operation with the "$+$" operator according to either an initially computed plan or via heuristics. Once all ADDs are merged and all variables are projected out, the final ADD has a single leaf node that indicates the model count.

2.3 Top-Down Model Counting

Existing top-down model counters, typically for CNF formulas, adopt similar search methodologies in solvers, which tend to be the Conflict-Driven Clause Learning (CDCL) algorithm, to compute the model count. Notable counters with the top-down component caching design include D4, GPMC, Ganak, and SharpSAT-TD [12, 15, 17, 11] which demonstrated superior performance as winners of recent model counting competitions. On a high level, top-down counters iteratively assign values to variables until all variables are assigned a value or a conflict is encountered. In the process, sub-components are created by either branching on a variable or splitting a component into smaller variable-disjoint components. The counters cache the components and their respective counts to avoid duplicate computation. When all variables are assigned, the counter reaches a satisfying assignment and will assign the current component the count 1 and backtrack to account for other branches. When the counter encounters a conflict i.e. the existing assigned variable already falsifies the formula, it learns the reason for the conflict and prunes the search space such that this sub-branch of the search will be avoided in the remaining computation process. A leaf component in the implicit search tree either has a count of 0 for conflicts and a count of 1 when all variables are assigned. The count of a component that has variable-disjoint sub-components is the product of the counts of its sub-components. The count of a component that branches on a variable is the sum of the counts from the two branches. After accounting for all search branches, the counter returns the final count of satisfying assignments.

3 Approach

In this section, we detail the general algorithm of our top-down PB model counter PBMC as well as design decisions that enable a performant PB counter.

3.1 Search-Based Algorithm

Algorithm 1 CountPBMC – Counting Algorithm of PBMC.

Input : PB formula F
Output : Model count
1 $F \leftarrow \mathsf{Preprocess}(F)$
2 component $\varphi \leftarrow F$; cache $\zeta \leftarrow \emptyset$; assignment $\sigma \leftarrow \emptyset$
3 $\mathsf{Count}(\varphi, \sigma, \zeta)$
4 **return** $\zeta[\varphi]$

The overall counting approach of PBMC is shown in Algorithms 1 and 2. CountPBMC starts with preprocessing the input PB formula F with techniques adapted from existing state-of-the-art counters [19]. Subsequently, Count sets the preprocessed F as the root component, and initializes an empty cache ζ and an empty assignment σ. CountPBMC calls Count to search the space of assignments and compute the model count. The main computation process of PBMC in Count comprises of the following steps – propagation (line 2), conflict handling (lines 3–7), variable decision (lines 11–13), and component decomposition (line 9). As shown in Count, PBMC starts with propagation so that variable decisions can be inferred and applied. If the propagation leads to a conflict, PBMC would perform conflict analysis and backjumping using similar techniques as RoundingSat [7] (lines 5–6). If the conflict is a top-level conflict, that is it cannot be resolved by backjumping, PBMC will terminate and return 0 as the formula is unsatisfiable (line 7). Otherwise, PBMC would learn a PB constraint that prunes the search space, backjump to the appropriate decision level d, and clear the components and recursive calls created after d.

If there are no conflicts, PBMC will attempt to split the current component into variable-disjoint child components, and the count of the current component would be the product of child component counts (lines 9–10). There are two cases of component splitting – (i) there are no new disjoint components and (ii) there are new disjoint components. In case (i) PBMC will proceed to branch on another variable, and the count of the current component would be the sum of counts from the two branches (lines 12–13). If there are no unassigned variables remaining, the count of the current component would be 1 (line 14). Finally, after completing the implicit search tree exploration PBMC returns the count of F in cache ζ.

3.2 VCIS Variable Decision Heuristic

In our top-down PB model counter PBMC, we introduce a variable decision ordering heuristic that we term *variable coefficient impact score* (VCIS). The intuition for a new variable decision heuristic for PB formulas came from the fact that the coefficient of each literal affects its impact on the overall counting process. Existing variable decision heuristics from CNF model counting literature do not have to consider the impact of coefficients, and therefore modifications are required to adapt to PB model counting. Additionally, tree decomposition may also be less effective when dealing with PB constraints that are relatively longer or constraints that share a lot of variables as in the case of multi-dimensional knapsack problems.

Algorithm 2 Count – helper function of CountPBMC.

```
   Input  : component φ, assignment σ, cache ζ
 1 if φ entry in ζ then return ζ[φ]
 2 isConf ← propagate(σ)
 3 if isConf then
 4     ζ[φ] ← 0
 5     success ← ConflictAnalysis(σ)
 6     if success then Backjump()    ▷ Clears related cache & recursive calls
 7     else ExitReturnZero()                  ▷ Unresolvable conflict, return 0
 8 else
 9     C ← SplitComponent(φ, σ)
10     if |C| > 1 then ζ[φ] ← ∏_{c∈C} Count(c, σ, ζ)
11     else if φ has unassigned variable then
12         l ← pickNextLit(φ, σ)
13         ζ[φ] ← Count(φ ∧ l, σ ∪ {l}, ζ) + Count(φ ∧ l̄, σ ∪ {l̄}, ζ)
14     else ζ[φ] ← 1
15 return ζ[φ]
```

In our VCIS heuristics, we add a coefficient impact score for each variable as an equal-weighted additional component to prior variable decision heuristics adopted from GPMC. The coefficient impact score for a variable x is given by $\left(\sum_{j\in\Omega_x} b_x^j/k_j\right) \div |\Omega_x|$ where Ω_x is the set of constraints that variable x appears in, b_x^j is the coefficient of x in constraint j, and k_j is the degree of constraint j. At the start of the counting process, the score for each variable is computed, and variables are decided in descending order of scores in the counting process. In addition, the phase preferences of variable decisions are set such that the term coefficient with the largest coefficient impact score is applied, this is reflected in line 12 of Algorithm 2. The intuition is that branching on a variable with a larger score would have a greater impact on satisfiability gaps than that on a variable with a smaller score.

3.3 Caching Scheme and Optimization

After computing the count of each component, we store it in the cache to avoid redundant computations when the same component is encountered in a different branch of the counting process. A component in our caching scheme contains the following information for unique identification – 1) the set of unassigned variables in the component, 2) the set of yet-to-be-satisfied PB constraints in the component, and 3) their current gaps. In our implementation, we stored variables and constraints of a component by their IDs.

▶ **Lemma 2.** *Given a PB formula F and partial assignments σ and σ'. Let component c be that of $F|_\sigma$ and c' be that of $F|_{\sigma'}$. If $VarIDs(c) = VarIDs(c')$, $CstrIDs(c) = CstrIDs(c')$, and each constraint ω in c satisfies $gap(\omega, \sigma) = gap(\omega, \sigma')$, then $c = c'$.*

Proof. It is clear that $Vars(c) = Vars(c')$. For each constraint ω with $CstrID(\omega) \in CstrIDs(c)$, ω_σ and $\omega_{\sigma'}$ appear in c and c', respectively. As $Vars(c) = Vars(c')$, the left-hand side of ω_σ is the same as that of $\omega_{\sigma'}$. Then we know $\omega_\sigma = \omega_{\sigma'}$ as $gap(\omega, \sigma) = gap(\omega, \sigma')$. ◀

In our implementation, we cache the variable and constraint IDs in ascending order, thereby only needing to record the first ID and the differences between each i^{th} and $i + 1^{th}$. Since a difference is often smaller than an ID, we can use fewer bits to record each component.

Since each gap s is positive, we can record it as $s-1$. Additionally, we do not record gap for a clausal constraint, that is a constraint with coefficients and degree 1. Such a scheme is particularly efficient for the constraints with a lot of literals.

As a cache entry optimization trick to increase the cache hit rate, we perform saturation on the gaps during the generation of component cache IDs. The intuition for cache gap saturation comes from existing literature on PB solving where saturation is performed on the coefficients of a PB constraint [7] i.e. a term ax in the PB constraint is modified to kx where k is the degree when $a > k$. Given a PB constraint $\sum_{j=1}^{m} a_j x_j \geq s$ where s is the gap, our caching scheme changes s in the cache ID to $\min a_j$ over all coefficients a_j of unassigned variables when $0 < s < \min a_j$. This is sound as assigning any of the remaining literals to true would lead to the constraint being satisfied which is the same outcome without modification of s i.e. the resulting gap being 0 or negative. As such the set of satisfying assignments in both cases, with and without modification of s, are the same.

4 Experiments

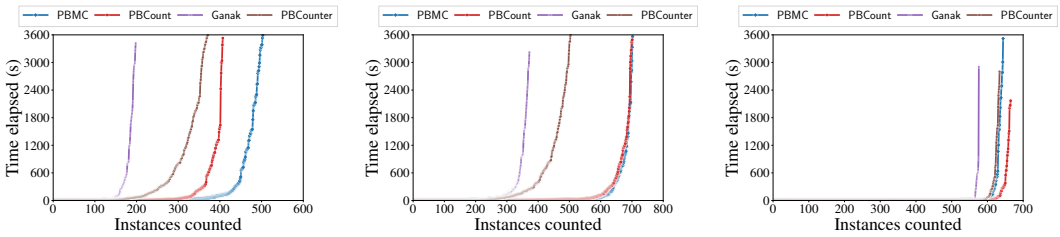
4.1 Setup

We implemented our top-down PB model counter, PBMC, in C++, building on methodologies introduced in the GPMC CNF model counter and RoundingSat PB solver. We performed comparisons to the existing state-of-the-art PB model counters PBCount [19] and PBCounter [13], and also model counting competition (MCC2024) winner Ganak [17]. For consistency, we employed the same benchmark suite as prior work [19]. The suite comprises of 3500 instances across three benchmark sets for PB model counting – *Auction* (1000 instances), *M-dim Knapsack* (1000 instances), and *Sensor placement* (1500 instances). We ran experiments on a cluster with AMD EPYC 7713 CPUs with 1 core and 16 GB memory for each instance and a timeout of 3600 seconds.

Through our evaluations, we seek to understand the performance of our top-down exact PB model counter PBMC. In particular, we investigate the following research questions:
RQ 1: What is the performance of PBMC compared to baselines?
RQ 2: What is the performance impact of our VCIS heuristic in Section 3.2?
RQ 3: What is the performance impact of our cache entry optimization in Section 3.3?

4.2 RQ 1: Performance of Top-down PB Counter PBMC

We compared PBMC against the state-of-the-art PB model counter PBCount, which has a bottom-up design, to understand how well a top-down design would perform. The results are shown in Table 1 and the cactus plot of runtime is shown in Figure 1.



(a) Auction. **(b)** $\mathcal{M}$-dim Knapsack. **(c)** Sensor placement.

Figure 1 Runtime cactus plots of Ganak, PBCount, PBCounter, and PBMC with 3600s timeout. A point on the plot indicates the respective counter could return counts for x instances in y time.

Table 1 Number of benchmark instances counted by Ganak, PBCount, PBCounter, and PBMC in 3600s, higher is better.

