31st International Conference on Principles and Practice of Constraint Programming

CP 2025, August 10–15, 2025, Glasgow, Scotland

Edited by

Maria Garcia de la Banda

LIPIcs – Vol. 340 – CP 2025 www.dagstuhl.de/lipics

Editors

Maria Garcia de la Banda
Monash University, Clayton, Australia
maria.garciadelabanda@monash.edu

ACM Classification 2012
Theory of computation → Constraint and logic programming; Applied computing → Operations research; Mathematics of computing → Combinatorial algorithms; Mathematics of computing → Combinatorial optimization; Mathematics of computing → Solvers; Computing methodologies → Combinatorial algorithms; Computing methodologies → Optimization algorithms; Computing methodologies → Planning and scheduling; Computing methodologies → Theorem proving algorithms

ISBN 978-3-95977-380-5

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/978-3-95977-380-5.

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.CP.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
Maria Garcia de la Banda 0:ix

List of Authors
..... 0:xi

Senior Program Committee
..... 0:xvii

Program Committee
..... 0:xix

Additional Reviewers
..... 0:xxi

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

Solving the Agile Earth Observation Satellite Scheduling Problem with CP and Local Search
Valentin Antuori, Damien T. Wojtowicz, and Emmanuel Hebrard 3:1–3:22

Conflict Analysis Based on Cutting-Planes for Constraint Programming
Robbin Baauw, Maarten Flippo, and Emir Demirović 4:1–4:19

Transition Dominance in Domain-Independent Dynamic Programming
J. Christopher Beck, Ryo Kuroiwa, Jimmy H. M. Lee, Peter J. Stuckey, and Allen Z. Zhong 5:1–5:23

Modeling and Explaining an Industrial Workforce Allocation and Scheduling Problem
Ignace Bleukx, Ryma Boumazouza, Tias Guns, Nadine Laage, and Guillaume Poveda 6:1–6:24

Optimizing 2D Cutting: A Bin Packing Approach to Minimize Scraps and Maximize Their Reusability
Manuel Chastenay, Xavier Zwingmann, Claude-Guy Quimper, and Jonathan Gaudreault 7:1–7:21

Breaking Symmetries with Involutions
Michael Codish and Mikoláš Janota 8:1–8:17

31st International Conference on Principles and Practice of Constraint Programming (CP 2025).
Editor: Maria Garcia de la Banda
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Constraint Models for Klondike
Nguyen Dang, Ian P. Gent, Peter Nightingale, Felix Ulrich-Oltean, and Jack Waller 9:1–9:20

Unit Types for MiniZinc
Jip J. Dekker, Jason Nguyen, Peter J. Stuckey, and Guido Tack 10:1–10:20

Cargo Routing Optimization in Liner Shipping Networks
Yousra El Ghazi, Djamal Habet, and Cyril Terrioux 11:1–11:18

An Expansion-Based Approach for Quantified Integer Programming
Michael Hartisch and Leroy Chew 12:1–12:26

Disjunctive Scheduling in Tempo
Emmanuel Hebrard 13:1–13:22

Guess and Prove: A Hybrid Approach to Linear Polynomial Recovery in Circuit Verification
Clemens Hofstadler and Daniela Kaufmann 14:1–14:22

Symmetric Core Learning for Pseudo-Boolean Optimization by Implicit Hitting Sets
Hannes Ihalainen, Jeremias Berg, Matti Järvisalo, and Bart Bogaerts 15:1–15:26

PrintTalk: A Language for Constraint-Based 3D Modelling
Jef Jacobs, Wolfgang De Meuter, and Jens Nicolay 16:1–16:22

Scalable Counting of Minimal Trap Spaces and Fixed Points in Boolean Networks
Mohimenul Kabir, Van-Giang Trinh, Samuel Pastva, and Kuldeep S Meel 17:1–17:26

Greed Is Slow on Sparse Graphs of Oriented Valued Constraints
Artem Kaznatcheev and Sofia Vazquez Alferez 18:1–18:13

Analyzing Self-Stabilization of Synchronous Unison via Propositional Satisfiability
Asma Khoualdia, Sami Cherif, Stéphane Devismes, and Léo Robert 19:1–19:21

Dependency-Curated Large Neighbourhood Search
Frej Knutar Lewander, Pierre Flener, and Justin Pearson 20:1–20:17

Practically Feasible Proof Logging for Pseudo-Boolean Optimization
Wietze Koops, Daniel Le Berre, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Yong Kiam Tan, and Marc Vinyals 21:1–21:27

Learning to Bound for Maximum Common Subgraph Algorithms
Buddhi W. Kothalawala, Henning Koehler, and Qing Wang 22:1–22:18

RPID: Rust Programmable Interface for Domain-Independent Dynamic Programming
Ryo Kuroiwa and J. Christopher Beck 23:1–23:21

The Work Task Variation Problem
Mikael Z. Lagerkvist and Magnus Rattfeldt 24:1–24:23

Aircraft Resource-Constrained Assembly Line Balancing with Learning Effect: A Constraint Programming Approach
Duc Anh Le, Stéphanie Roussel, and Christophe Lecoutre 25:1–25:24

Parallel MIP Solving with Dynamic Task Decomposition
Peng Lin, Shaowei Cai, Mengchuan Zou, and Shengqi Chen 26:1–26:19

Understanding the Impact of Value Selection Heuristics in Scheduling Problems
Tim Luchterhand, Emmanuel Hebrard, and Sylvie Thiébaux 27:1–27:23

SLS-Enhanced Core-Boosted Linear Search for Anytime Maximum Satisfiability
Ole Lübke and Jeremias Berg 28:1–28:20

From Prediction to Action: A Constraint-Based Approach to Predictive Policing
Younes Mechqrane and Ismail Elabbassi 29:1–29:18

Exact Methods for the Travelling Salesperson Problem with Self-Deleting Graphs
Daniel Pekar and J. Christopher Beck 30:1–30:19

Transformer-Based Feature Learning for Algorithm Selection in Combinatorial Optimisation
Alessio Pellegrino, Özgür Akgün, Nguyen Dang, Zeynep Kiziltan, and Ian Miguel 31:1–31:22

BFS-Based Canonical Codes for Generating Graphs with Constraint Programming
Xiao Peng and Christine Solnon 32:1–32:16

Constraint-Based In-Station Train Dispatching
Andreas Schutt, Matteo Cardellini, Jip J. Dekker, Daniel Harabor, Marco Maratea, and Mauro Vallati 33:1–33:24

DynamicSAT: Dynamic Configuration Tuning for SAT Solving
Zhengyuan Shi, Wentao Jiang, Xindi Zhang, Jin Luo, Yun Liang, Zhufei Chu, and Qiang Xu 34:1–34:23

Unite and Lead: Finding Disjunctive Cliques for Scheduling Problems
Konstantin Sidorov, Imko Marijnissen, and Emir Demirović 35:1–35:24

Balancing Latin Rectangles with LLM-Generated Streamliners
Florentina Voboril, Vaidyanathan Peruvemba Ramaswamy, and Stefan Szeider ... 36:1–36:17

Multi-League Sports Scheduling with Team Interdependencies: An Optimization Model
Nils Weidmann 37:1–37:19

Reducing Quantum Circuit Synthesis to #SAT
Dekel Zak, Jingyi Mei, Jean-Marie Lagniez, and Alfons Laarman 38:1–38:21

The 3-Decomposition Conjecture: A SAT-Based Approach with Specialized Propagators
Tianwei Zhang and Stefan Szeider 39:1–39:19

An Efficient and Uniform CSP Solution Generator Generator
Ghiles Ziat and Martin Pépin 40:1–40:18

Short Papers

Modeling and Solving a Composite Structure Design Problem with Constraint Programming
Miguel Antoons, Augustin Delecluse, Samih Zein, and Pierre Schaus 41:1–41:9

Towards Modern and Modular SAT for LCG
Jip J. Dekker, Alexey Ignatiev, Peter J. Stuckey, and Allen Z. Zhong 42:1–42:12

Scheduling Telescope Observations for the European Southern Observatory
Michael Prümm, Peter Nightingale, and Felix Ulrich-Oltean 43:1–43:10

Preface

This volume contains the proceedings of the 31st International Conference on Principles and Practice of Constraint Programming (CP 2025), which took place at the University of Glasgow in Scotland, from the 10th to the 15th of August 2025. Held annually, CP is the premier international conference on constraint programming. This year CP is co-located with the 28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025), the 18th International Symposium on Combinatorial Search (SoCS 2025), and the 23rd International Workshop on Satisfiability Modulo Theories (SMT 2025). More details about the CP 2025 conference can be found at `https://cp2025.a4cp.org/`.

CP 2025 received a total of 106 submissions of original unpublished research papers. After an initial screening, 101 papers were sent out for double-blind peer review – a process where authors and reviewers are anonymous to each other. Each paper received at least three reviews from members of the Program Committee and from several additional reviewers recruited whenever extra expertise was required, all under the guidance of a member of the Senior Program Committee. Authors had the opportunity to answer the questions posed by the reviewers in their reviews during an author response phase. This was followed by an extensive discussion of each paper among the reviewers and the member of the Senior Program Committee assigned to the paper. After the discussion, the Senior Program Committee member wrote a meta-review which summarised the discussion, the pros and cons of the paper, and either made a recommendation to accept/reject the paper or marked it as borderline. The latter generated extra discussions with me as Chair, sometimes leading to a request for additional Senior Program Committee members to provide their recommendations. At the end of this process, 41 papers were accepted for presentation at the conference and publication in these proceedings. Of these, 38 are regular length papers while 3 are short ones; 25 are technical papers while 16 are application ones; and 18 of the 41 are student papers, that is, papers where the first author is a student who contributed significantly more than any other author in the paper.

The Senior Program Committee nominated several papers for the best paper awards, and two sub-committees from the Senior Program Committee were organised to select the winners. These subcommittees awarded the Best Technical Paper Award to J. Christopher Beck, Ryo Kuroiwa, Jimmy H. M. Lee, Peter J. Stuckey, and Allen Z. Zhong for their paper "Transition Dominance in Domain-Independent Dynamic Programming", the Best Application Paper Awards to Ignace Bleukx, Ryma Boumazouza, Tias Guns, Nadine Laage, and Guillaume Poveda for their paper "Modeling and Explaining an Industrial Workforce Allocation and Scheduling Problem", and the Best Student Paper Award to Hannes Ihalainen, Jeremias Berg, Matti Järvisalo, and Bart Bogaerts for their paper "Symmetric Core Learning for Pseudo-Boolean Optimization by Implicit Hitting Sets".

It is traditional for CP to dedicate an entire day before the accepted papers are presented to a series of workshops where the latest ideas on emerging/focus areas are discussed, and to the doctoral program where PhD students come together to exchange ideas and network both with their peers and with senior researchers. This year, as part of the co-location with SAT, we shared these important activities and thus extended the program to dedicate two entire days to them. The selection and organisation of the workshops was chaired by Zeynep Kiziltan (CP) and Marc Vinyals (SAT), while the doctoral program was chaired by Mun See Chang (CP) and Katalin Fazekas (SAT). My thanks to all four for their wonderful work. See `https://cp2025.a4cp.org/workshops.html` for details on the workshops and `https://satcpdp25.github.io/` for the doctoral program.

31st International Conference on Principles and Practice of Constraint Programming (CP 2025).
Editor: Maria Garcia de la Banda

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

The co-location also gave us an opportunity to share invited speakers selected by their expertise on cutting-edge topics of interest to all three communities. Each conference selected one, with CP having the pleasure of inviting Christine Solnon to speak about "Anytime and exact search for planning problems: How to explore a DP-based state transition graph with A*, CP and LS?". The abstract of this talk, as well as the one for the talk selected by SAT (Ruzica Piskac, on "Privacy-preserving SAT Solving") are included in these proceedings, while the one for the talk selected by SoCS (Sylvie Thiebaux on "Graph Learning for Planning") did not make it on time to be published. I am really looking forward to all three! We also continued the CP tradition of offering tutorials that aim to disseminate insights on emerging topics and trends, provide surveys of complementary techniques to those commonly studied in CP, or inform industrial practitioners on state-of-the-art developments in the CP field. This year two tutorials were selected: "Modelling for Scheduling and Rostering Problems with Constraint Programming" by Helmut Simonis, and "Parallel Constraint Solving" by Shaowei Cai and Dominik Schreiber. My heartfelt thanks to the tutorial chair Andre Cire who organised the selection process and selected the final two. I would also like to welcome Mohamed Siala who joins Özgür Akgün as joint DEI chair, a role Mohamed will hold for two years. Together they organised the DEI event that is becoming traditional in our conference series.

A conference like CP relies on the support of many people to make it a success. First, I would like to thank everybody who contributed to the submitted papers, as without them there would be no conference. Then, of course, I would like to thank the Program Committee, the additional reviewers, and the Senior Program Committee for their careful reviews, meta-reviews and subsequent discussions. It was a pleasure to see engaging, respectful and insightful discussions where reviewers were willing to consider other points of view and even change their mind when presented with the right argument. A special thanks goes to the members of the Senior Program Committee for supporting many of my tasks, from proposing extra members for the Program Committee to helping me select the invited talk, and for providing their advice on the process to follow when non-existing citations were detected in a couple of the submitted papers. In addition, I would like to thank the Association for Constraint Programming (ACP) for its trust and for making the CP conference series possible, with a more specific thank to the ACP president Gilles Pesant, and the ACP conference coordinator Hélène Verhaeghe.