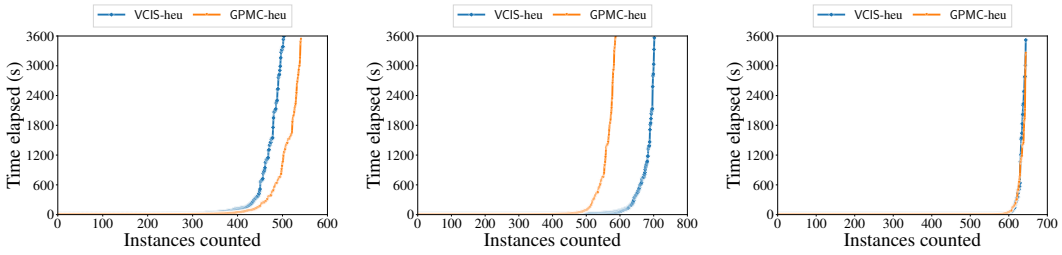
Counters	Auction	$\mathcal{M}$-dim knapsack	Sensor placement	Total
Ganak	198	372	576	1164
PBCount	407	701	**665**	1773
PBCounter	371	503	634	1508
PBMC	**503**	**702**	644	**1849**

Overall, PBMC is able to return counts for 1849 instances, 76 instances more than that of PBCount, 341 instances more than PBCounter, and 685 instances more than Ganak. More specifically, PBMC leads in *auction* and *M dim-knapsack* benchmark sets and came in second in the *sensor placement* benchmark set. In addition, there are 104 instances that could only be counted by PBMC.

It is worth noting that the *sensor placement* benchmark instances all have coefficients of 1 or -1, and that for each instance the degree of all but one PB constraint is 1. We also compared PBMC and PBCount on a modified version of *sensor placement*, whereby coefficients are not all 1 or -1 by accounting for potential redundancy requirements and varying costs associated with different sensor placement locations. On the cost-aware sensor placement instances PBMC outperforms PBCount by returning counts for 715 instances whereas PBCount could only return for 678 instances.

4.3 RQ 2: Performance Impact of VCIS Heuristics

We conducted further experiments to understand the impact of our VCIS heuristic as introduced in Section 3.2. In particular, we compared our implementation of PBMC with VCIS against a baseline version of PBMC that uses the existing variable decision heuristics from the GPMC model counter. The results are shown in Table 2 and the runtime cactus plot is shown in Figure 2.



(a) Auction. **(b)** $\mathcal{M}$-dim Knapsack. **(c)** Sensor placement.

Figure 2 Runtime cactus plots of PBMC with VCIS heuristic and GPMC variable decision heuristics (PBMC-GHeu) with 3600s timeout.

Table 2 Number of PB benchmark instances counted by PBMC, using VCIS and GPMC variable decision heuristics in 3600s.

Counters	Auction	$\mathcal{M}$-dim knapsack	Sensor placement	Total
GPMC-heu	**541**	587	644	1772
VCIS-heu	503	**702**	644	**1849**

In the evaluations, PBMC with our VCIS variable decision heuristic (VCIS-heu) is able to return counts for 1849 benchmark instances in total, substantially more than the 1772 benchmark instances when PBMC is coupled with the existing heuristics from GPMC (GPMC-heu). In particular, VCIS-heu returned counts for 115 more instances than GPMC-heu in $\mathcal{M}$-*dim knapsack* benchmarks, while losing out 38 instances in *Auction* benchmarks. Both heuristics performed the same in *sensor placement* benchmarks. This set of evaluations highlights the tremendous impact of variable decision ordering in the context of PB model counting and demonstrates the performance benefits of our VCIS variable decision heuristic.

4.4 RQ 3: Performance Impact of Cache Entry Optimization

We conducted additional experiments to understand the impact of our optimization for our caching scheme mentioned in Section 3.3, that is modifying the gap value to the smallest coefficient of remaining unassigned variables. We compared PBMC to a version with cache entry optimization disabled (denoted as PBMC-NCS). We show the number of benchmark instances each configuration could count in Table 3. The result shows that our cache entry optimization technique has a positive impact on the performance of PBMC, although less significant than VCIS heuristics in the previous set of experiments.

Table 3 Number of PB benchmark instances counted by PBMC with (PBMC) and without (PBMC-NCS) cache entry optimization in 3600s.

Counters	Auction	$\mathcal{M}$-dim knapsack	Sensor placement	Total
PBMC-NCS	500	700	644	1844
PBMC	**503**	**702**	644	**1849**

4.5 Summary

In **RQ 1**, PBMC demonstrated a significant lead over state-of-the-art competitors. In the process, we observed that PBMC performs better on PB instances with coefficients that are not just 1, as evident in the sensor placement benchmark set. In **RQ 2**, we showed the need for heuristics to consider PB-specific features such as coefficients through the significant performance improvement that VCIS enabled. We also observed that the top-down approach is rather sensitive to variable decision heuristics. Finally, in **RQ 3** we showed that by simply optimizing cache entries, one can get noticeable performance improvements.

5 Conclusion and Future Work

In this work, we explored the top-down design for model counters in the context of pseudo-Boolean model counting. In particular, we introduced our prototype PB model counter PBMC which made use of methodologies from PB solvers and CNF counters as well as our VCIS heuristics to demonstrate leading performance over the existing state-of-the-art PB counter PBCount. PBMC highlights the promising performance of the top-down counter design in the context of PB model counting. It is worth noting that the current performance lead of top-down PB counter PBMC over bottom-up PB counter (PBCount) is not as significant as witnessed in the case of top-down vs bottom-up CNF counters. This could be perhaps partly due to the fact that top-down counters are rather sensitive to variable decision ordering, especially in the context of PB formulas where coefficients and degrees have an impact on

overall performance, as shown in our evaluations. Therefore, it would be of interest to further study variable decision ordering heuristics in the context of PB formulas as future work, because existing heuristics from CNF model counting literature might not be ideal as they do not account for PB-specific features. In addition, it would also make sense to develop techniques to select the appropriate variable decision heuristic based on the given input. Another topic of interest would be preprocessing techniques for the top-down PB counter design, similar to CNF model counting where the top-down counters incorporated various preprocessing techniques which led to significant performance improvements over the years. We hope this study increases the visibility of PB model counting, inspires better counter designs, and more applications of PB model counting.

References

1 R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *International Conference on Computer Aided Design*, 1993.

2 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986. doi:10.1109/TC.1986.1676819.

3 Jeffrey M. Dudek, Vu Hoang Nguyen Phan, and Moshe Y. Vardi. Addmc: Weighted model counting with algebraic decision diagrams. In *AAAI Conference on Artificial Intelligence*, 2020.

4 Jeffrey M. Dudek, Vu Hoang Nguyen Phan, and Moshe Y. Vardi. Dpmc: Weighted model counting by dynamic programming on project-join trees. In *International Conference on Principles and Practice of Constraint Programming*, 2020.

5 Jeffrey M. Dudek, Vu Hoang Nguyen Phan, and Moshe Y. Vardi. Procount: Weighted projected model counting with graded project-join trees. In *International Conference on Theory and Applications of Satisfiability Testing*, 2021.

6 Leonardo Dueñas-Osorio, Kuldeep S. Meel, Roger Paredes, and Moshe Y. Vardi. Counting-based reliability estimation for power-transmission grids. In *AAAI Conference on Artificial Intelligence*, 2017.

7 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2018.

8 Linus Feiten, Matthias Sauer, Tobias Schubert, Alexander Czutro, Eberhard Böhl, Ilia Polian, and Bernd Becker. #SAT-based vulnerability analysis of security components – A case study. *2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 49–54, 2012. doi:10.1109/DFT.2012.6378198.

9 Mohimenul Kabir, Supratik Chakraborty, and Kuldeep S. Meel. Exact asp counting with compact encodings. *AAAI Conference on Artificial Intelligence*, 2024.

10 Mohimenul Kabir, Flavio Everardo, Ankit Shukla, Markus Hecher, Johannes Klaus Fichte, and Kuldeep S. Meel. Approxasp – A scalable approximate answer set counter. In *AAAI Conference on Artificial Intelligence*, 2022.

11 Tuukka Korhonen and Matti Järvisalo. Integrating tree decompositions into decision heuristics of propositional model counters. In *International Conference on Principles and Practice of Constraint Programming*, 2021.

12 Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In *International Joint Conference on Artificial Intelligence*, 2017.

13 Yong Lai, Zhenghang Xu, and Minghao Yin. Pbcounter: weighted model counting on pseudo-boolean formulas. *Frontiers of Computer Science*, 19:193402, 2024. doi:10.1007/S11704-024-3631-1.

14 Daniel Le Berre, Pierre Marquis, Stefan Mengel, and Romain Wallon. Pseudo-boolean constraints from a knowledge representation perspective. In *International Joint Conference on Artificial Intelligence*, 2018.

15 Kenji Hashimoto Ryosuke Suzuki and Masahiko Sakai. Improvement of projected model-counting solver with component decomposition using sat solving in components. In *JSAI Technical Report*, March 2017.

16 Tian Sang, Paul Beame, and Henry A. Kautz. Performing bayesian inference by weighted model counting. In *AAAI Conference on Artificial Intelligence*, 2005.

17 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. Ganak: A scalable probabilistic exact model counter. In *Proceedings of International Joint Conference on Artificial Intelligence*, August 2019.

18 Jiong Yang and Kuldeep S. Meel. Engineering an efficient pb-xor solver. In *International Conference on Principles and Practice of Constraint Programming*, 2021.

19 Suwei Yang and Kuldeep S. Meel. Engineering an exact pseudo-boolean model counter. In *Proceedings of the 38th Annual AAAI Conference on Artificial Intelligence*, 2024.

20 Suwei Yang and Kuldeep S. Meel. Towards projected and incremental pseudo-boolean model counting. In *Proceedings of the 39th Annual AAAI Conference on Artificial Intelligence*, 2025.

Efficient Certified Reasoning for Binarized Neural Networks

Jiong Yang
Georgia Institute of Technology, Atlanta, GA, USA

Yong Kiam Tan
Institute for Infocomm Research (I^2R), A*STAR, Singapore
Nanyang Technological University, Singapore

Mate Soos
University of Toronto, Canada

Magnus O. Myreen
Chalmers University of Technology, Gothenburg, Sweden
University of Gothenburg, Sweden

Kuldeep S. Meel
Georgia Institute of Technology, Atlanta, GA, USA
University of Toronto, Canada

Abstract

Neural networks have emerged as essential components in safety-critical applications – these use cases demand complex, yet trustworthy computations. Binarized Neural Networks (BNNs) are a type of neural network where each neuron is constrained to a Boolean value; they are particularly well-suited for safety-critical tasks because they retain much of the computational capacities of full-scale (floating-point or quantized) deep neural networks, but remain compatible with satisfiability solvers for qualitative verification and with model counters for quantitative reasoning. However, existing methods for BNN analysis suffer from either limited scalability or susceptibility to soundness errors, which hinders their applicability in real-world scenarios.

In this work, we present a scalable and trustworthy approach for both qualitative and quantitative verification of BNNs. Our approach introduces a native representation of BNN constraints in a custom-designed solver for qualitative reasoning, and in an approximate model counter for quantitative reasoning. We further develop specialized proof generation and checking pipelines with native support for BNN constraint reasoning, ensuring trustworthiness for all of our verification results. Empirical evaluations on a BNN robustness verification benchmark suite demonstrate that our certified solving approach achieves a 9× speedup over prior certified CNF and PB-based approaches, and our certified counting approach achieves a 218× speedup over the existing CNF-based baseline. In terms of coverage, our pipeline produces fully certified results for 99% and 86% of the qualitative and quantitative reasoning queries on BNNs, respectively. This is in sharp contrast to the best existing baselines which can fully certify only 62% and 4% of the queries, respectively.

2012 ACM Subject Classification Theory of computation → Logic and verification; Computing methodologies → Neural networks; Computing methodologies → Artificial intelligence

Keywords and phrases Neural network verification, proof certification, SAT solving, approximate model counting

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.32

Related Version *arXiv Version:*: `https://arxiv.org/abs/2507.02916`

Supplementary Material
Software (CryptoMiniSat-BNN): `https://github.com/msoos/cryptominisat`
Software (FRAT-xor-bnn and cake_xlrup-BNN): `https://github.com/meelgroup/frat-xor`
Software (ApproxMC-BNN): `https://github.com/meelgroup/approxmc`
Software (ApproxMCCert-BNN and CertCheck-BNN): `https://github.com/meelgroup/approxmc-cert`



© Jiong Yang, Yong Kiam Tan, Mate Soos, Magnus O. Myreen, and Kuldeep S. Meel;
licensed under Creative Commons License CC-BY 4.0



Text (Proof Format): https://github.com/meelgroup/frat-xor/blob/main/format.md
Dataset (Benchmark): https://doi.org/10.5281/zenodo.15630195

Funding This project was supported in part by Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference [RGPIN-2024-05956].
Yong Kiam Tan: Singapore NRF Fellowship Programme NRF-NRFF16-2024-0002.
Magnus O. Myreen: Swedish Research Council grant 2021-05165.

Acknowledgements We sincerely thank the anonymous reviewers for their constructive feedback on our earlier draft of this paper. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore https://www.nscc.sg, and Niagara supercomputer at the SciNet HPC Consortium. SciNet is funded by Innovation, Science and Economic Development Canada; the Digital Research Alliance of Canada; the Ontario Research Fund: Research Excellence; and the University of Toronto.

1 Introduction

Neural networks have had a transformative impact on fields such as image recognition [31], natural language processing [15], and decision making [42], achieving remarkable success in tackling complex tasks in each of those domains. This success has led to their potential deployment in critical scenarios, such as autonomous driving [9], aircraft collision avoidance [29], and drug discovery [50]. However, despite the impressive performance of neural networks, they often exhibit unexpected behaviors and their lack of explainability makes them difficult to control [45]; this has led to significant concerns about their use in high-stakes applications.

Verification techniques for neural networks can help address some of these safety and security concerns [6, 25, 30, 51, 53]. Here, we focus on verification for *binarized neural networks* (BNNs) [26, 27, 38], i.e., where the input/output of every neuron is quantized to a single bit. Our focus is motivated by two practical reasons. First, BNNs are computationally attractive for real-world, resource-constrained use cases – the absence of floating-point arithmetic in BNNs eliminates floating-point issues, and the bit-level representation enables faster inference speeds and more compact memory layouts [23, 26, 37, 40, 58]. Second, existing studies have demonstrated that BNNs can be verified against quantitative specifications, which is beyond the reach of verification methods for more general classes of neural networks [5].

To elaborate on the latter point, the Boolean nature of BNNs allows both the networks and their specifications to be encoded as Boolean formulas which, in turn, enables the use of off-the-shelf SAT solvers and model counters to verify those specifications. For example, prior research has explored both qualitative and quantitative reasoning for BNN robustness, susceptibility to Trojan attacks, and fairness specifications [5, 27, 38, 55]. Unfortunately, existing combinatorial solving methods for such analyses suffer from limited scalability – they apply only to toy-sized BNNs with hundreds of neurons and a few layers. In contrast, custom methods based on abstraction and bounds propagation for verifying input-output specifications scale to much larger (and more general) neural networks [30, 51, 53], but are susceptible to errors; in the annual neural network verification competitions, participating tools have been repeatedly shown to produce incorrect conclusions. *Certified reasoning* [36], where a tool generates both a conclusion and an independently-checkable proof of that conclusion, has long become a mainstay of trustworthiness for modern SAT solvers [24, 32, 46, 52] but it remains nascent for neural network verification tools [53].

In this work, we address the challenges of scalability and trustworthiness in BNN verification by developing an efficient and certified approach for both qualitative and quantitative reasoning on BNNs. Our contributions are as follows.

- For qualitative reasoning, we integrate a native representation of BNN constraints into a modern CNF-XOR SAT solver to enhance its BNN solving efficiency.
- We then extend the solver's associated UNSAT proof format and verified proof checker with native support for BNN constraints, together enabling an efficient solving and certification pipeline for CNF-XOR-BNN formulas.
- Building upon this pipeline, we further develop a certified approximate model counter for quantitative reasoning over BNNs, leveraging native XOR and BNN representations for both effective reasoning and certification.

Empirically, our approach achieves state-of-the-art certified performance in both qualitative and quantitative reasoning for BNNs. Notably, we observe that the compactness of proof certificates with native BNN proof steps enables fast, verified proof checking.

- For qualitative reasoning, our end-to-end approach produced certified answers for 99% of the benchmark UNSAT queries, achieving a 9× speedup over alternative CNF- and PB-based approaches.
- For quantitative reasoning, our certified counting approach answered 86% of the queries, with 218× speedup over the baseline which could only fully certify 4% of the queries.

By developing solving and certification in tandem, our work offers a trustworthy approach to BNN verification with promising scalability. In fact, during the development of our certification pipeline, we identified and fixed a bug in our solver's implementation of the watching scheme for BNN constraints, which highlights the practical importance of certification.

The rest of this paper is organized as follows. Section 2 presents the preliminaries on BNNs used throughout the paper. Section 3 discusses further background and related work. Section 4 introduces the proposed certified solving and counting approaches for BNN verification, and Section 5 evaluates their empirical performance against existing methods. We summarize our findings in Section 6.

2 Preliminaries: Binarized Neural Networks

Given a set of Boolean variables $\{x_1, x_2, \ldots, x_n\}$, a *literal* l is either a variable x or its negation $\neg x$. A *clause* C is a disjunction of $k > 0$ literals: $C \coloneqq l_1 \vee l_2 \vee \ldots \vee l_k$. A *Conjunctive Normal Form (CNF) formula* φ is a conjunction of $m > 0$ clauses: $\varphi \coloneqq C_1 \wedge C_2 \wedge \ldots \wedge C_m$. A *solution* to φ is an assignment of truth values to the variables such that φ evaluates to `True`. The formula φ is *satisfiable* if at least one solution exists, and *unsatisfiable* otherwise. The *model count* of φ refers to the total number of solutions.

Binarized Neural Networks (BNNs) are a class of neural networks in which each neuron is constrained to a Boolean value. Given an input tensor $\mathbf{x} \in \{0,1\}^n$, an output tensor $\mathbf{y} \in \{0,1\}^m$, a weight matrix $\mathbf{w} \in \{-1,1\}^{n \times m}$, and a bias vector $\mathbf{b} \in \mathbb{Z}^m$, a BNN layer maps $\mathbf{x}$ to $\mathbf{y}$ as follows:

$$\mathbf{y} = \mathsf{sign}\left(\mathbf{w}^\top \mathbf{x} + \mathbf{b}\right)$$

Here, the sign function represents the non-linear sign activation used in BNNs, functioning as a binarized analogue of the ReLU activation. A fully connected BNN consists of several layers laid out in sequence, i.e., the output of one layer is the input of the next, and so on; they can be trained using the straight-through estimator, which enables gradient flow through the sign function during backpropagation. For additional details on BNN training and architecture, we refer the readers to [26].

A BNN constraint corresponds to a neuron in the network, encoding the logical relationship between its input neurons and its output. Formally, we define this relationship as follows:

▶ **Definition 1** (BNN Constraint). *Let y denote the output of a given neuron, and let $x_1, x_2, \ldots, x_n$ denote the values of its n input neurons. Let w_i represent the weight associated with the connection from the i-th input neuron, and b denote the bias term. Then, the logical input-output relationship of a BNN constraint is defined as:*

$$y \leftrightarrow \sum_{i=1}^{n} w_i x_i + b \geq 0 \tag{1}$$

where y and x_i are Boolean variables; w_i and b are constants with $w_i \in \{1, -1\}$ for $i \in [1..n]$ and $b \in \mathbb{Z}$.

We model the value of each binarized neuron as a Boolean variable (in contrast to $y, x_i \in \mathbb{R}$ in conventional neural networks). Some prior work represents neuron values using $\{-1, 1\}$; the two notations are interchangeable via a simple linear transformation. In this work, we adopt the Boolean representation for clarity and compatibility with SAT-based reasoning. The greater-than-or-equal operator in Equation 1 encodes the sign function.

A BNN constraint can be readily transformed into an equivalent *conditional cardinality constraint*, which is defined below. In the rest of this paper, we use the terms BNN constraint and conditional cardinality constraint interchangeably when the context is clear.

▶ **Definition 2** (Conditional Cardinality Constraint). *Given literals $l_1, l_2, \ldots, l_n$, an output literal l_y, and an integer constant k, a conditional cardinality constraint is defined as:*

$$l_y \leftrightarrow \sum_{i=1}^{n} l_i \geq k \tag{2}$$

where l_y represents the truth value of the cardinality constraint $\sum_{i=1}^{n} l_i \geq k$, and $l_1, l_2, \ldots, l_n$ are referred as the left-hand-side (LHS) literals.

Using the above constraint representations, a BNN with t layers, input bits $\mathbf{l}^0$, and output bits $\mathbf{l}^t$ can be represented by a Boolean formula $\text{BNN}(\mathbf{l}^0, \mathbf{l}^t)$. An input-output specification of the BNN is then a Boolean formula $\text{Prop}\left(\mathbf{l}^0, \mathbf{l}^t\right)$. For instance, a common safety specification is the robustness property, which assesses either the existence (qualitative reasoning) or the number (quantitative reasoning) of adversarial inputs. This property is defined as follows:

▶ **Definition 3** (BNN Robustness). *A BNN is ε-robust for the input $\mathbf{l}_g^0$ and output $\mathbf{l}_g^t$ if there are no satisfying assignments for the following property; $\|\cdot\|_1$ denotes the Hamming distance.*

$$\text{Prop}\left(\mathbf{l}^0, \mathbf{l}^t\right) \coloneqq \|\mathbf{l}^0 - \mathbf{l}_g^0\|_1 \leq \varepsilon \ \wedge \ \mathbf{l}^t \neq \mathbf{l}_g^t$$

The Boolean nature of BNNs and their property specifications makes it possible to leverage existing solvers and model counters for answering both qualitative and quantitative robustness queries [5, 27, 38, 55]. However, due to the inherent statistical nature of neural network training, the existence of adversarial inputs is expected, making purely qualitative queries potentially uninformative. Quantitatively estimating the prevalence of adversarial inputs allows us to draw statistically significant conclusions [5]. Moreover, the absence of floating-point arithmetic in the formulation of BNNs eliminates floating-point issues when answering these critical robustness queries.