Last, but not least, I would like to thank everyone involved in the local conference organisation chaired by Ciaran McCreesh and Blair Archibald, whose great work both on the physical site and on the website made the conference possible, as well as our sponsorship chair, Ruth Hoffmann who secured the generous support of our sponsors. I am writing this two months before CP 2025 starts, but I am sure the conference venue and organization will be fabulous, and I am really looking forward to again be among my colleagues discussing the many exciting ideas in the invited talks, papers, workshops, tutorials and student posters. I cannot wait!

Maria Garcia de la Banda

List of Authors

Özgür Akgün (31)
School of Computer Science,
University of St Andrews, Scotland

Miguel Antoons (41)
UCLouvain, Belgium

Valentin Antuori (3)
LAAS – CNRS, Toulouse, France

Robbin Baauw (4)
Delft University of Technology, The Netherlands

J. Christopher Beck (5, 23, 30)
Department of Mechanical and Industrial
Engineering, University of Toronto, Canada

Jeremias Berg (15, 28)
University of Helsinki, Finland

Ignace Bleukx (6)
Department of Computer Science,
KU Leuven, Belgium

Bart Bogaerts (15)
KU Leuven, Belgium;
Vrije Universiteit Brussel, Belgium

Ryma Boumazouza (6)
Airbus SAS, Institut de Recherche
Technologique Saint-Exupery, Toulouse, France

Shaowei Cai (26)
Key Laboratory of System Software (Chinese
Academy of Sciences) and State Key Laboratory
of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing, China;
School of Computer Science and Technology,
University of Chinese Academy of Sciences,
Beijing, China

Matteo Cardellini (33)
DIBRIS, University of Genova, Italy

Manuel Chastenay (7)
Lab-Usine – Joint Research Unit for Advanced
Manufacturing, Université Laval, Québec,
Canada;
SOKÏO Industrie, Saint-Augustin-de-Desmaures,
Canada

Shengqi Chen (26)
Department of Computer Science and
Technology, Tsinghua University, Beijing, China

Sami Cherif (19)
MIS UR 4290, Université de Picardie Jules
Verne, Amiens, France

Leroy Chew (12)
TU Wien, Austria

Zhufei Chu (34)
Ningbo University, China

Michael Codish (8)
Department of Computer Science, Ben-Gurion
University of the Negev, Beer-Sheva, Israel

Nguyen Dang (9, 31)
School of Computer Science,
University of St Andrews, UK

Wolfgang De Meuter (16)
Software Languages Lab,
Vrije Universiteit Brussel, Belgium

Jip J. Dekker (10, 33, 42)
Department of Data Science and Artificial
Intelligence, Monash University, Clayton,
Australia;
ARC Training Centre in Optimisation
Technologies, Integrated Methodologies, and
Applications (OPTIMA), Melbourne, Australia

Augustin Delecluse (41)
TRAIL, ICTEAM, UCLouvain, Belgium

Emir Demirović (4, 35)
Delft University of Technology, The Netherlands

Stéphane Devismes (19)
MIS UR 4290, Université de Picardie Jules
Verne, Amiens, France

Yousra El Ghazi (11)
Aix Marseille Univ, CNRS, LIS,
Marseille, France

Ismail Elabbassi (29)
Ai movement – International Artificial
Intelligence Center of Morocco, University
Mohammed VI Polytechnic, Rabat, Morocco

Pierre Flener (20)
Department of Information Technology,
Uppsala University, Sweden

Maarten Flippo (4)
Delft University of Technology, The Netherlands

31st International Conference on Principles and Practice of Constraint Programming (CP 2025).
Editor: Maria Garcia de la Banda
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Jonathan Gaudreault (7)
Lab-Usine – Joint Research Unit for Advanced Manufacturing, Université Laval, Québec, Canada

Ian P. Gent (9)
School of Computer Science, University of St Andrews, UK

Tias Guns (6)
Department of Computer Science, KU Leuven, Belgium

Djamal Habet (11)
Aix Marseille Univ, CNRS, LIS, Marseille, France

Daniel Harabor (33)
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia

Michael Hartisch (12)
University of Passau, Germany; FAU Erlangen-Nürnberg, Germany

Emmanuel Hebrard (3, 13, 27)
LAAS-CNRS, Toulouse, France

Clemens Hofstadler (14)
Johannes Kepler University, Linz, Austria

Alexey Ignatiev (42)
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia

Hannes Ihalainen (15)
University of Helsinki, Finland

Jef Jacobs (16)
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Mikoláš Janota (8)
CIIRC, Czech Technical University in Prague, Czech Republic

Wentao Jiang (34)
Ningbo University, China; National Center of Technology Innovation for EDA, Nanjing, China

Matti Järvisalo (15)
University of Helsinki, Finland

Mohimenul Kabir (17)
School of Computing, National University of Singapore, Singapore

Daniela Kaufmann (14)
TU Wien, Austria

Artem Kaznatcheev (18)
Department of Mathematics, and Department of Information and Computing Sciences, Utrecht University, The Netherlands

Asma Khoualdia (19)
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Zeynep Kiziltan (31)
Dept. of Computer Science and Engineering, University of Bologna, Italy

Frej Knutar Lewander (20)
Department of Information Technology, Uppsala University, Sweden

Henning Koehler (22)
Massey University, Palmerston North, New Zealand

Wietze Koops (21)
Lund University, Sweden; University of Copenhagen, Denmark

Buddhi W. Kothalawala (22)
The Australian National University, Canberra, Australia

Ryo Kuroiwa (5, 23)
National Institute of Informatics, Tokyo, Japan; The Graduate School of Advanced Studies, SOKENDAI, Tokyo, Japan

Nadine Laage (6)
Airbus Aerostructures GmbH, Hamburg, Germany

Alfons Laarman (38)
Leiden University, The Netherlands

Mikael Z. Lagerkvist (24)
SambaNova Systems, Stockholm, Sweden; Optischedule, Stockholm, Sweden

Jean-Marie Lagniez (38)
Artois University, Lens, France

Daniel Le Berre (21)
Univ. Artois, CNRS, UMR 8188 CRIL, Lens, France

Duc Anh Le (25)
DTIS, ONERA, Université de Toulouse, France

Christophe Lecoutre (25)
CRIL, Université d'Artois & CNRS, France

Jimmy H. M. Lee (5)
Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong

Yun Liang (34)
Peking University, Beijing, China

Peng Lin (26)
Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China; School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China

Tim Luchterhand (27)
LAAS – CNRS, Toulouse, France; Université de Toulouse, France

Jin Luo (34)
Peking University, Beijing, China; National Center of Technology Innovation for EDA, Nanjing, China

Ole Lübke (28)
Institute for Software Systems, Hamburg University of Technology, Germany

Marco Maratea (33)
Department of Mathematics and Informatics, University of Calabria, Rende, Italy

Imko Marijnissen (35)
Delft University of Technology, The Netherlands

Younes Mechqrane (29)
Ai movement - International Artificial Intelligence Center of Morocco, University Mohammed VI Polytechnic, Rabat, Morocco

Kuldeep S Meel (17)
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

Jingyi Mei (38)
Leiden University, The Netherlands

Ian Miguel (31)
School of Computer Science, University of St Andrews, Scotland

Magnus O. Myreen (21)
Chalmers University of Technology, Gothenburg, Sweden; University of Gothenburg, Sweden

Jason Nguyen (10)
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia; ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Jens Nicolay (16)
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Peter Nightingale (9, 43)
Department of Computer Science, University of York, UK

Jakob Nordström (21)
University of Copenhagen, Denmark; Lund University, Sweden

Andy Oertel (21)
Lund University, Sweden; University of Copenhagen, Denmark

Samuel Pastva (17)
Faculty of Informatics, Masaryk University, Brno, Czechia

Justin Pearson (20)
Department of Information Technology, Uppsala University, Sweden

Daniel Pekar (30)
Department of Mechanical and Industrial Engineering, University of Toronto, Canada

Alessio Pellegrino (31)
Department of Mathematics and Computer Science, University of Southern Denmark, Denmark

Xiao Peng (32)
LAAS-CNRS, Université de Toulouse, Toulouse, France

Vaidyanathan Peruvemba Ramaswamy (36)
Algorithms and Complexity Group, TU Wien, Austria

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

Guillaume Poveda (6)
Airbus SAS, Toulouse, France

Michael Prümm (43)
European Southern Observatory, Garching, Germany

Martin Pépin (40)
Université Caen Normandie, ENSICAEN, CNRS, Normandie Univ, GREYC UMR 6072, F-14000 Caen, France

Claude-Guy Quimper (7)
Lab-Usine – Joint Research Unit for Advanced Manufacturing, Université Laval, Québec, Canada

Magnus Rattfeldt (24)
Jeppesen, Gothenburg, Sweden;
Optischedule, Stockholm, Sweden

Léo Robert (19)
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Stéphanie Roussel (25)
DTIS, ONERA, Université de Toulouse, France

Pierre Schaus (41)
ICTEAM, UCLouvain, Belgium

Andreas Schutt (33)
Data61, CSIRO, Melbourne, Australia

Zhengyuan Shi (34)
The Chinese University of Hong Kong, Hong Kong;
National Center of Technology Innovation for EDA, Nanjing, China

Konstantin Sidorov (35)
Delft University of Technology, The Netherlands

Christine Solnon (2, 32)
Univ Lyon, INSA Lyon, Inria, CITI, EA3720, 69621 Villeurbanne, France

Peter J. Stuckey (5, 10, 42)
Department of Data Science and AI, Monash University, Clayton, Australia;
ARC Training Centre OPTIMA, Melbourne, Australia

Stefan Szeider (36, 39)
Algorithms and Complexity Group, TU Wien, Austria

Guido Tack (10)
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia;
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Yong Kiam Tan (21)
Institute for Infocomm Research (I²R), A*STAR, Singapore;
Nanyang Technological University, Singapore

Cyril Terrioux (11)
Aix Marseille Univ, CNRS, LIS, Marseille, France

Sylvie Thiébaux (27)
LAAS-CNRS, Toulouse, France;
Université de Toulouse, France;
Australian National University, Canberra, Australia

Van-Giang Trinh (17)
Inria Saclay, EP Lifeware, Palaiseau, France

Felix Ulrich-Oltean (9, 43)
Department of Computer Science, University of York, UK

Mauro Vallati (33)
Department of Computer Science, University of Huddersfield, UK

Sofia Vazquez Alferez (18)
Department of Mathematics, and Department of Information and Computing Sciences, Utrecht University, The Netherlands

Marc Vinyals (21)
University of Auckland, New Zealand

Florentina Voboril (36)
Algorithms and Complexity Group, TU Wien, Austria

Jack Waller (9)
Skyscanner, London, UK

Qing Wang (22)
The Australian National University, Canberra, Australia

Nils Weidmann (37)
Westdeutscher Tischtennis Verband e.V., Duisburg, Germany

Damien T. Wojtowicz (3)
LAAS – CNRS, Toulouse, France

Qiang Xu (34)
The Chinese University of Hong Kong, Hong Kong;
National Center of Technology Innovation for EDA, Nanjing, China

Dekel Zak (38)
Leiden University, The Netherlands

Samih Zein (41)
High Performance Composite team, Cenaero, Gosselies, Belgium

Tianwei Zhang (39)
Algorithms and Complexity Group, TU Wien, Austria

Xindi Zhang (34)
Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Science, Beijing, China

Allen Z. Zhong (5, 42)
Department of Data Science and AI, Monash University, Clayton, Australia;
ARC Training Centre OPTIMA, Melbourne, Australia

Ghiles Ziat (40)
EPITA Reasearch Lab (LRE),
Le Kremlin-Bicêtre, France

Mengchuan Zou (26)
Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Xavier Zwingmann (7)
Lab-Usine – Joint Research Unit for Advanced Manufacturing, Université Laval, Québec, Canada;
SOKÏO Industrie, Saint-Augustin-de-Desmaures, Canada

Senior Program Committee

- Özgür Akgün (University of St Andrews)
- J. Christopher Beck (University of Toronto)
- Sophie Demassey (Mines Paris)
- Ian P. Gent (University of St Andrews)
- Tias Guns (KU Leuven)
- Emmanuel Hebrard (LAAS)
- Zeynep Kiziltan (University of Bologna)
- Jimmy H. M. Lee (Chinese University of Hong Kong)
- Laurent Michel (University of Connecticut)
- Nina Narodytska (VMWare)
- Gilles Pesant (Polytechnic Montréal)
- Christine Solnon (INSA Lyon)
- Peter J. Stuckey (Monash University)
- Charlotte Truchet (Nantes Université)
- Standa Zivny (University of Oxford)

31st International Conference on Principles and Practice of Constraint Programming (CP 2025).
Editor: Maria Garcia de la Banda
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committee