3 Related Work

Neural Network Verification. Neural network verification has received growing attention, as reflected in the organization of annual benchmark competitions [10]. Existing techniques span several categories, including linear bound propagation (e.g., α, β-CROWN [51]), reachability analysis (e.g., CORA [1], NNV [35], nnenum [4], NeVer2 [13], and PyRAT [33]), and SMT-based solving (e.g., Marabou [53], NeuralSAT [17], and Reluplex [30]). These approaches focus on qualitative reasoning, particularly the robustness of ReLU networks [22], and often rely on commercial solvers such as Gurobi or MATLAB. However, no fully verified proof checker currently exists for traditional neural network verification frameworks [14]. Another line of research targets the verification of BNNs. This includes qualitative reasoning for robustness [27, 38], quantitative reasoning for robustness, susceptibility to Trojan attacks, and fairness specifications [5, 55], efforts to accelerate verification through increased sparsity in BNNs [27, 39], and techniques for networks with both binarized and non-binarized neurons [2].

BNN Encodings. With the conditional cardinality constraint representation, BNN constraints can be further encoded into clauses via cardinality encodings or represented directly as PB constraints. Encoding BNNs into CNF formulas results in a substantial increase in formula size, primarily due to the large number of conditional cardinality constraints. For a BNN with m neurons, where each neuron is represented using Equation 1 with n Boolean variables, the CNF encoding introduces $\mathcal{O}(n)$ variables and clauses per neuron, leading to a total of $\mathcal{O}(nm)$ variables and clauses. As a result, the formula size can easily reach millions of variables and clauses even for networks with only a few hundred neurons, which poses significant challenges for existing SAT solvers and model counters. The complete CNF encoding of BNNs is presented in [5, 38], where BNNs are first encoded as conditional cardinality constraints and subsequently translated into clauses using cardinality encodings. A PB-based encoding for BNNs can be found in [55].

Propagation and Conflict Analysis for BNN Constraints. To avoid the cumbersome clause-based encoding of Equation 1, Jia and Rinard introduced an alternative representation of BNN constraints in their solver, MiniSatCS, which enables efficient detection of propagation and conflicts specific to BNNs [27]. MiniSatCS transforms Equation 1 into a reified cardinality constraint, which is logically equivalent to the conditional cardinality constraint in Equation 2. Propagation and conflict detection over Equation 2 are handled as follows.

- **Operand-inferring:** If l_y is assigned `True` and $n-k$ literals among $l_1, \ldots, l_n$ are already assigned `False`, then the remaining unassigned l_i must be assigned `True`; otherwise, a conflict is detected. Conversely, if l_y is assigned `False` and $k-1$ literals among $l_1, \ldots, l_n$ are assigned `True`, then the remaining unassigned literals must be assigned `False`.
- **Target-inferring:** If k literals among $l_1, \ldots, l_n$ are assigned `True`, then l_y must be assigned `True`; otherwise, a conflict is detected. Similarly, if $n-k+1$ literals among $l_1, \ldots, l_n$ are assigned `False`, then l_y must be assigned `False`.

When a conflict is detected, the responsible literals are added to the conflict clause. This design eliminates the need to encode Equation 1 into large sets of clauses, significantly reducing formula size. The native support for direct propagation and conflict detection allows for faster solving and scalability to networks with thousands of neurons. However, reified cardinality constraints are not supported by the UNSAT proof format used in standard SAT solvers, rendering the results uncertifiable without further developments.

Certified Solving and Counting. Certification has become a cornerstone of trustworthiness in the SAT community [24, 32, 46, 52]. Recent efforts have extended certification to the CNF-XOR solver, CryptoMiniSat and even to probabilistic counting [49]. The model counting problem aims to determine the number of satisfying assignments for a Boolean formula, while *approximate model counting* seeks to compute a high-quality approximation of this count; approximate model counters have shown practical performance in many applications [5, 16, 21]. The state-of-the-art algorithm, ApproxMC [12, 56], provides a $(1+\varepsilon)$ approximation of the model count with confidence at least $1-\delta$, given tolerance ε and confidence δ. To ensure correctness, a certification pipeline has been developed for ApproxMC [49], enabling independent verification of its output. In parallel, recent work has also begun to explore *certified exact model counting* [11, 20].

4 Certified Reasoning for BNNs

This section presents our integration of native BNN constraints for BNN solving, counting, and proof checking. Section 4.1 introduces our new solver, CryptoMiniSat-BNN, which incorporates native support for BNN constraints to enable efficient qualitative reasoning on BNNs; it also describes our proof format with native support for BNN reasoning in unsatisfiability proofs. Section 4.2 describes our certified model counter, ApproxMCCert-BNN, which similarly leverages native BNN constraint support for efficient and certified quantitative reasoning. Finally, Section 4.3 concludes with a case study of a subtle bug in the watching scheme of CryptoMiniSat-BNN, uncovered by our certification pipeline – this highlights the complementary role of certification in ensuring the correctness of automated reasoning tools.

4.1 Certified Solving for Qualitative BNN Reasoning

Our new solver efficiently reasons over both XOR and BNN constraints, collectively referred to as CNF-XOR-BNN formulas; support for XOR constraints is essential for CNF-BNN model counting (which will be presented in Section 4.2). We formally define CNF-XOR-BNN formulas as follows:

▶ **Definition 4** (CNF-XOR-BNN Formula). *Given a set of clauses $C_1, C_2, \ldots, C_m$, XOR constraints $XOR_1, XOR_2, \ldots, XOR_t$, and BNN constraints $BNN_1, BNN_2, \ldots, BNN_s$, a CNF-XOR-BNN formula φ is defined as the conjunction of these constraints:*

$$\varphi \coloneqq \bigwedge_{i=1}^{m} C_i \wedge \bigwedge_{j=1}^{t} XOR_j \wedge \bigwedge_{k=1}^{s} BNN_k$$

A solution (or model) ω for φ is a Boolean assignment to variables that simultaneously satisfies all of the clauses, XORs, and BNN constraints.

4.1.1 Solving CNF-XOR-BNN Formulas

Input format. Figure 1 (left) shows an example of a CNF-XOR-BNN formula in our input format, which extends the standard DIMACS format [28] to support XOR and BNN constraints. In the header, the first integer specifies the number of variables, and the second indicates the total number of constraints, including clauses, XORs, and BNNs. Each line beginning with the prefix **b** represents a BNN constraint, while lines starting with the prefix **x** denote XOR constraints. The last line in Figure 1 (left) encodes a BNN constraint corresponding to $x_1 + \neg x_2 + x_3 \geq 2 \leftrightarrow x_4$. The list of integers following the prefix **b**,

CNF-XOR-BNN formula

```
p cnf 4 5
1 -2 0
-1 3 0
x 1 -2 -3 0
-4 0
b 1 -2 3 0 2 4 0
```

FRAT-XOR-BNN proof

```
o 1 1 -2 0
o 2 -1 3 0
o x 1 1 -2 -3 0
o 3 -4 0
o b 1 1 -2 3 0 k 2 4 0
i 4 -1 -3 0 b l 1 0 u 3 0
i 5 2 -3 0 b l 1 0 u 3 0
a 6 -3 0 l 1 5 4 0
a 7 -1 0
a 8 -2 0
i 9 1 2 3 0 l 1 0
a 10 0
f 1 1 -2 0
f ...
f 10 0
f x 1 1 -2 -3 0
f b 1 1 -2 3 0 k 2 4 0
```

XLRUP proof

```
o x 1 1 -2 -3 0
i cb 4 -1 -3 0 1 u 3 0
i cb 5 2 -3 0 1 u 3 0
5 d 3 0
6 -3 0 4 1 5 0
6 d 5 4 0
7 -1 0 6 2 0
7 d 2 0
8 -2 0 7 1 0
8 d 1 0
i cx 9 1 2 3 0 1 0
10 0 7 6 9 8 0
```

Figure 1 Example of a CNF-XOR-BNN formula (left) and its unsatisfiability proof in FRAT-XOR-BNN (middle) and XLRUP (right) formats. Green highlights indicate special keywords, while yellow highlights denote constraint indices.

terminated by a 0, specifies the LHS literals of the cardinality constraint, in this case $x_1, \neg x_2$, and x_3. The last two integers, following the terminating 0, indicate the cutoff value and the output literal, respectively. In this example, 2 is the cutoff value, and 4 denotes the output literal x_4. Similarly, the line `x 1 -2 -3 0` encodes an XOR constraint: $x_1 \oplus \neg x_2 \oplus \neg x_3 = 1$.

CNF-XOR-BNN Solving. A CNF-XOR-BNN solver takes a formula φ in the extended DIMACS format and determines its satisfiability. We introduce the first CNF-XOR-BNN solver, CryptoMiniSat-BNN, by extending an existing CNF-XOR solver, CryptoMiniSat [44], with native support for BNN constraints. Our implementation maintains an internal representation of each BNN constraint in the form of a conditional cardinality constraint and directly detects when it causes a unit propagation or conflict. The solver follows the standard Conflict-Driven Clause Learning (CDCL) procedure used in modern SAT solvers while incorporating specialized propagation and conflict detection schemes for XOR and BNN constraints, as outlined in Algorithms 1 and 2.

In Algorithm 1, the unit propagation procedure iteratively processes unit literals stored in trail, starting from the index qhead. For each unit literal l, the solver retrieves its watched constraints in Line 4, which may correspond to either a clause or a BNN constraint; XOR watches are maintained separately. The procedure then iterates over all watched constraints in Lines 5–9. If a clause is watched, clausal unit propagation is performed in Line 7. If a BNN constraint is watched, a specialized BNN propagation procedure, PropagateBnn (Algorithm 2), is executed in Line 9. If no conflict is detected during clausal and BNN unit propagation, Gauss-Jordan elimination is applied to propagate the literal l over XOR constraints and to detect potential conflicts (Line 11). Finally, if no conflict is encountered, the index qhead is incremented and the loop continues.

Algorithm 2 describes the procedure, PropagateBnn(w, l), for unit propagation and conflict clause generation from a BNN constraint w with respect to a literal l. Let n and k denote the number of LHS literals and the cutoff value of the cardinality constraint in w, respectively,

Algorithm 1 Propagate(qhead, trail).

```
1: conflict ← empty clause
2: while qhead < GetSize(trail) and conflict is empty do
3:     l ← trail[qhead]
4:     watches ← GetWatches[¬l]
5:     for w ∈ watches do
6:         if w is a clause then
7:             conflict ← PropagateClause(w, l)
8:         else
9:             conflict ← PropagateBnn(w, l)
10:    if conflict is empty then
11:        conflict ← GaussJordanElim(l)
12:    qhead ← qhead + 1
13: return conflict
```

and let l_y denote the output literal. For each BNN constraint, the solver maintains two counters: the number of literals assigned `True` (trueCount) and the number of unassigned literals (undefCount), to enable prompt detection of unit propagation or conflicts. If l is a positive LHS literal in w, trueCount is incremented and undefCount is decremented (Lines 7–8); if l is a negative LHS literal, undefCount is decremented (Line 10). Lines 11–24 perform unit propagation in a BNN constraint based on the values of trueCount and undefCount. Specifically, Lines 11–20 handle the propagation of LHS literals. If l_y is `True` and the sum of trueCount and undefCount is less than k (Line 12), a conflict is detected because the cardinality constraint cannot be satisfied even if all unassigned LHS literals are assigned `True`, contradicting the assignment $l_y = \texttt{True}$. This situation arises when at least $n - k + 1$ LHS literals are assigned `False`, causing the sum of trueCount and undefCount to fall below k. Consequently, $\neg l_y$ and the $n - k + 1$ `False` LHS literals are added to the conflict clause (Line 13). Alternatively, if the sum of trueCount and undefCount equals k, all unassigned LHS literals must be assigned `True` (Line 15) to satisfy the assignment $l_y = \texttt{True}$. Similarly, conflict detection and assignment inference for the case where l_y is assigned `False` are handled in Lines 16–20. On the other hand, Lines 21–24 handle the propagation of the output literal. If trueCount $\geq k$, meaning that at least k LHS literals are assigned `True`, the cardinality constraint in w is satisfied, and the output literal l_y must be assigned `True` (Line 22). Alternatively, if trueCount + undefCount $< k$, it is impossible to satisfy the cardinality constraint, and l_y must be assigned `False` (Line 24).

Whenever a BNN constraint infers the assignment of a literal, a reason clause is generated to facilitate conflict analysis in the CDCL algorithm. For instance, when l_y is inferred to be `True` in Line 22, the reason clause includes l_y and the negation of the k LHS literals assigned to `True`, thereby explaining the inference of l_y based on these assignments.

Comparison. CryptoMiniSat-BNN differs from MiniSatCS [27] in several key aspects. First, we maintain BNN constraints in the standard AtLeastK form, as defined in Equation 2, whereas MiniSatCS represents them as AtMostK cardinality constraints. Second, our solver generates minimal conflict and reason clause from BNN constraints, while MiniSatCS does not. For instance, when l_y is assigned `False` and at least k LHS literals are assigned `True`, MiniSatCS includes the negation of all `True` LHS literals in the conflict clause, in addition to l_y. In contrast, CryptoMiniSat-BNN adds only the negations of exactly k `True` literals, which

Algorithm 2 PropagateBnn(w, l).

```
1: n ← GetLiteralCount(w)
2: k ← GetCutOff(w)
3: l_y ← GetOutputLiteral(w)
4: trueCount ← GetTrueCount(w)
5: undefCount ← GetUndefCount(w)
6: if l is a positive literal in w then
7:     trueCount = trueCount + 1
8:     undefCount = undefCount − 1
9: else if l is a negative literal in w then
10:    undefCount = undefCount − 1
11: if l_y is True then
12:    if trueCount + undefCount < k then
13:        add ¬l_y and n − k + 1 False literals to conflict.
14:    else if trueCount + undefCount == k then
15:        assign True to all unassigned literals.
16: else if l_y is False then
17:    if trueCount ≥ k then
18:        add l_y and the negations of k True literals to conflict.
19:    else if trueCount + 1 == k then
20:        assign False to all unassigned literals.
21: else if trueCount ≥ k then
22:    assign True to l_y.
23: else if trueCount + undefCount < k then
24:    assign False to l_y.
```

constitute the minimal reason for the conflict. Third, CryptoMiniSat-BNN adopts a unified data structure to consistently maintain watches for both clauses and BNN constraints, whereas MiniSatCS employs separate data structures for clause and BNN watches. Lastly, MiniSatCS is built on top of MiniSat, whereas CryptoMiniSat-BNN integrates the BNN constraints directly in the state-of-the-art CNF-XOR solver, CryptoMiniSat, which incorporates more advanced techniques than MiniSat. The native support for both XOR and BNN constraints in CryptoMiniSat-BNN is crucial for our development of a certified CNF-BNN approximate model counter in Section 4.2.

4.1.2 Certifying CNF-XOR-BNN Solving

Our new pipeline for certified CNF-XOR-BNN unsatisfiability consists of several parts.

1. We extend the FRAT-XOR[1] format [49] by introducing support for BNN-specific reasoning steps, resulting in an extended FRAT-XOR-BNN format that can be readily generated by CryptoMiniSat-BNN.
2. We develop a format elaborator, FRAT-xor-bnn, which converts FRAT-XOR-BNN proofs into an elaborated proof format, XLRUP (eXtended LRUP). This translator extends FRAT-xor [49], originally designed for FRAT-XOR proofs.

[1] The FRAT-XOR format is itself an extension of the FRAT format [3] with support for XOR-related reasoning steps.

3. The XLRUP proof format supports RUP proofs, extended with both XOR- and BNN-specific steps; we design, implement, and formally verify an efficient proof checker for XLRUP proofs (specifically, we add new BNN reasoning support).

Continuing the example from Figure 1, the unsatisfiability proof for its CNF-XOR-BNN formula in both FRAT-XOR-BNN and XLRUP formats is displayed (middle and right columns, respectively); green highlights indicate special keywords, while yellow highlights denote the constraint indices.

FRAT-XOR-BNN format. In the FRAT-XOR-BNN proof format, BNN-related steps are marked with the keyword **b**. The prefix **o b** denotes an original BNN constraint in the input formula, while the prefix **f b** indicates a final BNN constraint that remains after the empty clause is derived. The `clause-from-bnn` step is prefixed with the identifier **i** (denoting implication), followed by the clause implied by a BNN constraint. The keyword **b l** specifies the index of the BNN constraint, while the keyword **u** lists the set of unit clauses used to simply the BNN constraint.

For example, the following step records the derivation of a new clause $(\neg x_1 \vee \neg x_3)$ to be stored at index 4:

```
i 4 -1 -3 0 b l 1 0 u 3 0
```

This clause is implied by the first BNN constraint:

$$x_1 + \neg x_2 + x_3 \geq 2 \leftrightarrow x_4 \tag{3}$$

together with the unit clause $\neg x_4$ at index 3. The chain of unit clauses simplifies the BNN constraint by applying the corresponding variable assignments.

Similarly, the `clause-from-xor` step is prefixed with the identifier **i**, followed by the keyword **l**, which indicates the XOR constraints implying the clause. For instance, the following step records the derivation of a new clause $(x_2 \vee x_1 \vee x_3)$ with index 9:

```
i 9 1 2 3 0 l 1 0
```

This clause is implied by the first XOR constraint:

$$x_1 \oplus \neg x_2 \oplus \neg x_3 = 1 \tag{4}$$

XLRUP format. The XLRUP format also marks BNN-specific steps with the keyword **b** and XOR-specific steps with **x**, with slight modifications in notation. Specifically, the keyword (**i cb**) indicates a `clause-from-bnn` step in XLRUP. For example, the following step records the derivation of the clause $(\neg x_1 \vee \neg x_3)$ with index 4, inferred from the first BNN constraint (Equation 3) and the unit clause indexed by 3:

```
i cb 4 -1 -3 0 1 u 3 0
```

Similarly, the keyword (**i cx**) denotes a `clause-from-xor` step in XLRUP. For instance, the following step records the derivation of the clause $(x_1 \vee x_2 \vee x_3)$ with index 9, inferred from the first XOR constraint (Equation 4):

```
i cx 9 1 2 3 0 1 0
```

Derivation of UNSAT in Figure 1. We use the FRAT-XOR-BNN proof as an example to illustrate the UNSAT derivation for the CNF-XOR-BNN formula shown in Figure 1. In the proof, after recording the original constraints using the keyword **o**, the first new clause $(\neg x_1 \vee \neg x_3)$ is derived by a `clause-from-bnn` step at index 4. This clause is inferred from the BNN constraint (Equation 3) together with the unit clause $\neg x_4$ indexed by 3. At this step, the solver makes a decision on x_3, which triggers propagation of the BNN constraint (Equation 3) as described in Line 20 of Algorithm 2. Specifically, the assignments $\neg x_4$ and x_3 imply a `False` assignment to the remaining literals in the BNN constraint. Consequently, $\neg x_3$ (as the reason) and $\neg x_1$ (as the propagated assignment) are added to the reason clause at step 4. Similarly, the reason clause $(x_2 \vee \neg x_3)$ is derived at step 5 from the same BNN constraint to infer a `True` assignment to x_2.

Subsequently, $\neg x_1$ and x_2 lead to a conflict in the first input clause $(x_1 \vee \neg x_2)$, resulting in the unit learned clause $\neg x_3$ at step 6 through conflict analysis. After backtracking to decision level 0 and applying unit propagation, the assignments $\neg x_1$ and $\neg x_2$ are derived at steps 7 and 8, respectively. Finally, the assignments $\neg x_1, \neg x_2$, and $\neg x_3$ cause a conflict in the XOR constraint (Equation 4), from which the conflict clause $(x_1 \vee x_2 \vee x_3)$ is generated via a `clause-from-xor` step at step 9. The empty clause is then derived at step 10, concluding the proof. The XLRUP proof follows the same reasoning but uses a different syntax.

Proof generation from CryptoMiniSat-BNN. CryptoMiniSat-BNN records `clause-from-bnn` steps in a lazy manner (`clause-from-xor` steps are handled similarly). Specifically, the solver logs conflicts and unit propagations from BNN constraints during the search process and generates only the conflict clauses and relevant reason clauses during conflict analysis, which are recorded as `clause-from-bnn` steps. Additionally, we track the list of unit clauses used to simplify BNN constraints during both pre-processing and in-processing, and incorporate them into `clause-from-bnn` steps following the keyword **u**. We disable the variable replacement technique, which replaces a variable in a BNN constraint with another, as it requires reasoning over a BNN constraint and two clauses (or an XOR constraint).[2] We also disable BNN (and XOR) propagation during the distillation phase [18], as our purely clausal RUP proofs do not support BNN propagation.