- Carlos Ansótegui (Universitat de Lleida)
- John Aoga (UCLouvain)
- Andre Augusto Cire (University of Toronto)
- David Bergman (University of Connecticut)
- Kyle Booth (AWS)
- Kenneth N. Brown (University College Cork)
- Silvia Butti (University of Oxford)
- Shaowei Cai (Institute of Software, Chinese Academy of Sciences)
- Quentin Cappart (École Polytechnique de Montréal)
- Clément Carbonnel (LIRMM/CNRS)
- Mats Carlsson (Research Institutes of Sweden)
- Michael Codish (Ben-Gurion University of the Negev)
- Martin C. Cooper (University of Toulouse)
- Nguyen Dang (University of St Andrews)
- Jip J. Dekker (Monash University)
- Marianne Defresne (KU Leuven)
- Emir Demirović (Delft University of Technology)
- Pierre Flener (Uppsala University)
- María Andreína Francisco Rodríguez (Uppsala University)
- Simon de Givry (INRAE - MIAT)
- Stephan Gocht (AI Software Engineering)
- Ruth Hoffmann (University of St Andrews)
- Vinasetan Ratheil Houndji (Université d'Abomey-Calavi)
- (Eliza) Xin-Yi Hu (Chinese University of Hong Kong)
- Marie-José Huguet (LAAS-CNRS)
- Alexey Ignatiev (Monash University)
- Matti Järvisalo (University of Helsinki)
- Christopher Jefferson (University of St Andrews)
- George Katsirelos (MIA Paris INRAE AgroParisTech)
- Ryo Kuroiwa (National Institute of Informatics, Japan)
- Mikael Z. Lagerkvist (Sambanova Systems)
- Arnaud Lallouet (Huawei)
- Edward Lam (Monash University)
- Nadjib Lazaar (Université Paris-Saclay)
- Kevin Leo (Monash University)
- Andrea Lodi (Cornell Tech)
- Michele Lombardi (University of Bologna)
- Jayanta Mandi (KU Leuven)
- Vasco Manquinho (Universidade de Lisboa)
- Ian Miguel (University of St Andrews)
- Nysret Musliu (TU Wien)
- Peter Nightingale (University of York)
- Anastasia Paparrizou (CNRS-LIRMM, University of Montpellier)
- Xiao Peng (INSA Lyon)
- Thierry Petit (Emotia)

31st International Conference on Principles and Practice of Constraint Programming (CP 2025).
Editor: Maria Garcia de la Banda
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committee

- Charles Prud'Homme (IMT Atlantique, LS2N)
- Claude-Guy Quimper (Université Laval)
- Louis-Martin Rousseau (Polytech Montreal)
- Stéphanie Roussel (Onera)
- Pierre Schaus (UCLouvain)
- Andreas Schutt (CSIRO)
- Ilankaikone Senthooran (Monash University)
- Paul Shaw (IBM)
- Mohamed Siala (INSA Toulouse & LAAS-CNRS)
- Laurent Simon (Bordeaux University)
- Kostas Stergiou (University of Western Macedonia)
- Guido Tack (Monash University)
- Dimos Tsouros (KU Leuven)
- Willem-Jan Van Hoeve (Carnegie Mellon University)
- Hélène Verhaeghe (UCLouvain)
- Petr Vilím (Coenzyme Fr)
- Roland Yap (National University of Singapore)
- Allen Z. Zhong (Monash University)

Additional Reviewers

- Eric Bourreau (LIRMM, Université de Montpellier)
- Carleton Coffrin (Los Alamos National Laboratory)
- Frits de Nijs (Monash University)
- Welington de Oliveira (Mines PSL)
- Rohit Dureja (Advanced Micro Devices)
- Marijn Heule (Carnegie Mellon University)
- Kuldeep S. Meel (University of Toronto)
- Tomáš Peitl (Technische Universität Wien)
- Alexey Solovyev (University of Utah)
- Marcel Steinmetz (LAAS / CNRS)
- Adel N. Toosi (University of Melbourne)

31st International Conference on Principles and Practice of Constraint Programming (CP 2025).
Editor: Maria Garcia de la Banda

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

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.CP.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.CP.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.

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.

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.

Solving the Agile Earth Observation Satellite Scheduling Problem with CP and Local Search

Valentin Antuori
LAAS – CNRS, Toulouse, France

Damien T. Wojtowicz
LAAS – CNRS, Toulouse, France

Emmanuel Hebrard
LAAS – CNRS, Toulouse, France

Abstract

The increasing hunger for remote sensing data fuels a boom in satellite imagery, leading to larger agile Earth observation satellite (AEOS) constellations. Therefore, instances of the AEOS scheduling problem (AEOSSP) has become harder to solve. As most existing approaches to solve AEOSSP are designed for a single spacecraft or smaller constellations in mind, they are not tailored to the need of our industrial partner that is about to launch a constellation of 20 AEOSs. Hence, we designed a local search solver able to schedule observations and downloads at such a scale. It relies on solving a series of sub-problems as travelling salesman problem with time windows (TSPTW), first greedily, then using a CP-SAT exact solver in order to find a solution when the greedy insertion fails. Lastly, it schedules downloads and enforces memory constraints with greedy algorithms. Experiments were carried out on instances from the literature as well as generated instances from a simulator we designed. Our experiments show that using CP to solve the sub-problem significantly improve the solutions, and overall our method is slightly better than state-of-the-art approaches.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling; Applied computing → Operations research

Keywords and phrases Local Search, Greedy Algorithms, Aerospace Applications

Digital Object Identifier 10.4230/LIPIcs.CP.2025.3

Supplementary Material *Dataset*: `https://github.com/ssquilla/Earth_Observing_Satellites_benchmarks` [35]; archived at `swh:1:dir:c1c87cf9c3a3e7ffe616bdffdc9c52af9e30cb35`

Funding France Relance grant (project JAPETUS)

1 Introduction

Fueled by an increasingly easier access to space, technological advances, and an ever-growing hunger for up-to-date information, satellite imagery is booming. Since the introduction of the Pléiades project [13], satellite surveillance has typically been carried out using agile Earth observation satellites (AEOS), because of their ability to observe an area within a large horizon thanks to their yawing, pitching and rolling capabilities (see Fig. 1). Those satellites are typically deployed within constellations of a few spacecrafts. Current existing projects, such as JAPETUS, lead by our industrial partner Prométhée[1], aim at launching constellations of several dozen satellites.

As a consequence of the growing size of instances and constellations, the already NP-hard AEOS Scheduling Problem (AEOSSP) becomes even harder to solve in practice. This is a major problem for AEOS constellation operators, as low-quality observation plans would

[1] See `https://promethee.earth/`.

© Valentin Antuori, Damien T. Wojtowicz, and Emmanuel Hebrard;
licensed under Creative Commons License CC-BY 4.0

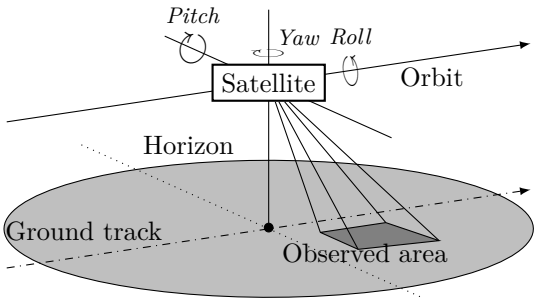


Figure 1 Schematic view of an agile satellite's observation capabilities.

compromise their profitability. Moreover, their business model is increasingly based on reactivity, allowing their customers to request images and have them delivered within a few hours. AEOSSP must then be solved as fast as possible, since it is part of broader processes that are all subject to tightening deadlines. As a side effect, image download planning has to be efficient in order to minimize the overall processing time of a request. In the past two decades, extensive work has been carried out on solving several versions of the AEOSSP [34]. Many methods have been proposed so far, including exact methods, heuristics and metaheuristics. However, most of these proposals are designed with a single spacecraft or smaller constellations in mind, and without considering download planning or a reactive business model.

In this paper, we introduce the industrial application of an acquisition planning method using several techniques from CP, that can deal with large scale instances including complex requests while performing download planning. Our method can quickly find good solutions for a constellation of twenty AEOSs on instances reaching up to a thousand complex requests over a planning horizon of six hours. Our approach is similar to the algorithm designed by Squillaci, Pralet and Roussel [25] in that it relies on a local search approach hybridized with an external solver. However, our method not only uses a SAT-based CP solver to solve sub-problems rather than solely relying on greedy algorithms, but also considers satellite memory constraints and takes download decisions instead of pre-assigning time intervals dedicated to downloads.

It solves the AEOSSP by decomposing it into an acquisition planning problem and a download planning problem. The first one is solved via local search whose moves aims at building a sequence of acquisitions for a single satellite over a given time period consisting of several overlapping acquisition opportunities, which often correspond to a daylight portion of the satellite's orbit over land. Those moves are performed using calls to a CP solver and greedy algorithms. The download planning problem takes an acquisition plan as input and is solved using a greedy algorithm.

Experiments were carried out over a set of instances from the literature as well as instances generated using a simulator we designed. Results shows that our solver is comparatively faster and finds better solutions than both the state-of-the-art aforementioned LNS method tailored for AEOSSP and a baseline model implemented with IBM's CP Optimizer.

This paper is organized as follows. AEOSSP is defined in Sect. 2. Then, a brief overview of related work is provided in Sect. 3. Our approach is explained in Sect. 4, then evaluated and compared with the literature in Sect. 5. Finally, this paper is concluded in Sect. 6.

2 Problem Statement

The AEOSSP problem aims at scheduling observations and data transfers in order to maximize the profit of the constellation's operator. Together with our industrial partner, we adopted an integrated planning definition, i.e. including constraints on AEOSs' memory and computing not only an observation planning but also a download planning. In this respect, it is similar to that adopted in some of the latest research [11, 10]. Since our partner emphasizes reactivity, AEOSSP is periodically solved over a few-hour rolling planning horizon. Hereafter, we describe the elements of the problem before formalizing it.

2.1 Input Data

Let $\mathcal{R}$ be a set of customer *requests*. A request r involves observing an area split into a set $\mathcal{M}_r$ of *meshes*. Requests can be repeated, hence we define the set $\mathcal{P}_r$ of those *repetitions*. For instance, there can be a request to observe the area around Glasgow, which is decomposed into a grid of meshes covering the city, with a two-hour period during the CP 2025 conference.

The constellation we consider has a set $\mathcal{S}$ of *satellites*, that can be used to observe a mesh for a given period. Such an action is called an *acquisition opportunity*, although we shall use *acquisition* for short. Let $\mathcal{A}$ be the set of acquisitions. We define the subset $\mathcal{A}_{r,t,m}$ as the set of possible acquisitions of the request r for the period t and the mesh m, and $\mathcal{A}^s$ as the subset of acquisitions which can be made by satellite s. Each acquisition a has a duration p_a to make that observation, which must occur during a time window $[r_a, d_a]$. It is associated to a profit v_a, and it consumes a quantity of memory γ_a. Satellites may have some acquisitions in memory at the beginning of the planning period. For each satellite s, $\mathcal{F}^s$ is the set of such acquisitions, and we note $\mathcal{F} = \cup_{s\in\mathcal{S}}\mathcal{F}^s$. There can be many alternative acquisitions for the same observation *task* (i.e. a mesh for a given period), since the same image can be shot by different satellites or the same satellite on different orbits. Therefore, the goal is to select acquisitions among the many alternatives, so that as many requests as possible are fulfilled.

Then, the acquisitions must be scheduled. As satellites need to maneuver in order to aim at their targets (i.e. to observe a mesh), there are *transition times* between successive acquisitions performed by the same satellite. In agreement with our industrial partner and in line with other works in the literature [3], we assume that transition times are time-independent. Thus, $T(a_1, a_2)$ is the transition time between two acquisitions $\{a_1, a_2\} \subseteq \mathcal{A}^s$ on a satellite s.

Finally, a network of *ground stations* (GS) are used to download the observations. For each acquisition a, there is a set $\mathcal{W}_a$ of possible *download windows*. A download time window w is a time window $[r_w, d_w]$ during which acquisitions can be downloaded. It has a total capacity γ_w which is the product of its bandwidth by its duration. Each satellite s has a memory capacity C_s.

Memory management is the second part of the problem. It consists of assigning each selected acquisition in the plan to a download time windows, while ensuring that onboard memory and download capacity constraints are respected.

2.2 Decision Variables and Constraints

The following model formally describes the problem. For each acquisition $i \in \mathcal{A}$, we have a boolean selection variable $x_a \in \{0, 1\}$ and a start time variable $s_a \in \{r_a, d_a - p_a\}$. Moreover, for each acquisition $a \in \mathcal{A} \cup \mathcal{F}$ and for each of its download options $w \in \mathcal{W}_a$, there is a boolean selection variable $y_{a,w} \in \{0, 1\}$.