FRAT-XOR-BNN to XLRUP through FRAT-xor-bnn. FRAT-xor-bnn follows the *lightweight* design principle of FRAT-xor [49]: it does not verify the correctness of BNN-specific steps but delegates the checking of those steps to a formally verified tool, cake_xlrup-BNN. Our primary modification involves tracking clauses derived from BNN constraints to ensure their correct usage in subsequent clausal steps and for the automatic elaboration of RUP proofs [3].

Formally verified proof checking with cake_xlrup-BNN. Our cake_xlrup-BNN tool extends an earlier proof checker [49] with efficient and verified BNN-specific reasoning. The verification is customized to check `clause-from-bnn` (**i cb**) steps quickly. Briefly speaking, the literals in each BNN constraint are stored in a bitset; whenever a clause C is to be derived, the proof checker tracks propagations for all units from $\neg C$ (and others provided in the proof hint) using the bitset, and accepts the derivation of C if a contradiction is derived. The procedure for checking conflicts is similar to Algorithm 2.

[2] Such replacements can be logged using two clauses, $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$, or an XOR constraint, $(x_1 \oplus x_2 = 0)$ to replace x_1 with x_2. The clause-based replacement is compatible with clause steps, while the XOR-based replacement is compatible with XOR steps. However, neither is compatible with BNN-specific steps in our proof format.

A particularly useful optimization is to keep the size of each bitset minimal for the corresponding BNN constraint. In straightforward numbering schemes for BNN neurons, the variables appearing in the resulting constraints are dense and contiguous, i.e., $x_i, x_{i+1}, \ldots, x_{i+n}$. In such cases, the corresponding bitset will only allocate memory for the indices i to $i+n$. This optimization ensures that the overall representation takes $O(n)$ space where n is the number of neurons in the BNN, as opposed to $O(nv)$, where v is the total number of variables (if the bitset naively stored indices for every possible variable in the formula).

Note that, by design of the XLRUP extensions, our new BNN steps interact smoothly with the existing proof system for clauses and XOR constraints. As with earlier versions, our extensions are formally verified down to machine code implementations using CakeML [47].

4.2 Certified Counting for Quantitative BNN Reasoning

We present our certified model counter, ApproxMCCert-BNN, along with its certificate checker, CertCheck-BNN, designed for efficient quantitative reasoning on BNNs.

4.2.1 Model Counting for CNF-BNN Formulas

Let us start by briefly explaining the key idea behind ApproxMC, a state-of-the-art approximate model counting algorithm and implementation [12, 56]. It repeatedly samples and adds random XOR constraints to halve the solution space of the formula; eventually, after adding a number m of such constraints, the number of remaining solutions n will become sufficiently small. Then, the overall solution count is approximated as $n \cdot 2^m$. The core requirement for an efficient implementation is the use of an incremental CNF-XOR solver to find solutions or determine that no further ones exist.

When the input formula φ contains both clauses and BNN constraints – that is, when φ is a CNF-BNN formula – the same counting algorithm can be used except that a CNF-XOR-BNN solver is required to compute the number of solutions to the combined CNF-XOR-BNN formula $\varphi \wedge XOR_1 \wedge \ldots \wedge XOR_m$.

To this end, we develop ApproxMC-BNN, an approximate model counter built on top of our CNF-XOR-BNN solver, CryptoMiniSat-BNN, introduced earlier. ApproxMC-BNN takes a CNF-BNN formula as input and computes an approximate solution count. The correctness of ApproxMC-BNN is guaranteed by the following theorem; a formalized proof in Isabelle/HOL has established the correctness of ApproxMC for arbitrary Boolean theories [48, 49].

▶ **Theorem 5.** *For a CNF-BNN formula φ, a tolerance parameter ε, and a confidence parameter δ,* ApproxMC-BNN *returns an approximate count c such that*

$$\mathsf{Pr}\left[\frac{|\mathtt{sol}(\varphi)|}{1+\varepsilon} \leq c \leq (1+\varepsilon)|\mathtt{sol}(\varphi)|\right] \geq 1-\delta$$

We emphasize that ApproxMC-BNN is the first model counter that natively supports BNN constraints. This enables it to use a succinct input representation (without extension variables), which significantly improves counting efficiency.

4.2.2 Certifying CNF-BNN Counting

The counting certification pipeline for CNF-BNN formulas follows the same framework as the certified CNF model counter, ApproxMCCert, and its certificate checker, CertCheck [49]. Specifically, the certified approximate CNF-BNN model counter, ApproxMCCert-BNN, takes

a CNF-BNN formula as input and produces an approximate model count along with a certificate, which can be independently verified by the certificate checker, CertCheck-BNN, to ensure correct execution modulo randomness [49].

Most relevant for us, the design of certificates for CertCheck records the number of XOR constraints used, as well as the solutions to the resulting CNF-XOR-BNN formulas, which are essential for estimating the model count. During certificate checking, CertCheck-BNN verifies the correctness of the solutions to the CNF-XOR-BNN formulas and employs CryptoMiniSat-BNN and cake_xlrup-BNN to generate and validate UNSAT proofs, ensuring exhaustive enumeration of solutions. If all solutions and UNSAT proofs pass verification by CertCheck-BNN, it outputs a certified model count; otherwise, an error is reported [49].

Our new CertCheck-BNN tool was built by extending the earlier formal verification. Thanks to the genericity of the theorems proved in prior work [48], the main verification task here was to develop a theory of CNF-BNN formulas in Isabelle/HOL and then instantiate earlier results to derive a correct-by-construction certificate checking framework, including a formalized version of Theorem 5.

4.3 Bug Report from Certification

During the development of our certification pipeline, we identified and resolved a subtle bug in the implementation of CryptoMiniSat-BNN, related to the watching scheme for BNN constraints (Equation 2). As shown in Algorithm 2, CryptoMiniSat-BNN maintains two counters–trueCount (literals assigned `True`) and undefCount (unassigned literals)–to determine the propagation state of a BNN constraint. Each literal is watched according to its polarity: assigning a positive literal increments trueCount and decrements undefCount (Lines 7–8), while assigning a negative literal decrements undefCount (Line 10).

To preserve the AtLeastK form (Equation 2) when the output literal l_y is assigned `False`, CryptoMiniSat-BNN flips the polarity of LHS literals. However, the implementation mistakenly failed to update the polarity of watched literals during this process, resulting in incorrect BNN propagations. This bug was nontrivial to detect through conventional testing but was successfully uncovered by our attempts at running benchmarks through the certification pipeline.

5 Experimental Evaluation

In this section, we evaluate the performance of our tools, namely, our proof-generating BNN-based solver (CryptoMiniSat-BNN), proof checker (cake_xlrup-BNN), certified BNN-based model counter (ApproxMCCert-BNN) and its certificate checker (CertCheck-BNN). Experiments were conducted on a BNN robustness benchmark [5], which consists of 960 problem instances with varying BNN architecture sizes.

For comparison, we also transformed the BNN formulas into CNF and PB encodings and evaluated our tools against state-of-the-art certified solvers and counters. Due to the large number of compared tools, we introduce them in their respective subsections.

Setting. All experiments were conducted on a high-performance computing cluster, where each node is equipped with an AMD EPYC-Milan processor featuring 2×64 physical cores and 512GB of RAM. Each solver or counter was required to produce a checkable proof/certificate alongside its output. For every tool, we set a time limit of 500 seconds for qualitative reasoning and 5,000 seconds for quantitative reasoning, with a memory limit of 16 GB. For approximate counting, we used the default parameter values of $\delta = 0.2$ and $\varepsilon = 0.8$.

We report the PAR-2 score for each tool, a standard evaluation metric used in the SAT competition. The PAR-2 score is the weighted average runtime across all instances, including the actual runtime for successfully completed instances, and double the time limit for instances that exceed the timeout (i.e., 1,000 seconds for qualitative reasoning and 10,000 seconds for quantitative reasoning in our setting).

Our experimental evaluation is designed to address the following research questions:

(RQ1) How does the certified solving performance of CryptoMiniSat-BNN compare to alternative certified approaches using CNF and PB encodings for BNNs?

(RQ2) How does the certified counting performance of ApproxMCCert-BNN compare to the CNF-based baseline for BNNs?

Summary. Overall, our approach significantly improves the runtime performance in both qualitative and quantitative reasoning for BNNs, and makes certified BNN verification practically feasible. Specifically,

(RQ1) CryptoMiniSat-BNN and cake_xlrup-BNN produced fully certified answers for 99% of the qualitative queries, achieving a 9× speedup over alternative CNF and PB approaches, which fully certified only 62% of all queries.

(RQ2) ApproxMCCert-BNN and CertCheck-BNN fully certified results for 86% of the quantitative queries, achieving a 218× speedup over the CNF baseline, which could fully verify only 4% of the queries.

5.1 Certified Qualitative Reasoning for BNNs

This section evaluates (RQ1) the runtime performance and proof-checking efficiency of CryptoMiniSat-BNN and cake_xlrup-BNN in answering qualitative queries for BNNs. The baselines include the state-of-the-art CNF solver CaDiCaL [7] (f13d744) with its formally verified LRAT proof checker cake_lpr [46] (36b917a), and the PB solver RoundingSat [19] (73aaf09) with its proof checker VeriPB [8] (178904a).[3] We selected the 125 UNSAT instances (out of 960) as qualitative BNN verification tasks; certifying satisfiability of the remaining instances is straightforward so we reserve those for qualitative reasoning (counting).

A summary performance comparison is shown in Table 1. From the top row, we observe that CryptoMiniSat-BNN successfully solved all 125 instances, achieving the lowest PAR-2 score of 16. The lower row compares the overall performance of solving and proof checking across our three configurations. All but one of the benchmarks were successfully certified by our approach; notably, although RoundingSat solved 123 instances, only 67 proofs were verified by VeriPB within the time limit. This shows the benefit of our native design for both solving and certification.

Figure 2 (left) plots the cumulative number of solved or checked instances for a given time limit, i.e., each point (x, y) indicates that the tool successfully completed y instances within x seconds. CryptoMiniSat-BNN, together with cake_xlrup-BNN, consistently appears at the top of the figure, providing improved solving and checking performance compared to RoundingSat with VeriPB and CaDiCaL with cake_lpr. For solving times, CryptoMiniSat-BNN is on average 26× faster than CaDiCaL and 3× faster than RoundingSat. In fact, CryptoMiniSat-BNN is also 2× faster than MiniSatCS [27]; however, the latter does not support UNSAT proof

[3] The VeriPB toolchain also has a formally verified checker which we did not run for our evaluation. We opted for the static binary of VeriPB from the SAT competition 2023, as our evaluation showed that the 2024 version performed slightly slower on our problem instances.

Table 1 Runtime performance (time in seconds) for certified qualitative reasoning for BNNs over 125 UNSAT instances. The first row shows solver performance with proof generation, while the second row presents the combined performance of solving and proof checking for various toolchains.