The problem is to compute a mission, i.e. select and schedule the acquisitions and downloads for each satellite in order to maximize the profit, satisfying the following constraints:

Acquisitions time windows: Selected acquisitions must occur wholly within their time windows.

$$x_a \implies r_a \leq s_a \leq d_a - p_a \quad \forall a \in \mathcal{A} \tag{1}$$

Acquisitions no-overlap: On a satellite $s \in \mathcal{S}$, the selected acquisitions must not overlap in time.

$$x_a \wedge x_b \implies s_a + p_a + T(a,b) \leq s_b \vee s_b + p_b + T(b,a) \leq s_a \quad \forall \{a,b\} \subseteq \mathcal{A}^s \tag{2}$$

Acquisitions alternatives: No more than one acquisition among alternatives for each observation target shall be done.

$$\sum_{a \in \mathcal{A}_{r,t,m}} x_a \leq 1 \quad \forall r \in \mathcal{R}, \forall t \in \mathcal{P}_r, \forall m \in \mathcal{M}_r \tag{3}$$

Acquisitions download: Selected acquisitions cannot be downloaded more than once, they must be finished before being downloaded[2].

$$\sum_{w \in \mathcal{W}_a} y_{a,w} \leq x_a \qquad \forall a \in \mathcal{A} \tag{4}$$

$$y_{a,w} \implies s_a + p_a \leq r_w \quad \forall a \in \mathcal{A}, \forall w \in \mathcal{W}_a \tag{5}$$

Satellite memory: At any given time t, the quantity of data stored on a satellite $s \in \mathcal{S}$ (equal to the total acquired before t minus the total download before t) must not exceed its capacity[3].

$$\sum_{a \in \mathcal{A}^s | x_a \wedge s_a < t} \gamma_a - \sum_{a \in \mathcal{A}^s, w \in \mathcal{W}_a | y_{a,w} \wedge d_w < t} \gamma_a \leq C_s \quad \forall t \tag{6}$$

Downloads capacity: The total quantity of data downloaded during a time window must not exceed its capacity.

$$\sum_{a \in \mathcal{A}} y_{a,w} \gamma_a \leq \gamma_w \quad \forall w \in \mathcal{W} \tag{7}$$

The objective is then simply to maximize the total profit:

$$\sum_{a \in \mathcal{A}} x_a v_a \tag{8}$$

This objective function might seem overly simple. However, from the point of view of the operators, this is relatively easy way to steer the plan toward their true objectives. In particular, some efforts have been put into complex requests with non-additive rewards [25]. For instance, imagine that you want a complete map of Scotland. It requires to perform a large set of acquisitions, hence having only a subset of the observations may be worthless, or at least the profit should not be linear with the ratio of the observations that were made. However, such complex requests are likely to span over several planning horizons. Therefore, modeling a complex cost system might be detrimental if the whole request can never fit within one planning horizon. On the other hand, with a simple additive objective, the operators may for instance start with attaching low profits to each observation of a complex request, and then increase the weight associated to missing observations on subsequent planning sequences, depending on how close to completion they are.

3 Related Work

Initial work on AEOSSP was carried out as part of the development of the Pléiades constellation [13]. Since then, there have been numerous articles on this topic [34, 24], considering several variations of the problem, and exploring a wide array of methods that are generally in line with related aerospace problems [36]. In this section, we provide a brief overview of the state of the art on AEOSSP.

[2] We make the conservative assumption that an acquisition can be downloaded only if the acquisition is finished at the start of the download time window.

[3] We make the conservative assumption that memory is released at the end of the download time windows.

3.1 AEOSSP variations

Substantial differences exist in the scope of the AEOSSP used throughout the literature, mainly on extensions to the core observation scheduling problem [13]. This core is NP-hard [13]. Moreover, exhaustively taking into account all possible extensions of the AEOSSP can make it too complex, therefore intractable for existing methods. As a consequence, all the publications that we are aware of about AEOSSP consider only a subset of the constraints with more or less radical simplifications. We also defined with our industrial partner a somewhat simplified version of AEOSSP in this paper, as previously discussed in Sect. 2.

Some models [28, 6, 14] consider the breakdown into meshes and strips (i.e. long polygons imaged while the satellite is moving) of the requests' areas of interest as decision. As in most of the literature, we assume that this step is part of some preprocessing and therefore outside of the scope of our work.

Depending on the AEOSs' capabilities, the nature of the requests themselves can vary. Indeed, they can be simple, one-shot images of a given area, strips covering long areas [1], repetitive, stereoscopic [20], video requests [5], or even be instrument-specific when several sensors are on the spacecrafts [18, 17, 37]. Our solver is agnostic to the nature of the acquisitions; expressing punctual and stereoscopic requests can be done at preprocessing time using repetitions.

The *quality* of the images is time-dependent [21, 16, 29, 33, 22] and multifaceted (shooting angle, luminosity, cloud cover, customer preferences, etc.), leading some authors to consider AEOSSP as a multi-objective problem [16, 15]. Hence, in addition to profit, those works seek to maximize image quality or use of the satellites. *Profit* itself can be time-dependent [28] since it is quality-dependent. It can also be affected by urgency and priority policies among requests [29, 27], and even be sequence-dependent in the case of repetitive or stereoscopic requests [4, 12, 14, 20]. In our work, we have a black-box approach to profit as it is determined for each acquisition by our partner: quality and urgency are therefore sidelined as they are embedded within the profit, hence removing its time-dependence.

There are also operational constraints. Firstly, *transition times*, which is the time the satellite needs to maneuver towards its target. They are by nature dependent on the time and sequence of acquisitions made by AEOS [3, 22, 2]. They are either modeled as such, or approximated by piecewise linear functions or constants, or simply ignored. Our approach also approximates transition time using a transition time matrix between categories of attitude adopted by the AEOSs when performing the acquisitions. Hence, it allows to take into account time and sequence dependencies at a granularity defined at preprocessing time while formally removing time-dependence from our model.

Then there can be constraints about onboard *energy* management. The energy comes from solar panels, and the rate at which batteries are refilled therefore depends on both the time (the exposition to the sun depends on the orbit of the satellites) and the tasks performed by the satellite since it constrains the orientation of the solar panels [32, 33]. However, on-board energy management is rarely a bottleneck, and hence it is usually not taken into account when planning the observations. In accordance with the needs expressed by our industrial partner, this aspect is not covered in our work.

Finally, there is the management of onboard *memory* and its corollary: the management of *downloads* [21, 10, 11, 19]. Scheduling both acquisitions and downloads is often referred to *integrated scheduling* in the literature [34]. Memory can be looked at in two ways. By ignoring the files, it becomes a reservoir that is filled and emptied according to linear functions. If the files are taken into account, then it is a set of blocks that must be distributed between the download windows according to their capacity. We take the second approach, as it best fits our partner's processes.

3.2 Resolution methods

The wide range of variations in AEOSSP has resulted in the study of numerous resolution methods.

Exact methods have been extensively used to solve the AEOSSP. For instance, a branch-and-bound algorithm taking into account time-dependent constraints was proposed and shown to be effective on small instances [3]. Several MILP formulations have also been proposed [29, 12], including in the context of integrated scheduling [10, 11]. Some work only focus on data download [19] and are more in line with broader work on the Antenna-Satellite scheduling problem. Although MILP was successfully used to perform integrated scheduling with a commercial solver for Planet Lab's constellation of over a hundred spacecrafts [23], they are not *agile* satellites and hence the scheduling problem is a lot easier than the AEOSSP. Moreover, while projects with a problem statement roughly similar to ours [11] does use MILP, their instances are way smaller (4 satellites, up to 3 ground stations and 200 requests), do not take into account transition times, and have a time limit orders of magnitude too high (approx. 3 h) to be used in our industrial partner's processes. For all these reasons, we have not retained the possibility of using CP to solve the problem as a whole, but rather to solve sub-problems as a local search move.

Much of the work on AEOSSP has chosen to abandon optimality guarantees and propose methods based on heuristics and metaheuristics. Several methods solely based on heuristics have been proposed to solve AEOSSP. The scheduler of the COSMO-SkyMed constellation, comprising four spacecrafts, simply selects as much acquisitions as it can by descending order of priority [1]. Heuristics can also be used to compute the acquisitions given areas of interest in the context of an online scheduler [14]. A method using deep reinforcement learning and heuristic algorithm has also recently been proposed to solve AEOSSP for a single satellite [2]. Although those heuristics-based methods are easy to implement and yield good results for a single satellite or smaller constellation, they are not expected to be as good when applied to a whole constellation because of the larger number of alternative acquisitions. Methods using machine learning are also of great interest to solve AEOSSP, as satellite operators accumulate many instances over time, but the size of the instances and the lack of real-world data from our industrial partner led us away from learned heuristics.

Many metaheuristics-based methods has also been proposed since the first studies on AEOSSP, such as taboo search [4]. Genetic algorithms have been widely used to tackle multi-objective versions of the AEOSSP, for instance when considering operator's preferences [16], load balancing [7], or solution stability in an online setting [15] in addition to profit. They were also applied to decompose sub-problems. In a first example, it is used for single-satellite scheduling within a broader game theory framework where satellites are players [20]. In a second example, it is used to choose between overlapping acquisitions within multi-level clusters computing using priority metrics [8]. Agent-based methods are also interesting to solve small-scale instances of AEOSSP [5].

Finally, local search and large neighborhood search (LNS) methods have also been extensively studied, including in the context of multi-objective optimization [28, 37] and online solvers [27]. Moves proposed in the literature rely on a wide array of algorithms. On the more complex side, there is a move based on computing maximum independent sets within acquisitions interval graphs using an evolutionary algorithm [6]. Unsupervised learning has also been used as a preprocessing tool for destructive LNS moves [21]. However, most of the constructive LNS moves are simpler, as they widely rely on random choices [9] and greedy algorithms [9, 22, 25]. It should be noted that these local search and LNS methods are similar to what can be found in the team orienteering problem [30], since it shares many characteristics of the acquisition selection sub-problem of AEOSSP.

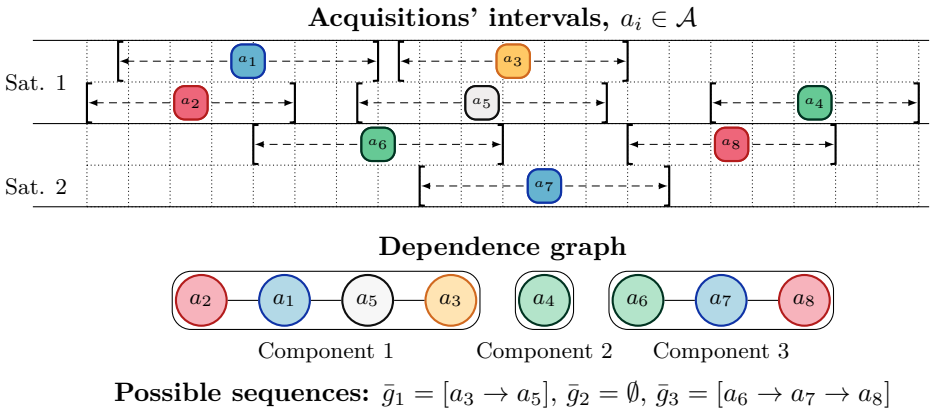


Figure 2 Example of sequences defined from the dependence graph built from a set of acquisistions. Acquisitions with the same colour are alternatives to cover the same mesh for the same task of the same request. The interval graph contains three connected components.

The solver we present in this article is also in this line of work, as local search methods are particularly effective for solving complex instances in satellite constellations, while making it easier to break down the problem into acquisition planning and download management.

4 A Local Search Approach

In order to solve our version of AEOSSP, we leverage the structure of the problem to decompose it into two parts. Firstly, the selection and allocation of the acquisitions – the master problem – is solved using a local search method that computes sequences of acquisitions for each satellite. Secondly, one planning of file downloads – the sub-problem – is solved using a greedy algorithm. Those two parts are repeated until a time limit.

4.1 Acquisition Planning

The acquisition planning phase aims at building a sequence of acquisitions for each satellite maximizing the profit under the aforementioned constraints 1, 2 and 3.

Our method uses the fact that there exists subsets of acquisitions of a given satellite that are temporally independent of each other [6, 26]. This can happen when the satellite flies over a large area with little interest (e.g. an ocean, cloudy areas, etc.), or simply because of the satellite's orbital nighttime. Formally, for each satellite $s \in \mathcal{S}$ we can define a dependence graph whose nodes are the acquisitions $a \in \mathcal{A}^s$. There is an edge between two acquisitions $\{a, b\} \subseteq \mathcal{A}^s$ if $d_a + T(a, b) > r_b \vee d_b + T(b, a) > r_a$, i.e., when their time windows overlap while taking into account transition time. As a consequence, each connected component of these graphs are temporally independent of each other (see example in Fig. 2). Hence, local optimization may be performed over each connected component in order to build the plan in the form of sequences of acquisitions.

Our algorithm for acquisition planning has two phases. Firstly, it builds a solution from scratch with a greedy algorithm. Secondly, a local search phase aims at improving the solution until a local minimum is reached.

4.1.1 Greedy Initial Solution Construction

The greedy algorithm iteratively tries to insert an acquisition. It uses a utility criterion $U(a) = v_a/p_a$ defined as the ratio of an acquisition a's profit over its duration to select the next acquisition to insert. The algorithm iterates over the acquisitions whose associated mesh and task are not currently covered by descending order of their utility, by trying to insert them in their satellite's plan to minimize the increment of transition times.

The time complexity to find the best valid position is linear in the length of the current sequence and to insert the acquisition is linear in the size of the sequence under construction.

Therefore, the algorithm runs in $O(|\mathcal{A}|^2)$ time overall for a sequence of acquisition $\mathcal{A}$.

Let $\mathcal{A} = a_0, a_1, \ldots, a_k, a_{k+1}$ be a feasible sequence of acquisitions with a_0 and a_{k+1} dummy acquisitions of null duration, null transition time with all other acquisitions and such that $r_{a_0} = 0$ and $d_{a_{k+1}} = \infty$. The greedy insertion algorithm returns the position $i \in [0, k]$ such that inserting a between acquisitions a_i and a_{i+1} yields a feasible sequence and $T(a_i, a) + T(a, a_{i+1})$ is minimum.

The following quantities can be accessed in constant time:

$$s_{a_i} = \max(r_{a_i}, s_{a_{i-1}} + p_{a_{i-1}} + T(a_{i-1}, a_i)) \qquad \forall 1 \leq i \leq k \tag{9}$$

$$slack(a_i) = d_{a_i} - s_{a_i} - p_{a_i} \qquad \forall 1 \leq i \leq k \tag{10}$$

$$gap(a_i) = s_{a_i} - s_{a_{i-1}} + p_{a_{i-1}} + T(a_{i-1}, a_i) \qquad \forall 1 \leq i \leq k \tag{11}$$

Then, we store the values of the cumulative gap up until acquisition a_i in linear space:

$$cgap[a_i] = \sum_{j=0}^{i-1} gap(a_j) \qquad \forall 1 \leq i \leq k \tag{12}$$

Therefore, we can access in constant time $F(a_i, a_j)$ the time length available between tasks a_i and a_j (with $j > i$) in the sequence:

$$F(a_i, a_j) = cgap[a_j] - cgap[a_i] + slack(a_j) \qquad \forall 1 \leq i \leq k \tag{13}$$

Finally, we store the time length available for a task insertion immediately after task a_i using linear space:

$$I[a_i] = min_{i<j\leq k} F(a_i, a_j) \qquad \forall 1 \leq i \leq k \tag{14}$$

When trying to insert acquisition a, the algorithm makes a linear search over the sequence $a_0, a_1, \ldots, a_k$ to find the position $i \in [0, k]$ that minimizes $T(a_i, a) + T(a, a_{i+1})$, whilst ensuring that the insertion is feasible using the value of $I[a_i]$. Then, after the actual insertion, the values of s, $cgap$ and I are updated in linear time by once again traversing the acquisition in the sequence. An insertion therefore requires two linear-time traversals of the sequence.

Finally, we introduce randomization. Indeed, instead of taking the next acquisition in utility order, there is a small probability to take a random one instead. Once this algorithm has produced an initial solution, the local search starts.

4.1.2 Local Search

Our method locally optimizes the plan by iteratively recomputing sub-sequences. A move for our local search consists in solving a kind of *orienteering problem* over the complete graph formed by the acquisitions of a given connected component. In this graph, each node is an acquisition with the same time window, and is associated to a profit, which we will describe

below. The transition time between two acquisitions is the length of the path between the two associated nodes. The goal is to find a path in this graph that maximizes the sum of the profit of the visited nodes, while respecting time windows (i.e. finding a maximum-profit sequence of acquisitions).

The profit of selecting an acquisition a is defined as follows. It uses the marginal profit $\mathcal{P}$ that is the profit increment that would occur if an acquisition was inserted in the plan instead of another alternative acquisition (i.e., the cost of swapping acquisition a for an alternative acquisition b if it was in the current plan, that is, if $x_b = 1$).

$$\mathcal{P}(a) = v_a - \sum_{b \in \mathcal{A}_{r,t,m} \setminus a} x_b v_b \quad r \in \mathcal{R}, t \in \mathcal{P}_r, m \in \mathcal{M}_r \mid a \in \mathcal{A}_{r,t,m} \tag{15}$$

Should the move find a solution with a higher overall profit than the current plan, the new sequence that was built for the considered connected component substitutes the one in the plan. The latter is updated as a whole, by deselecting already selected alternative acquisitions from the new sequence.

To solve this sub-problem, we use the same procedure as for the initialization: for each acquisition in decreasing order of marginal profit, it tries to insert acquisitions within the sequence. This procedure stops when the first acquisition with a non positive marginal profit is reached. Two insertion techniques are used. The first tries to insert the acquisition in the sequence so that transition times are minimized, just as in the initial solution construction. The second technique is used when the first fails: an external solver is called to perform the insertion of the new acquisition by rescheduling the whole connected component.

We use *Tempo*[4], a CP-SAT solver that we are developing as part of a distinct project. It is asked to solve a traveling salesman problem with time windows (TSPTW), where the goal is to find a valid tour in the graph formed by the acquisitions already in the sequence extended with the acquisition we are trying to insert. The time window of a node is the time window of the acquisition, and the transition time between two nodes is the transition time between the two associated acquisitions. We model this problem with one interval variable for each acquisition i. The release and due dates of the interval variable of an acquisition i are r_a and d_a respectively, and its duration is p_a. Then the only constraint of the model is a NoOverlap on all the acquisitions, taking into account the transition times. There is no objective function as we just seek a valid sequence of acquisitions. Tempo's search is limited by a fail limit. The main benefit of this approach is that it allows to reorder the sequence when a solution is found, possibly leaving space for further acquisition insertions.

Finally, the neighborhood of a given plan is defined by the number of connected components. Indeed, we have one possible move per connected component, that eventually leads to an improving neighbor. Therefore we choose to perform a simple descent, choosing at each iteration the first improving neighbor among all randomly ordered neighbors, until a local minimum is reached, i.e. none of move can improve the current plan.

4.2 Download Planning

The download planning phase takes place when a local minimum is found. Its aim is twofold. It first computes a download plan by affecting download windows to the planned acquisitions and the initial memory elements aboard the satellite. Then, it fixes potential remaining memory overloads on the satellites by progressively deselecting acquisitions.

[4] See `https://gitlab.laas.fr/roc/emmanuel-hebrard/tempo`.

For each satellite, the algorithm iterates over the download windows by increasing order of their starting time, in order to greedily affect acquisitions until the download window's capacity is reached or there is no remaining acquisition to download. This greedy algorithm starts by collecting the acquisitions that are still on board at the starting time of the download time window, and sort them by decreasing order of their memory consumption. Then acquisitions are assigned in this order to the window if they fit.

Despite download decisions being taken, memory constraints may still be violated as the algorithm is blind to memory overloads. Hence, a memory overload repairing procedure is applied to satellites with overloads. Since we assume that the memory is freed at the end of the download window, overloads are checked at these time points. When one is found, a knapsack problem is solved over the onboard acquisitions. The knapsack capacity is the memory capacity of the satellite, the weights are the memory consumption of the acquisitions, and the values are the profits of the acquisitions. We use a simple greedy algorithm with a profit over size ratio utility to solve the knapsack problem.

It should be noted that while both procedures are using greedy algorithms, it is not an issue as download planning is often an easy task and memory is rarely a bottleneck. Finally, it should be noted that since our overall method is solved on a rolling horizon, some images may not be downloaded during the current planning horizon. They will be the content of the set $\mathcal{F}$ of the next planning horizon.

5 Experiments and Results

We evaluated our approach against a state-of-the-art commercial solver and recent proposals from the literature, over instances from the literature as well as generated by a simulator of our own design.

5.1 Experimental setting

Experiments were run on a cluster of computers with 2,1 GHz Intel CPUs and 256 GB RAM. Results are the aggregation of 5 runs with different seeds and a 10 min timeout. We implemented our solver with C++ 20.

5.1.1 Instances

We used two instance sets to evaluate our work.

The first instance set, hereafter referred to as "Squillaci et al.", comes from recent work[5] [25]. It contains 32 instances containing up to 1140 requests. Details of the instances are provided in Table 7 (Appendix. A). They are regional instances focusing on Europe, half of them are generated using a uniform target distribution and the other half focus on fifty large European airports, containing one-shot, video, stereoscopic and periodic requests. It should be noted that video requests are like one-shot requests but longer and more profitable. Moreover, a stereoscopic request implies observing the same area twice with the same satellite with two different angles. Therefore, we slightly adapted our method in order to support stereoscopic requests, by trying to insert the second acquisition right after the first one has been inserted into its satellite's sequence. If the second insertion fails, then the first acquisition's selection is canceled. The marginal profit of an acquisition from a stereoscopic

[5] Instances retrieved from `https://github.com/ssquilla/Earth_Observing_Satellites_benchmarks`.

Table 1 Satellites orbital parameters for the simulation, including plane inclination i (deg), altitude a (km), longitude of ascending node Ω (deg), mean anomaly M and period P (min).

(a) Inclined plane orbits.

i	a	Ω	M	P
90	500	-180	$\{-180, -90, 0, 90\}$	90
90	500	-90	$\{-180, -90, 0, 90\}$	90
90	500	0	$\{-180, -90, 0, 90\}$	90
90	500	90	$\{-180, -90, 0, 90\}$	90

(b) Heliosynchronous orbits.

i	a	Ω	M	P
98	550	-144	-180	90
98	550	-60	-90	90
98	550	60	0	90
98	550	144	90	90

request is the sum of both's marginal profits. We did not use the profit function proposed in [25] for periodic requests since we have an all-or-nothing approach for the requests. Finally, as the instance files did not provide transition times, we followed the method of Squillaci et al. to compute it as a function of the euclidean distance between the targets.

The second instance set we used was produced by an instance generator we designed[6]. The latter's input is a set of coordinates corresponding to points of interest (POIs) distributed over the Earth, then calculates meshes covering them using a greedy algorithm. It then simulates satellite orbits to determine the visibility time windows for each mesh and ground station over a given planning horizon. Requests are then generated by randomly selecting a subset of the meshes, with a probability of also selecting their neighbors in order to simulate requests over large geographical areas. The number of periods depends on the view accesses associated with the selected meshes, and is also randomly selected.

We produced a set of 20 instances[7], containing up to 500 requests over a planning horizon of 6 h, assuming a constellation of twenty satellites evenly spaced on four parallel inclined orbits and an heliosynchronous orbit (see orbital parameters in Table 1). We assumed a network of 6 ground stations with characteristics similar to KSATlite[8]. As in many other work, we used non-uniform POI distribution at regional scale [1, 9, 25] – hereafter referred to as "Antilles", "Sahara" and "Fountains" – and worldwide scale – "Military" and "World Heritage" – that we extracted from OpenStreetMap. For each POI set, we generated 4 instances from 50 to 500 requests. Further details about the instances are in Table 6 (Appendix. A).

In order to make a comparison, we use the same hypothesis to compute transition time as in Squillaci et al. instances. For a given target, we choose a base profit and a decreasing slope randomly, each acquisition has then a profit that depends on its start date, the later it is, the lower will be the profit. In addition to that, each acquisition has a chance to have a large penalty in order to simulate bad weather.

5.2 Comparison points

We compared our algorithm with the results of the aforementioned LNS method from the literature, as well as a baseline model with IBM's CP Optimizer (CPO).

[6] See source code at `https://gitlab.laas.fr/roc/damien-wojtowicz/aeossp_instance_generator`.
[7] Instances available at `https://gitlab.laas.fr/roc/damien-wojtowicz/aeossp_instances`.
[8] See `https://www.ksat.no/ground-network-services/ksatlite/`.

5.2.1 LNS Method

We choose Squillaci, Roussel and Pralet's [25] LNS method because, to our knowledge, it is the most similar to ours in the literature. It uses dedicated moves to relax sets of requests before reinserting the requests in the plan.

Its destructive phases use the same concept of connected components within an acquisitions' time windows graph as our method, as well as other work in the literature [6]. Those phases randomly select a subset of connected components, then relax a subset of acquisitions within, so as to have more choice to reinsert a request after relaxation.

Repair phases use CPO to solve an orienteering problem over the subset of the selected connected components. Although this algorithm can run in a multi-core parallel mode, we run the experiments with a single core in order to do a fair comparison with our algorithm.

5.2.2 CP Model with CPO

Finally, we implemented a baseline CP model with CPO version 20.1.0 using its C++ API, solved with a single thread and running on the same machines as our solver. Under the default parameter setting, CPO runs LNS until a local optimum is detected and then switches to complete search, namely, Failure Directed Search [31]. We defined three sets of interval variables to model this problem. First, there is an optional interval $X_a := \langle s_a, e_a \rangle$ for each acquisition $a \in \mathcal{A}$, with $s_a \geq r_a$, $e_a \leq d_a$ and $e_a - s_a = p_a$. Then, there is an interval $Y_w := \langle s_w, e_w \rangle$ for each download window $w \in \mathcal{W}$ with $s_w \geq r_w$ and $e_w \leq d_w$. Lastly, there is an optional interval $Z_{a,w} := \langle s_{a,w}, e_{a,w} \rangle$ for each download window $w \in \mathcal{W}_a$ for each acquisition $a \in \mathcal{A}$ with $s_{a,w} \geq s_w \wedge s_{a,w} \geq e_a$, $e_{a,w} \leq e_w$ and $e_{a,w} - s_{a,w} = m_a / b_w$.