	CaDiCaL	RoundingSat	CryptoMiniSat-BNN
Solved Instances	77	123	**125**
PAR-2 Score	478	72	**16**
	+cake_lpr	+VeriPB	+cake_xlrup-BNN
Solved Instances	77	67	**124**
PAR-2 Score	874	1,068	**60**

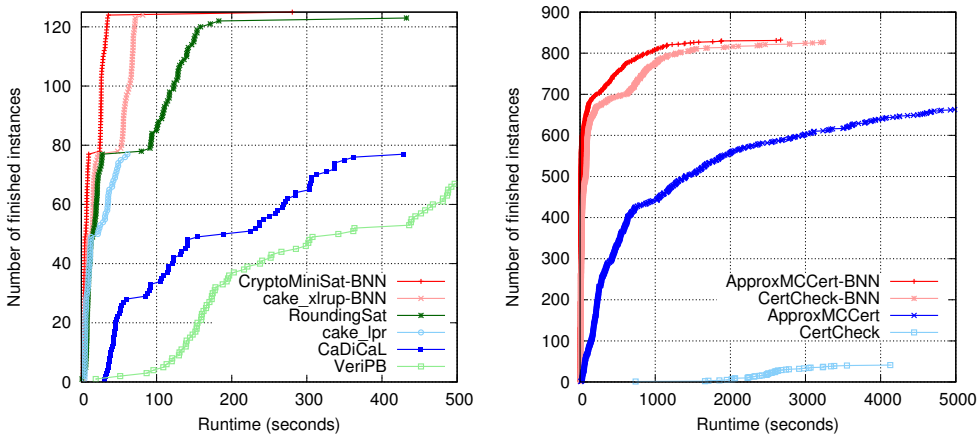


Figure 2 (Left) Runtime performance comparison of solvers and proof checkers on qualitative reasoning benchmarks. (Right) Runtime performance comparison of counters and certificate checkers. Note that the solving/counting and checking times are plotted separately in both plots.

generation. A detailed comparison with MiniSatCS is presented in Appendix A. For combined solving and checking times, CryptoMiniSat-BNN+cake_xlrup-BNN achieved a 9× speedup over CaDiCaL+cake_lpr and an 18× speedup over RoundingSat+VeriPB. The ratios are calculated over the common solved (and checked) instances for each pair.

5.2 Certified Quantitative Reasoning for BNNs

This section evaluates (RQ2) the runtime performance and certificate-checking efficiency of ApproxMCCert-BNN and CertCheck-BNN for quantitative BNN queries. We compare our approach against the certified CNF counter ApproxMCCert and its certificate checker CertCheck [49]. Additionally, we disabled the counting preprocessor Arjun [43] due to its poor performance on BNN benchmarks. Our evaluation is conducted on the full 960 instances of the quantitative robustness benchmark for BNNs [5].

Table 2 presents the runtime performance of the counters and their corresponding certificate checkers. In terms of counting performance (with certificate generation), we observed a significant improvement where ApproxMCCert-BNN solved 169 more instances and more than halved the PAR-2 score to 1,443. For the combined performance of counting and certificate checking, ApproxMCCert together with CertCheck could produce fully certified

Table 2 Runtime performance (time in seconds) for certified quantitative reasoning for BNNs over 960 counting instances. The first two columns show ApproxMCCert's counting performance, followed by the combined counting and certificate checking performance with CertCheck; the latter two columns similarly show the performance of ApproxMCCert-BNN together with CertCheck-BNN.

	ApproxMCCert	+CertCheck	ApproxMCCert-BNN	+CertCheck-BNN
Finished	663	41	**832**	**827**
PAR-2 Score	3,778	19,256	**1,443**	**3,068**

results for only 4% of the instances (41 out of 960) within the time limit. In contrast our approach, ApproxMCCert-BNN+CertCheck-BNN, completed both counting and certificate checking for 86% of the instances (827 out of 960), and substantially reduced the PAR-2 score from 19,256 to 3,068.

Figure 2 (right) shows the number of instances completed by the counters and checkers over time. Here, ApproxMCCert-BNN consistently solved the largest number of instances and it is closely trailed by CertCheck-BNN – thus, ApproxMCCert-BNN offers superior counting runtime and its generated certificates were also readily checked by CertCheck-BNN. In terms of average counting and certification times, ApproxMCCert-BNN+CertCheck-BNN achieved a 218× speedup over ApproxMCCert+CertCheck on their common, fully certified instances.

Upon deeper investigation, a key issue for the certificates of CertCheck is that the CNF encodings of BNN benchmarks lead to *projected* counting instances with an extremely large number of extension variables. The counting certificate format must record satisfying assignments for all variables, which leads to substantial file size, memory, and timing overheads in the CNF-based approach.

In fact, ApproxMCCert-BNN outperforms even existing counters that do not provide certification for quantitative reasoning on BNNs, such as the exact CNF counter Ganak [41], the exact PB counters PBCount [57], and the approximate PB counter ApproxMC-PB [55]. A detailed comparison is provided in Appendix B.

6 Conclusion

Certified verification is critical for neural networks deployed in safety-critical applications. This work presents an efficient certified solver for qualitative reasoning and a certified approximate model counter for quantitative reasoning on BNNs. Our approach significantly outperforms existing CNF and PB-based methods and produces fully certified results for the vast majority of queries, making certified BNN verification practically feasible. Looking forward, this framework opens the door to certifying large-scale binarized vision and language models [23, 58], as well as extending certification to quantized neural networks for efficient on-device deployment [34, 54].

References

1 Matthias Althoff. An introduction to CORA 2015. In Goran Frehse and Matthias Althoff, editors, *ARCH@CPSWeek*, volume 34 of *EPiC Series in Computing*, pages 120–151. EasyChair, 2015. doi:10.29007/ZBKV.

2 Guy Amir, Haoze Wu, Clark W. Barrett, and Guy Katz. An SMT-based approach for verifying binarized neural networks. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *TACAS*, volume 12652 of *LNCS*, pages 203–222. Springer, 2021. doi:10.1007/978-3-030-72013-1_11.

3 Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. *Log. Methods Comput. Sci.*, 18(2), 2022. doi:10.46298/LMCS-18(2:3)2022.

4 Stanley Bak. nnenum: Verification of ReLU neural networks with optimized abstraction refinement. In Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz, and Ivan Perez, editors, *NFM*, volume 12673 of *LNCS*, pages 19–36. Springer, 2021. doi:10.1007/978-3-030-76384-8_2.

5 Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel, and Prateek Saxena. Quantitative verification of neural networks and its security applications. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *CCS*, pages 1249–1264. ACM, 2019. doi:10.1145/3319535.3354245.

6 Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *NIPS*, pages 2613–2621, 2016. URL: https://proceedings.neurips.cc/paper/2016/hash/980ecd059122ce2e50136bda65c25e07-Abstract.html.

7 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *CAV*, volume 14681 of *LNCS*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

8 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.

9 Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016. arXiv:1604.07316.

10 Christopher Brix, Stanley Bak, Taylor T. Johnson, and Haoze Wu. The fifth international verification of neural networks competition (VNN-COMP 2024): Summary and results. *CoRR*, abs/2412.19985, 2024. doi:10.48550/arXiv.2412.19985.

11 Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Certified knowledge compilation with application to formally verified model counting. *J. Artif. Intell. Res.*, 82, 2025. doi:10.1613/JAIR.1.15958.

12 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In Christian Schulte, editor, *CP*, volume 8124 of *LNCS*, pages 200–216. Springer, 2013. doi:10.1007/978-3-642-40627-0_18.

13 Stefano Demarchi, Dario Guidotti, Luca Pulina, and Armando Tacchella. NeVer2: learning and verification of neural networks. *Soft Comput.*, 28(19):11647–11665, 2024. doi:10.1007/S00500-024-09907-5.

14 Remi Desmartin, Omri Isac, Grant O. Passmore, Kathrin Stark, Ekaterina Komendantskaya, and Guy Katz. Towards a certified proof checker for deep neural network verification. In Robert Glück and Bishoksan Kafle, editors, *LOPSTR*, volume 14330 of *LNCS*, pages 198–209. Springer, 2023. doi:10.1007/978-3-031-45784-5_13.

15 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *NAACL-HLT*, pages 4171–4186. Association for Computational Linguistics, 2019. doi:10.18653/V1/N19-1423.

16 Leonardo Dueñas-Osorio, Kuldeep S. Meel, Roger Paredes, and Moshe Y. Vardi. Counting-based reliability estimation for power-transmission grids. In Satinder Singh and Shaul Markovitch, editors, *AAAI*, pages 4488–4494. AAAI Press, 2017. doi:10.1609/AAAI.V31I1.11178.

17 Hai Duong, Dong Xu, ThanhVu Nguyen, and Matthew B. Dwyer. Harnessing neuron stability to improve DNN verification. *Proc. ACM Softw. Eng.*, 1(FSE):859–881, 2024. doi:10.1145/3643765.

18 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005. doi:10.1007/11499107_5.

19 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In Jérôme Lang, editor, *IJCAI*, pages 1291–1299. ijcai.org, 2018. doi:10.24963/IJCAI.2018/180.

20 Johannes Klaus Fichte, Markus Hecher, and Valentin Roland. Proofs for propositional model counting. In Kuldeep S. Meel and Ofer Strichman, editors, *SAT*, volume 236 of *LIPIcs*, pages 30:1–30:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.SAT.2022.30.

21 Andreas Gittis, Eric Vin, and Daniel J. Fremont. Randomized synthesis for diversity and cost constraints with control improvisation. In Sharon Shoham and Yakir Vizel, editors, *CAV*, volume 13372 of *LNCS*, pages 526–546. Springer, 2022. doi:10.1007/978-3-031-13188-2_26.

22 Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík, editors, *AISTATS*, volume 15 of *JMLR Proceedings*, pages 315–323. JMLR.org, 2011. URL: http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf.

23 Yefei He, Zhenyu Lou, Luoming Zhang, Jing Liu, Weijia Wu, Hong Zhou, and Bohan Zhuang. BiViT: Extremely compressed binary vision transformers. In *ICCV*, pages 5628–5640. IEEE, 2023. doi:10.1109/ICCV51070.2023.00520.

24 Marijn Heule, Warren A. Hunt Jr., Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *ITP*, volume 10499 of *LNCS*, pages 269–284. Springer, 2017. doi:10.1007/978-3-319-66107-0_18.

25 Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In Rupak Majumdar and Viktor Kuncak, editors, *CAV*, volume 10426 of *LNCS*, pages 3–29. Springer, 2017. doi:10.1007/978-3-319-63387-9_1.

26 Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *NIPS*, pages 4107–4115, 2016. URL: https://proceedings.neurips.cc/paper/2016/hash/d8330f857a17c53d217014ee776bfd50-Abstract.html.

27 Kai Jia and Martin C. Rinard. Efficient exact verification of binarized neural networks. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *NeurIPS*, 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/1385974ed5904a438616ff7bdb3f7439-Abstract.html.

28 David S. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. DIMACS/AMS, 1996. doi:10.1090/DIMACS/026.

29 Kyle D. Julian, Jessica Lopez, Jeffrey S. Brush, Michael P. Owen, and Mykel J. Kochenderfer. Policy Compression for Aircraft Collision Avoidance Systems. In *DASC*, pages 1–10, 2016.

30 Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kuncak, editors, *CAV*, volume 10426 of *LNCS*, pages 97–117. Springer, 2017. doi:10.1007/978-3-319-63387-9_5.

31 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *NIPS*, pages 1106–1114, 2012. URL: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

32 Peter Lammich. Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.*, 64(3):513–532, 2020. doi:10.1007/S10817-019-09525-Z.

33 Augustin Lemesle, Julien Lehmann, and Tristan Le Gall. Neural network verification with PyRAT. *CoRR*, abs/2410.23903, 2024. doi:10.48550/arXiv.2410.23903.

34 Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: activation-aware weight quantization for on-device LLM compression and acceleration. In Phillip B. Gibbons, Gennady Pekhimenko, and Christopher De Sa, editors, *MLSys*. mlsys.org, 2024. URL: https://proceedings.mlsys.org/paper_files/paper/2024/hash/42a452cbafa9dd64e9ba4aa95cc1ef21-Abstract-Conference.html.

35 Diego Manzanas Lopez, Sung Woo Choi, Hoang-Dung Tran, and Taylor T. Johnson. NNV 2.0: The neural network verification tool. In Constantin Enea and Akash Lal, editors, *CAV*, volume 13965 of *LNCS*, pages 397–412. Springer, 2023. doi:10.1007/978-3-031-37703-7_19.

36 Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Comput. Sci. Rev.*, 5(2):119–161, 2011. doi:10.1016/J.COSREV.2010.09.009.

37 Duncan J. M. Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit K. Mishra, Debbie Marr, Suchit Subhaschandra, and Philip Heng Wai Leong. High performance binary neural networks on the xeon+fpga™ platform. In Marco D. Santambrogio, Diana Göhringer, Dirk Stroobandt, Nele Mentens, and Jari Nurmi, editors, *FPL*, pages 1–4. IEEE, 2017. doi:10.23919/FPL.2017.8056823.

38 Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. Verifying properties of binarized deep neural networks. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *AAAI*, pages 6615–6624. AAAI Press, 2018. doi:10.1609/AAAI.V32I1.12206.

39 Nina Narodytska, Hongce Zhang, Aarti Gupta, and Toby Walsh. In search for a SAT-friendly binarized neural network architecture. In *ICLR*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=SJx-j64FDr.

40 Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet classification using binary convolutional neural networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *ECCV*, volume 9908 of *LNCS*, pages 525–542. Springer, 2016. doi:10.1007/978-3-319-46493-0_32.

41 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In Sarit Kraus, editor, *IJCAI*, pages 1169–1176. ijcai.org, 2019. doi:10.24963/IJCAI.2019/163.

42 David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016. doi:10.1038/NATURE16961.

43 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, *ICCAD*, pages 71:1–71:9. ACM, 2022. doi:10.1145/3508352.3549406.

44 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *SAT*, volume 5584 of *LNCS*, pages 244–257. Springer, 2009. doi:10.1007/978-3-642-02777-2_24.

45 Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In Yoshua Bengio and Yann LeCun, editors, *ICLR*, 2014. URL: http://arxiv.org/abs/1312.6199.

46 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. Verified propagation redundancy and compositional UNSAT checking in CakeML. *Int. J. Softw. Tools Technol. Transf.*, 25(2):167–184, 2023. doi:10.1007/S10009-022-00690-Y.

47 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019. doi:10.1017/S0956796818000229.

48 Yong Kiam Tan and Jiong Yang. Approximate model counting. *Archive of Formal Proofs*, March 2024. , Formal proof development. URL: https://isa-afp.org/entries/Approximate_Model_Counting.html.

49 Yong Kiam Tan, Jiong Yang, Mate Soos, Magnus O. Myreen, and Kuldeep S. Meel. Formally certified approximate model counting. In Arie Gurfinkel and Vijay Ganesh, editors, *CAV*, volume 14681 of *LNCS*, pages 153–177. Springer, 2024. doi:10.1007/978-3-031-65627-9_8.

50 Bie Verbist, Günter Klambauer, Liesbet Vervoort, Willem Talloen, Ziv Shkedy, Olivier Thas, Andreas Bender, Hinrich W. H. Göhlmann, Sepp Hochreiter, and the QSTAR Consortium. Using transcriptomics to guide lead optimization in drug discovery projects: Lessons learned from the QSTAR project. *Drug Discovery Today*, 20(5):505–513, May 2015.

51 Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. *CoRR*, abs/2103.06624, 2021. arXiv:2103.06624.

52 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, *SAT*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014. doi:10.1007/978-3-319-09284-3_31.

53 Haoze Wu, Omri Isac, Aleksandar Zeljic, Teruhiro Tagomori, Matthew L. Daggitt, Wen Kokke, Idan Refaeli, Guy Amir, Kyle Julian, Shahaf Bassan, Pei Huang, Ori Lahav, Min Wu, Min Zhang, Ekaterina Komendantskaya, Guy Katz, and Clark W. Barrett. Marabou 2.0: A versatile formal analyzer of neural networks. In Arie Gurfinkel and Vijay Ganesh, editors, *CAV*, volume 14682 of *LNCS*, pages 249–264. Springer, 2024. doi:10.1007/978-3-031-65630-9_13.

54 Guangxuan Xiao, Ji Lin, Mickaël Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *ICMS*, volume 202 of *Proceedings of Machine Learning Research*, pages 38087–38099. PMLR, 2023. URL: https://proceedings.mlr.press/v202/xiao23c.html.

55 Jiong Yang and Kuldeep S. Meel. Engineering an efficient PB-XOR solver. In Laurent D. Michel, editor, *CP*, volume 210 of *LIPIcs*, pages 58:1–58:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CP.2021.58.

56 Jiong Yang and Kuldeep S. Meel. Rounding meets approximate model counting. In Constantin Enea and Akash Lal, editors, *CAV*, volume 13965 of *LNCS*, pages 132–162. Springer, 2023. doi:10.1007/978-3-031-37703-7_7.

57 Suwei Yang and Kuldeep S. Meel. Engineering an exact pseudo-Boolean model counter. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan, editors, *AAAI*, pages 8200–8208. AAAI Press, 2024. doi:10.1609/AAAI.V38I8.28660.

58 Yichi Zhang, Ankush Garg, Yuan Cao, Lukasz Lew, Behrooz Ghorbani, Zhiru Zhang, and Orhan Firat. Binarized neural machine translation. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *NeurIPS*, 2023. URL: http://papers.nips.cc/paper_files/paper/2023/hash/bd1fc5cbedfe4d90d0ac2d23966fa27e-Abstract-Conference.html.

Table 3 Runtime performance comparison of CryptoMiniSat-BNN and MiniSatCS over 960 instances.

	MiniSatCS	CryptoMiniSat-BNN
Solved Instances	960	960
PAR-2 Score	3	1

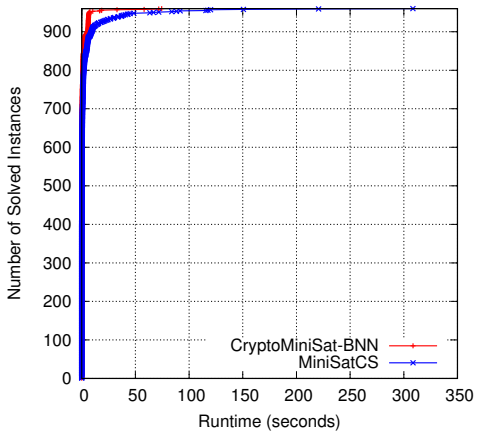


Figure 3 Runtime performance comparison of CryptoMiniSat-BNN and MiniSatCS.

A Solving Performance Comparison with MiniSatCS

We present a performance comparison between CryptoMiniSat-BNN and MiniSatCS [27], excluding proof generation, which is not supported by MiniSatCS. Our evaluation is conducted on the full set of 960 instances from the quantitative robustness benchmark for BNNs [5], including both SAT and UNSAT cases. We set a time limit of 500 seconds and a memory limit of 4GB for each instance.

As shown in Table 3, both CryptoMiniSat-BNN and MiniSatCS solved all instances. However, CryptoMiniSat-BNN achieved a lower PAR-2 score of 1, compared to 3 for MiniSatCS. Figure 3 illustrates the number of solved instances over time. CryptoMiniSat-BNN solved all instances within 75 seconds, whereas MiniSatCS required up to 309 seconds. Overall, CryptoMiniSat-BNN achieved a 2× speedup over MiniSatCS.

B Counting Performance Comparison

We present the counting performance comparison between ApproxMC-BNN and state-of-the-art CNF counters, Ganak [41] and ApproxMC [56], and PB counters, PBCount [57] and ApproxMC-PB [55] in Table 4; ApproxMC-BNN refers to ApproxMCCert-BNN without certification generation. We also repeat the numbers for ApproxMCCert-BNN for comparison. Each counter uses a memory limit of 4GB.

The CNF counters demonstrated the worst performance. The exact and approximate counter, Ganak and ApproxMC solved only 424 and 770 out of 960 instances, respectively, with the two highest two PAR-2 scores in the table. The approximate PB counter, ApproxMC-PB solved 816 instances and lowered the PAR-2 score to 1,741. PBCount failed to solve any instance so we removed it from the table. Lastly, ApproxMC-BNN achieved the best perform-

Table 4 Counting performance comparison over 960 instances.

	Ganak	ApproxMC	ApproxMC-PB	ApproxMC-BNN	ApproxMCCert-BNN
Counted Instances	424	770	816	**868**	832
PAR-2 Score	6248	2600	1741	**1128**	1,443

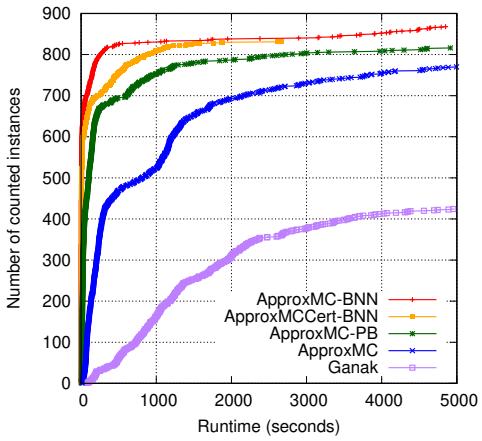


Figure 4 Runtime performance comparison between counters.

ance with 868 instances solved and further reduced the PAR-2 score to 1,128. ApproxMC-BNN solved 98 more instances than the best CNF counter (ApproxMC) and outperformed the best PB counter ApproxMC-PB by 52 more solved instances. Even with the overhead of certificate generation, ApproxMCCert-BNN still outperforms other non-BNN-native approaches.

Figure 4 compares the counting performance in terms of counted instances per runtime. The plot shows that ApproxMC-BNN consistently solves the most instances at any time limit, followed by ApproxMC-PB, ApproxMC, and Ganak.