The objective function is stated as follow:

$$\max \sum_{a \in \mathcal{A}} \textsc{PresenceOf}(X_a) \times v_a \tag{16}$$

We defined the following constraints. Constraint 17 ensures that acquisitions do not overlap in the plan for each satellite, taking into account a transition matrix T. Constraint 18 ensures uniqueness of observations. Constraint 19 defines a profile of memory usage, and Constraint 20 sets a limit on this profile. It should be noted that this is the typical way to model a reservoir using CPO, which we chose over a cumulative global constraint in order to avoid adding a task that starts with the acquisitions and ends at the download. Constraints 21 and 22 enforce consistency between acquisitions and downloads: only acquisitions in the plan can be downloaded and only one download can be done for each acquisition. Finally, Constraint 23 ensures that the capacity of each download window is not exceeded.

$$\textsc{NoOverlap}(\{X_a \mid S_a = s\}, T) \qquad \forall s \in \mathcal{S} \tag{17}$$

$$\textsc{Alternative}(\{X_a \mid R_a = r \wedge T_a = t \wedge M_a = m\}) \qquad \forall r \in \mathcal{R}, \forall t \in \mathcal{P}_r, \forall m \in \mathcal{M}_r \tag{18}$$

$$p_s := \sum_{a \in \mathcal{A}_s} step(X_a, m_a) - \sum_{w \in \mathcal{W}_s} step(Z_{a,w}, m_a) \qquad \forall s \in \mathcal{S} \tag{19}$$

$$\textsc{AlwaysIn}(p_s, 0, C_s) \qquad \forall s \in \mathcal{S} \tag{20}$$

$$\textsc{PresenceOf}(Z_{a,w}) \implies \textsc{PresenceOf}(X_a) \qquad \forall a \in \mathcal{A}, \forall w \in \mathcal{W}_a \tag{21}$$

$$\textsc{Alternative}(\{Z_{a,w} \mid w \in \mathcal{W}_a\}) \qquad \forall a \in \mathcal{A} \tag{22}$$

$$\textsc{NoOverlap}(\{Z_{a,w} \mid \forall a \in \mathcal{A}_w\}) \qquad \forall w \in \mathcal{W} \tag{23}$$

It should be noted that, for Squillaci et al. instances, this model has to be adapted by removing all constraints and variables dealing with the memory and the downloads, treating all the downloads as acquisitions except that their intervals are not optional, and by adding an equality constraint between the presence variable of stereoscopic pairs of acquisitions.

5.3 Results and Discussion

We now present and discuss the experimental results, by first comparing our method with the aforementioned comparison approaches. Then, we assess the impact of our use of the CP-SAT solver Tempo within the local search move, as well as the consequences of how our method manages satellite's memory.

5.3.1 Comparison with LNS and CPO

We compared our method (noted as `LS-tempo`, with a fail limit of 50 when using the solver) against the LNS from the literature (`LNS`) and the CPO model (`CPO`) over Squillaci et al.'s instances (see Table 2) and over our set of instances in (see Table 3).

The baseline `CPO` model is the most competitive for the smaller instances in every instance set. However, it does not scale up to large instances as well as the two other methods. On Squillaci et al.'s instances, it has good results up to 500 requests, is the best on 4 instances and can also prove optimality on 4 instances. On our instances, it is the best up to 100 requests and proves optimality on 3 instances. Auxiliary experiments on Squillaci's instances showed that CPO performs better with a 1 hour time budget (up to 27 % increase in profit), but still stays behind other methods (with 10 minutes budgets), especially on large instances (for example the 0-1000-0-0 instances: 22 % and 17 % behind our method). Nevertheless, our industrial context does not allow such a time budget as the resolution is part of a larger process.

Secondly, `LS-tempo` shows good performances and has often the best solutions on both instance sets (21 times the best solution, compared to 16 for `CPO` and 7 for `LNS`). On Squillaci et al. instances, while our solver is the worst over instances with periodic requests only, it is especially good for the 1000 video requests instance of concentrated target (16 % gap versus 21 % for `LNS`). `LNS` has the most balanced performances, is particularly good for the periodic-only instances, and has the best gap in average. Over our instances, `LS-tempo` is the best solver in average (0.5 % gap better). Therefore, the overall performance difference between `LNS` and `LS-tempo` is not clear since `LNS` seems to be good in average, but `LS-tempo` finds the best solution more often.

It should be noted that the low efficiency of `LS-tempo` on periodic-only instances is the consequence of a comparatively smaller number of successful acquisition insertion using the CP-SAT solver. Even when allowing our method to run for up to an hour and with a larger fail limit for the solver, results on the large periodic-only instances solely improved the gap by at most 1 %.

5.3.2 Impact of the CP-SAT Solver

We assessed the advantages of using Tempo inside our approach. Hereafter, `LS-tempo` refers to our method when it includes calls to Tempo and `Greedy` is our method without calls to Tempo.

Over Squillaci et al. instances, `LS-tempo`, the impact is significant. We observes a 0.9 % gap improvement in average, with `LS-tempo` giving better results on all but two instances. Averaging the gap makes the differences between the two solvers smaller, as 15 instances over the 32 are solved optimally with the base solver (see Table 4 for data on a subset of instances).

Over our set of instances, however, the advantage of using Tempo is not as clear. We observe no difference in the average gap of `LS-tempo` and `Greedy`. Indeed, the method makes a very high number of calls to Tempo with a very low success rate. In other words, the

Table 2 Profit and gap of the solutions found by `CPO`, `LNS` and `LS-tempo` over Squillaci et al.'s instances. Optimal profit is noted with a star (*). For the sake of clarity, only instances where at least one solver has not found the optimal solution are shown.

$\|\mathcal{R}\|$	OS	V	S	P	CPO		LNS		LS-tempo	
					profit	gap	profit	gap	profit	gap
500	500	0	0	0	**5070***	0.00	5069	<0.01	**5070***	0.00
1000	1000	0	0	0	10165	0.13	11382	0.03	**11536**	0.01
500	0	500	0	0	21176	0.01	20988	0.02	**21200**	0.01
1000	0	1000	0	0	40280	0.39	52024	0.21	**55367**	0.16
250	0	0	0	250	**4100**	0.03	4018	0.05	3946	0.07
500	0	0	0	500	10856	0.12	**11072**	0.10	10512	0.15
1000	0	0	0	1000	14643	0.33	**15415**	0.29	13953	0.36
570	120	150	270	30	22379	0.02	22409	0.02	**22585**	0.01
1140	240	300	540	60	13075	0.19	**16223**	<0.01	16220	<0.01
500	500	0	0	0	**9270***	0.00	9263	<0.01	9267	<0.01
1000	1000	0	0	0	13409	0.26	17920	0.02	**18055**	0.01
250	0	250	0	0	**18500***	0.00	18456	<0.01	**18500***	0.00
500	0	500	0	0	**36592**	0.01	36304	0.02	36420	0.02
1000	0	1000	0	0	51732	0.29	67868	0.07	**68564**	0.06
250	0	0	0	250	**6445***	0.00	6442	<0.01	6388	0.01
500	0	0	0	500	12024	0.10	**12307**	0.08	11662	0.13
1000	0	0	0	1000	17410	0.35	**19385**	0.27	18055	0.32
570	120	150	270	30	22132	<0.01	22152	<0.01	**22187**	<0.01
1140	240	300	540	60	31666	0.29	43828	0.02	**43899**	0.02
				Average		0.134		0.064		0.070

Table 3 Profit and gap of the solutions found by `CPO`, `LNS` and `LS-tempo` over our instances. Optimal profit is noted with a star (*).

Set	$\|\mathcal{R}\|$	CPO		LNS		LS-tempo	
		profit	gap	profit	gap	profit	gap
`antilles`	50	**20104**	<0,01	20050,6	0,01	20090	<0,01
	100	**36480**	0,02	36192,6	0,03	36328	0,02
	250	71324	0,30	73033,8	0,28	**74035**	0,27
	500	81745	0,60	**89858,4**	0,56	88222	0,57
`sahara`	50	**17751**	0,01	17725	0,01	17743	0,01
	100	**38702**	0,03	38393,6	0,04	38570	0,03
	250	73972	0,32	75885,2	0,30	**76020**	0,30
	500	76581	0,61	**87462,4**	0,56	85559	0,57
`fountains`	50	**20304**	0,01	20265,4	0,01	20287	0,01
	100	**37048**	0,02	36897,8	0,02	37014	0,02
	250	90934	0,15	95060,6	0,11	**97721**	0,08
	500	106102	0,52	141014,6	0,37	**141331**	0,37
`military`	50	**22467***	0.00	22442,2	<0,01	**22467***	0.00
	100	**38489***	0.00	38447	<0,01	**38489***	0.00
	250	102849	<0,01	101945,6	0,01	**102868**	<0,01
	500	206813	0,03	207729,2	0,03	**211080**	0,01
`world heritage`	50	**18222***	0.00	18213,8	<0,01	**18222***	0.00
	100	**36699**	<0,01	36655,4	<0,01	36691	<0,01
	250	103648	0,02	103253,8	0,02	**103860**	0,01
	500	185196	0,09	192808,4	0,05	**197782**	0,03
		Average	0.136		0.120		0.115

TSPTW sub-problems are either easily solved by the greedy insertion, or are very hard to solve, and hence using a complete solver is not worth the effort. Full details on those results are given in appendix B.

Table 4 Average gap, average number of restart (r), average number of calls to Tempo (t), average success rate when calling Tempo (s), time spent in the solver (Ts) and time ratio spent in modelling the sub-problem with the solver (Tm) for various fail limits. For Squillaci et al. instances, we only used the instances where at least one of the solvers did not find the optimal solution

Fail limit	Squillaci et al. instances						Our instances					
	Gap	r	t	s	Ts	Tm	Gap	r	t	s	Ts	Tm
0	0.089	2507	-	-	-	-	0.115	45772	-	-	-	-
50	0.070	337	153300	47.6%	32%	15%	0.115	5302	1341790	7.68%	21%	54%
100	0.071	264	104603	48.0%	36%	13%	0.115	3047	851625	7.81%	26%	48%
500	0.071	251	122940	50.1%	51%	8%	0.115	5418	1046200	8.10%	41%	40%
1000	0.074	233	114420	50.6%	56%	7%	0.115	4649	946012	8.15%	48%	34%
10000	0.083	208	123406	50,9%	65%	5%	0.116	3665	525106	8.20%	66%	19%
100000	0.089	188	113339	51,5%	69%	5%	0.118	3364	414748	8.26%	73%	14%

Table 4 shows the impact of the fail limit given to Tempo. We can see this parameter is not sensitive, since as a fail limit of 50 and 1000 yields roughly the same results. Beyond this value, the gap increases as well as the time spent calling the solver. More importantly, we note that the simple fact of using Tempo with a small fail limit dramatically improves the gap. It means that solutions to the TSPTW sub-problem solutions can be easily solved with a small tree search in about half the cases. Therefore, there is no use in increasing the fail limit too much, indicating that heuristics designed to solve TSPTW could yield even better performances than using a complete CP-SAT solver.

5.3.3 Memory Management Impact

Finally, Table 5 shows the comparison between the profit and the gap yielded by `CPO`, `LS-tempo` and `Greedy` when taking into account the memory constraints and computing a download plan. We can see that `CPO` loses 3.5 % on the average gap when adding these constraints, while our solver performance does not change. It means that memory constraints are not tight, and that the 6 ground stations are sufficient for the parameters we used for our instance generator. Those results are consistent with other work in the literature, which explains why memory management is often overlooked or simplified when dealing with AEOSSP [34].

6 Conclusion

In this paper, we presented a solver for the AEOSSP that is designed for our industrial partner Prométhée's the upcoming constellation of 20 AEOSs. This solver is able to tackle two sub-problems of AEOSSP, namely acquisition selection and download planning. Acquisition selection is solved by a local search procedure, which is based on greedily solving a sequence of TSPTW, with occasional help from a CP solver in order to get a maximal-profit sequence of acquisitions for each satellite. Download planning is seen as a sequence of knapsack problems, that are solved greedily. When we include the download planning, experiments showed that our solver find better solution than IBM CPO on average, especially on larger instances. Overall, with and without the download planning, IBM CPO unsurprisingly does not scale very much. We show that including calls to a CP-SAT solver inside our solver lead to an improvement on average, but these improvements depend on the type of instances. Also, the type of instances is important to choose which solver to use. Especially we show that while our solver is not good on period requests only, it exhibits state-of-the-art capabilities on other types of instances.

Table 5 Result on our set of instance with memory management

Set	$\|\mathcal{R}\|$	CPO		Greedy		LS-tempo	
		profit	gap	profit	gap	profit	gap
antilles	50	**20102**	<0.01	20089	<0.01	20087	0.01
	100	36323	0.02	36308	0.02	**36326**	0.02
	250	69674	0.31	**74115**	0.27	73929	0.27
	500	67979	0.67	**89069**	0.56	88329	0.57
sahara	50	**17748**	0.01	17736	0.01	17744	0.01
	100	38517	0.03	**38568**	0.03	38564	0.03
	250	73805	0.32	**76101**	0.30	76036	0.30
	500	68814	0.65	**86572**	0.56	85834	0.57
fountains	50	**20301**	0.01	20282	0.01	20289	0.01
	100	36939	0.02	36991	0.02	**37012**	0.02
	250	84771	0.21	97456	0.09	**97703**	0.09
	500	93629	0.58	**142525**	0.36	141493	0.36
military	50	**22467***	0.00	**22467***	0.00	**22467***	0.00
	100	**38489***	0.00	38488	<0.01	**38489***	0.00
	250	100592	0.03	102838	<0.01	**102865**	<0.01
	500	165914	0.23	211001	0.02	**211088**	0.01
world heritage	50	**18222***	0.00	18217	<0.01	**18222***	0.00
	100	**36692**	<0.01	36686	<0.01	36691	<0.01
	250	98776	0.06	103766	0.01	**103856**	0.01
	500	152770	0.25	197722	0.03	**197794**	0.03
	Average	0.170		0.115		0.116	

Finally, further work could focus on adding different moves and variations to our solver. We did experiment the resolution of the orienteering problem with our CP solver, but it does not scale for this problem today. We also would like to use it for the insertion of several acquisitions, in order to make its use more efficient by decreasing the number of calls.

References

1 Nicola Bianchessi and Giovanni Righini. Planning and Scheduling Algorithms for the COSMO-SkyMed Constellation. *Aerospace Science and Technology*, 12(7):535–544, 2008. doi:10.1016/j.ast.2008.01.001.

2 Jiawei Chen, Ming Chen, Jun Wen, Lei He, and Xiaolu Liu. A Heuristic Construction Neural Network Method for the Time-Dependent Agile Earth Observation Satellite Scheduling Problem. *Mathematics*, 10(19):3498, 2022. doi:10.3390/math10193498.

3 Xiaogeng Chu, Yuning Chen, and Lining Xing. A Branch and Bound Algorithm for Agile Earth Observation Satellite Scheduling. *Discrete Dynamics in Nature and Society*, 2017, 2017. doi:10.1155/2017/7345941.

4 J. F. Cordeau and G. Laporte. Maximizing the Value of an Earth Observation Satellite Orbit. *The Journal of the Operational Research Society*, 56(8):962–968, 2005. doi:10.1057/PALGRAVE.JORS.2601926.

5 Kaikai Cui, Junhua Xiang, and Yulin Zhang. Mission Planning Optimization of Video Satellite for Ground Multi-Object Staring Imaging. *Advances in Space Research*, 61(6):1476–1489, 2018. doi:10.1016/j.asr.2017.10.056.

6 Duncan Eddy and Mykel J. Kochenderfer. A Maximum Independent Set Method for Scheduling Earth-Observing Satellite Constellations. *Journal of Spacecraft and Rockets*, 58(5):1416–1429, 2021. doi:10.2514/1.A34931.

7 Yanxiang Feng, Ruipeng Zhang, Sida Ren, Shuailin Zhu, and Yikang Yang. A Distributed Approach for Time-Dependent Observation Scheduling Problem in the Agile Earth Observation Satellite Constellation. *Remote Sensing*, 15(7):1761, 2023. doi:10.3390/rs15071761.

8 Mohamed Elamine Galloua, Shuai Li, and Jiahao Cui. Earth Observation Satellite Imaging Task Scheduling with Metaheuristics: Multi-Level Clustering and Priority-Driven Pre-Scheduling. *Advances in Space Research*, 2024. doi:10.1016/j.asr.2024.11.023.

9 Lei He, Xiaolu Liu, Gilbert Laporte, Yingwu Chen, and Yingguo Chen. An Improved Adaptive Large Neighborhood Search Algorithm for Multiple Agile Satellites Scheduling. *Computers & Operations Research*, 100:12–25, 2018. doi:10.1016/j.cor.2018.06.020.

10 Lijun He, Ben Liang, Jiandong Li, and Min Sheng. Joint Observation and Transmission Scheduling in Agile Satellite Networks. *IEEE Transactions on Mobile Computing*, 21(12):4381–4396, 2022. doi:10.1109/TMC.2021.3076088.

11 Chae-Hyeon Kim, Sung Jun Kim, and Han-Lim Choi. Optimal Integrated Scheduling of Observation and Download Tasks for Multiple Satellites with Memory Constraints. In *2024 IEEE 20th International Conference on Automation Science and Engineering (CASE)*, pages 2287–2294, 2024. doi:10.1109/CASE59546.2024.10711478.

12 Junhong Kim, Doo-Hyun Cho, Jaemyung Ahn, and Han-Lim Choi. Task Scheduling of Multiple Agile Satellites with Transition Time and Stereo Imaging Constraints, 2019. doi:10.48550/arXiv.1912.00374.

13 Michel Lemaître, Gérard Verfaillie, Frank Jouhaud, Jean-Michel Lachiver, and Nicolas Bataille. Selecting and Scheduling Observations of Agile Satellites. *Aerospace Science and Technology*, 6(5):367–381, 2002. doi:10.1016/S1270-9638(02)01173-2.

14 Rich Levinson, Sreeja Nag, and Vinay Ravindra. Agile Satellite Planning for Multi-Payload Observations for Earth Science, 2021. doi:10.48550/arXiv.2111.07042.

15 Hai Li, Yongjun Li, Yuanhao Liu, Kai Zhang, Xin Li, Yu Li, and Shanghong Zhao. A Multi-Objective Dynamic Mission-Scheduling Algorithm Considering Perturbations for Earth Observation Satellites. *Aerospace*, 11(8):643, 2024. doi:10.3390/aerospace11080643.

16 Longmei Li, Feng Yao, Ning Jing, and Michael Emmerich. Preference Incorporation to Solve Multi-Objective Mission Planning of Agile Earth Observation Satellites. In *Proceedings of the 2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 1366–1373, 2017. doi:10.1109/CEC.2017.7969463.

17 Wei-Chen Lin and Da-Yin Liao. A Tabu Search Algorithm for Satellite Imaging Scheduling. In *2004 IEEE International Conference on Systems, Man and Cybernetics*, volume 2, pages 1601–1606, 2004. doi:10.1109/ICSMC.2004.1399860.

18 Wei-Cheng Lin, Chung-Yang Liu, Da-Yin Liao, and Yung-Yao Lee. Daily Imaging Scheduling of an Earth Observation Satellite. In *Proceedings of the 2003 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, volume 2, pages 1886–1891 vol.2, 2003. doi:10.1109/ICSMC.2003.1244686.

19 Lorena Linares, Rafael Vazquez, Federico Perea, and Jorge Galán-Vioque. A Mixed Integer Linear Programming Model for Resolution of the Antenna-Satellite Scheduling Problem. *IEEE Transactions on Aerospace and Electronic Systems*, 60(1):463–473, 2024. doi:10.1109/TAES.2023.3326422.

20 Lihao Liu, Zhenghong Dong, Haoxiang Su, and Dingzhan Yu. A Study of Distributed Earth Observation Satellites Mission Scheduling Method Based on Game-Negotiation Mechanism. *Sensors*, 21(19):6660, 2021. doi:10.3390/s21196660.

21 Xiaolu Liu, Gilbert Laporte, Yingwu Chen, and Renjie He. An Adaptive Large Neighborhood Search Metaheuristic for Agile Satellite Scheduling with Time-Dependent Transition Time. *Computers & Operations Research*, 86:41–53, 2017. doi:10.1016/j.cor.2017.04.006.

22 Guansheng Peng, Guopeng Song, Yongming He, Jing Yu, Shang Xiang, Lining Xing, and Pieter Vansteenwegen. Solving the Agile Earth Observation Satellite Scheduling Problem With Time-Dependent Transition Times. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 52(3):1614–1625, 2022. doi:10.1109/TSMC.2020.3031738.

23 Vishwa Shah, Vivek Vittaldev, Leon Stepan, and Cyrus Foster. Scheduling the World's Largest Earth-Observing Fleet of Medium-Resolution Imaging Satellites. In *11th International Workshop on Planning and Scheduling for Space (IWPSS)*, page 6, 2019.

24 Samuel Squillaci. *Mission plan optimization for a constellation of Earth observation satellites (Optimisation de plans de mission pour une constellation de satellites d'observation de la Terre)*. PhD thesis, Office national d'études et de recherches aérospatiales (ONERA) and Institut supérieur de l'aéronautique et de l'espace (ISAE-SUPAERO), Toulouse, France, 2023.

25 Samuel Squillaci, Cédric Pralet, and Stéphanie Roussel. Scheduling Complex Observation Requests for a Constellation of Satellites: Large Neighborhood Search Approaches. In *Proceedings of the 20th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, 2023. doi:10.1007/978-3-031-33271-5_29.

26 Samuel Squillaci, Stéphanie Roussel, and Cédric Pralet. Parallel Scheduling of Complex Requests for a Constellation of Earth Observing Satellites. In *PAIS 2022*, pages 100–113. IOS Press, 2022. doi:10.3233/FAIA220068.

27 Haiquan Sun, Wei Xia, Zhilong Wang, and Xiaoxuan Hu. Agile Earth Observation Satellite Scheduling Algorithm for Emergency Tasks Based on Multiple Strategies. *Journal of Systems Science and Systems Engineering*, 30(5):626–646, 2021. doi:10.1007/s11518-021-5506-4.

28 Panwadee Tangpattanakul, Nicolas Jozefowiez, and Pierre Lopez. A Multi-Objective Local Search Heuristic for Scheduling Earth Observations Taken by an Agile Satellite. *European Journal of Operational Research*, 245(2):542–554, 2015. doi:10.1016/j.ejor.2015.03.011.

29 Christopher G. Valicka, Deanna Garcia, Andrea Staid, Jean-Paul Watson, Gabriel Hackebeil, Sivakumar Rathinam, and Lewis Ntaimo. Mixed-Integer Programming Models for Optimal Constellation Scheduling given Cloud Cover Uncertainty. *European Journal of Operational Research*, 275(2):431–445, 2019. doi:10.1016/j.ejor.2018.11.043.

30 Pieter Vansteenwegen, Wouter Souffriau, Greet Vanden Berghe, and Dirk Van Oudheusden. Iterated Local Search for the Team Orienteering Problem with Time Windows. *Computers & Operations Research*, 36(12):3281–3290, 2009. doi:10.1016/j.cor.2009.03.008.

31 Petr Vilím, Philippe Laborie, and Paul Shaw. Failure-Directed Search for Constraint-Based Scheduling. In *Proceedings of the 12th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, pages 437–453, 2015. doi:10.1007/978-3-319-18008-3_30.

32 Jianjiang Wang, Erik Demeulemeester, and Dishan Qiu. A Pure Proactive Scheduling Algorithm for Multiple Earth Observation Satellites under Uncertainties of Clouds. *Computers & Operations Research*, 74:1–13, 2016. doi:10.1016/j.cor.2016.04.014.

33 Xinwei Wang, Guopeng Song, Roel Leus, and Chao Han. Robust Earth Observation Satellite Scheduling With Uncertainty of Cloud Coverage. *IEEE Transactions on Aerospace and Electronic Systems*, 56(3):2450–2461, 2020. doi:10.1109/TAES.2019.2947978.

34 Xinwei Wang, Guohua Wu, Lining Xing, and Witold Pedrycz. Agile Earth Observation Satellite Scheduling Over 20 Years: Formulations, Methods, and Future Directions. *IEEE Systems Journal*, 15(3):3881–3892, 2021. doi:10.1109/JSYST.2020.2997050.

35 Damien Wojtowicz. AEOSSP Instance Generator. Dataset, France Relance grant (project JAPETUS), swhId: swh:1:dir:c1c87cf9c3a3e7ffe616bdffdc9c52af9e30cb35 (visited on 2025-07-24). URL: https://github.com/ssquilla/Earth_Observing_Satellites_benchmarks, doi:10.4230/artifacts.24101.

36 Fatos Xhafa and Andrew W.H. Ip. Optimisation Problems and Resolution Methods in Satellite Scheduling and Space-Craft Operation: A Survey. *Enterprise Information Systems*, 15(8):1022–1045, 2021. doi:10.1080/17517575.2019.1593508.

37 Wenyuan Zhang and Gangtie Zheng. Scheduling an Agile Multipayload Earth-Observing Satellite. *Journal of Spacecraft and Rockets*, 61(1):143–156, 2024. doi:10.2514/1.A35726.

A Instances Details

Table 6 Generated instances details. It shows, for each instance, its set of POIs, the number of requests $|\mathcal{R}|$, of periodic requests P, of multi-mesh requests M, of downloads $|\mathcal{W}|$, of acquisition opportunities $|\mathcal{A}|$, of acquisitions needed to satisfy all the requests $s(\mathcal{A})$, of connected components nc, the average size of the components sc, and the average number of acquisition opportunities per target ar.

Set	$\|\mathcal{R}\|$	P	M	$\|\mathcal{W}\|$	$\|\mathcal{A}\|$	$s(\mathcal{A})$	nc	sc	ar
antilles	50	11	2	60	1017	70	19	54	15
	100	18	4	60	1993	133	19	105	15
	250	54	17	60	5313	370	19	280	14
	500	88	29	60	10510	717	19	553	15
sahara	50	3	4	74	872	59	22	40	15
	100	18	10	74	1857	147	23	81	13
	250	43	19	89	4455	361	23	193	13
	500	89	39	74	9043	721	23	393	13
fountains	50	9	3	82	1692	67	31	55	25
	100	14	4	82	3273	133	32	102	25
	250	51	14	82	8372	359	32	262	23
	500	112	33	82	17037	748	32	532	23
military	50	10	6	159	1503	83	104	14	18
	100	22	2	164	2648	140	108	24	19
	250	48	4	165	6735	347	141	48	19
	500	97	40	163	14039	741	163	86	19
world heritage	50	4	4	158	1729	67	119	14	26
	100	17	1	160	2952	130	135	22	23
	250	40	20	164	8153	360	136	59	23
	500	85	38	166	15979	717	133	120	22

Table 7 Squillaci et al. instances details. It shows, for each instance, the number of request $|\mathcal{R}|$, of one-shot requests OS, video requests V, stereoscopic requests S and periodic requests P, of downloads $|\mathcal{W}|$, of acquisition opportunities $|\mathcal{A}|$, of acquisitions needed to satisfy all the requests s($\mathcal{A}$), of connected components nc, the average size of the components sc(when $sc > 1$), and of acquisition opportunities per target ar. The first part of this table is about the concentrated instances and the second part is about the spread instances.

$\|\mathcal{R}\|$	OS	V	S	P	$\|\mathcal{W}\|$	$\|\mathcal{A}\|$	s($\mathcal{A}$)	nc	sc	ar
50	50	0	0	0	123	2607	50	124	23	50
250	250	0	0	0	123	13149	250	127	112	52
500	500	0	0	0	123	25646	500	116	221	51
1000	1000	0	0	0	123	51189	1000	117	437	51
50	0	50	0	0	123	2436	50	115	21	46
250	0	250	0	0	123	12162	250	114	107	48
500	0	500	0	0	123	25554	500	118	216	51
1000	0	1000	0	0	123	49149	1000	116	424	49
50	0	0	0	50	123	576	150	26	18	3
250	0	0	0	250	123	2367	750	25	91	3
500	0	0	0	500	123	4794	1501	28	168	3
1000	0	0	0	1000	123	9777	3000	27	359	3
57	12	15	27	3	123	2125	90	112	19	22
285	60	75	135	1	123	10327	450	115	90	23
570	120	150	270	30	123	19203	900	117	164	21
1140	240	300	540	60	123	45985	1800	111	414	25
50	50	0	0	0	123	2438	50	115	21	46
250	250	0	0	0	123	12269	250	118	104	49
500	500	0	0	0	123	24475	500	119	206	49
1000	1000	0	0	0	123	49035	1000	121	405	50
50	0	50	0	0	123	2427	50	115	21	46
250	0	250	0	0	123	12190	250	118	103	48
500	0	500	0	0	123	24326	500	119	204	48
1000	0	1000	0	0	123	48686	1000	120	406	49
50	0	0	0	50	123	568	150	27	17	3
250	0	0	0	250	123	2348	750	27	83	3
500	0	0	0	500	123	4594	1501	28	161	3
1000	0	0	0	1000	123	9162	3004	28	324	3
57	12	15	27	3	123	2025	90	115	18	21
285	60	75	135	15	123	10467	451	119	88	23
570	120	150	270	30	123	20820	900	118	176	23
1140	240	300	540	60	123	41693	1800	119	350	23

B Results Tables

Table 8 shows the profit and the gap yielded by both methods on Squillaci et al. instances, while Table 8shows it on our instances, both using a fail limit of 50 for Tempo. The gap was computed using an upper bound calculated with the CPO model. Thanks to a naive upper bound (sum of the profit of the best opportunities for each request), the solver can prove optimality on trivial instances.

Over Squillaci et al. instances, `LS-tempo` has better results on all instances but 2, with an improvement of 0.9 % gap in average. Averaging the gap make the differences between the two solvers smaller, as 15 instances over the 32 are solved optimal with the base solver (see Table 4 for data on a subset of instances). We can see the number of restart drops with the

number of requests, especially for instances with periodic requests only. The same remark holds for the Tempo calls success rate. We note that for instance with 1000 video requests with concentrated target, tempo has only 4 % of success, nevertheless the result of `LS-tempo` is much better than `Greedy`.

Over our set of instances, the use advantage of using Tempo is not as clear. At first glance, it seems that there is a difference in the results between regional and worldwide instances. Using Tempo always seems to give the best solution, but the difference are in fact very small and the average gap for `LS-tempo` and `Greedy` are the same. Compared to the other instance set the number of restart and calls to Tempo is very high. On the contrary the success rate is very low, especially on regional datasets. It seems that instance of the sub-problem are either easily solve with the greedy insertion, or are very hard and cannot be solved with a small fail limit.

Table 8 Profit, gap and number of restarts (r) found using `LS-tempo` and `Greedy` over our instances. Average values over 5 runs. A star (*) means the profit is optimal, a dash ("–") means the instance was solved optimally during the initial descent.

$\|\mathcal{R}\|$	OS	V	S	P	Greedy			LS-tempo				
					profit	gap	r	profit	gap	r	t	s
50	50	0	0	0	**390***	0.00	–	**390***	0.00	–	–	–
250	250	0	0	0	**1655***	0.00	–	**1655***	0.00	–	12	100%
500	500	0	0	0	**5070***	0.00	–	**5070***	0.00	–	26	100%
1000	1000	0	0	0	11440	0.02	601	**11536**	0.01	322	68427	79%
50	0	50	0	0	**4060***	0.00	–	**4060***	0.00	–	1	100%
250	0	250	0	0	**7440***	0.00	–	**7440***	0.00	–	13	100%
500	0	500	0	0	21040	0.01	3054	**21200**	0.01	1841	98269	89%
1000	0	1000	0	0	51524	0.22	441	**55367**	0.16	76	1439698	4%
50	0	0	0	50	**780***	0.00	11	**780***	0.00	–	3	100%
250	0	0	0	250	3777	0.11	9475	**3946**	0.07	39	37802	14%
500	0	0	0	500	10120	0.18	4997	**10512**	0.15	5	36189	5%
1000	0	0	0	1000	**13996**	0.36	1992	13953	0.36	1	34562	1%
57	12	15	27	3	**2010***	0.00	–	**2010***	0.00	–	1	100%
285	60	75	135	1	**6265***	0.00	3	**6265***	0.00	–	21	100%
570	120	150	270	30	22092	0.03	2414	**22585**	0.01	524	97259	60%
1140	240	300	540	60	16126	0.01	234	**16220**	<0.01	17	24299	36%
50	50	0	0	0	**720***	0.00	3	**720***	0.00	–	4	100%
250	250	0	0	0	**4625***	0.00	81	**4625***	0.00	–	15	100%
500	500	0	0	0	9138	0.01	1952	**9267**	<0.01	870	64068	81%
1000	1000	0	0	0	17833	0.02	444	**18055**	0.01	214	66872	69%
50	0	50	0	0	**2880***	0.00	7	**2880***	0.00	–	2	100%
250	0	250	0	0	**18500***	0.00	65	**18500***	0.00	7	137	89%
500	0	500	0	0	35416	0.04	3103	**36420**	0.02	1443	223688	65%
1000	0	1000	0	0	66660	0.08	317	**68564**	0.06	155	436114	17%
50	0	0	0	50	**1060***	0.00	–	**1060***	0.00	–	4	100%
250	0	0	0	250	6139	0.05	9243	**6388**	0.01	82	31641	41%
500	0	0	0	500	11177	0.17	4397	**11662**	0.13	5	28955	7%
1000	0	0	0	1000	**18078**	0.32	1964	18055	0.32	1	28728	2%
57	12	15	27	3	**1765***	0.00	–	**1765***	0.00	–	1	100%
285	60	75	135	15	**12210***	0.00	24	**12210***	0.00	1	43	95%
570	120	150	270	30	22048	0.01	2468	**22187**	<0.01	701	110152	84%
1140	240	300	540	60	42908	0.04	480	**43899**	0.02	105	85822	62%
				Average		0.053			0.042			67.6%

Table 9 Profit, gap and number of restarts (r) found using `LS-tempo` and `Greedy` over our instances. Average values over 5 runs. A star (*) means the profit is optimal, a dash ("-") means the instance was solved optimally during the initial descent.

POIs	$\|\mathcal{R}\|$	Greedy			LS-tempo				
		profit	gap	r	profit	gap	r	t	s
antilles	50	**20090**	<0.01	167166	**20090**	<0.01	13186	3055040	6%
	100	36307	0.02	82066	**36328**	0.02	2396	2296988	4%
	250	**74129**	0.27	22835	74035	0.27	255	2263862	1%
	500	**88913**	0.56	10785	88222	0.57	119	1983807	0%
sahara	50	17736	0.01	242973	**17743**	0.01	11743	1785669	13%
	100	**38575**	0.03	79356	38570	0.03	1504	1400336	4%
	250	75919	0.30	19381	**76020**	0.30	212	1898979	1%
	500	**86522**	0.56	10690	85559	0.57	85	1532961	0%
fountains	50	20282	0.01	122542	**20287**	0.01	10309	1986404	7%
	100	36989	0.02	39253	**37014**	0.02	1806	1824849	4%
	250	97433	0.09	8422	**97721**	0.08	112	891194	2%
	500	**142314**	0.36	3564	141331	0.37	36	1342948	0%
military	50	**22467***	0.00	44987	**22467***	0.00	30372	585242	19%
	100	38488	<0.01	15250	**38489***	0.00	9135	418030	29%
	250	102838	<0.01	3742	**102868**	<0.01	391	324473	12%
	500	211019	0.01	1116	**211080**	0.01	35	186169	6%
world heritage	50	18221	<0.01	23588	**18222***	0.00	19290	760620	24%
	100	36684	<0.01	13974	**36691**	<0.01	4706	1214798	13%
	250	103770	0.01	2714	**103860**	0.01	310	717050	6%
	500	197736	0.03	1040	**197782**	0.03	39	366371	3%
	Average		0.115			0.115			7.7%

Conflict Analysis Based on Cutting-Planes for Constraint Programming

Robbin Baauw
Delft University of Technology, The Netherlands

Maarten Flippo
Delft University of Technology, The Netherlands

Emir Demirović
Delft University of Technology, The Netherlands

Abstract

This paper introduces a novel constraint learning mechanism for Constraint Programming (CP) solvers that integrates cutting planes reasoning into the conflict analysis procedure. Drawing inspiration from Lazy Clause Generation (LCG), our approach, named Lazy Linear Generation (LLG), can generate linear integer inequalities to prune the search space, rather than propositional clauses as in LCG. This combines the strengths of constraint programming (strong propagation through global constraints) with cutting-planes reasoning. We present linear constraint explanations for various arithmetic constraints and the element constraint. An experimental evaluation shows that the improved generality of linear constraints has a practical impact on a CP solver by reducing the number of encountered conflicts in 45% of our benchmark instances. Our analysis and prototype implementation show promising results and are an important step towards a new paradigm to make constraint programming solvers more effective.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases constraint programming, learning, conflict analysis

Digital Object Identifier 10.4230/LIPIcs.CP.2025.4

Supplementary Material *Software (Source code and Evaluation)*: `https://doi.org/10.5281/zenodo.15591035`

Funding *Maarten Flippo*: Supported by the project "Towards a Unification of AI-Based Solving Paradigms for Combinatorial Optimisation" (OCENW.M.21.078) of the research programme "Open Competition Domain Science – M" which is financed by the Dutch Research Council (NWO).

1 Introduction

Constraint Programming [39] (CP) is an important paradigm for solving combinatorial optimization problems. It has applications in many domains, including resource allocation [41, 37], scheduling [32, 1], and verification [31, 3]. CP solvers use backtracking search algorithms to find solutions to models. Key to a good backtracking search algorithm is the ability to identify areas of the search space that do not contain solutions. Modern CP solvers use a combination of two types of reasoning to achieve this. The first is *propagation*, which is the process of identifying values that, based on the constraints in the problem, can never be part of a solution. The second is *conflict analysis*, which adds new constraints to the solver during the search process, which enables more propagation to occur.

Deriving new constraints is well-known to be beneficial to backtracking search algorithms [9]. Much work has been done to implement constraint learning effectively in CP solvers [20, 29, 38, 21, 24, 40]. Of the approaches, Lazy Clause Generation (LCG) [29, 38], is the most wide-spread, implemented by solvers such as OR-Tools [30] and Chuffed [8]. For many problems, constraint learning is crucial [38, 35] to the performance of a solver, as the learned constraints prune large parts of the search space.

© Robbin Baauw, Maarten Flippo, and Emir Demirović;
licensed under Creative Commons License CC-BY 4.0

All these solvers have in common that they reason over clausal constraints. However, other types of constraints can also be learned by CP solvers [40]. For paradigms other than CP, work has also been done exploring the learning of pseudo-Boolean (PB) constraints [12, 17] and integer linear constraints [27, 19, 2]. These systems have the potential of learning stronger constraints than clauses, although in practice more scientific and engineering efforts are needed to make these approaches as mature as the more studied clause-learning algorithms.

An example of conflict analysis on integer linear constraints can be found in the Integer Linear Programming (ILP) solver IntSat [27], which serves as a starting point of our approach. It stands out from other ILP solvers because it does not reason using the LP relaxation of the problem. Instead, it uses cutting planes reasoning and a generalized CDCL [22] algorithm, which combines integer linear constraints to derive new (implied) integer linear constraints. The method is promising, as it is already competitive with other state-of-the-art ILP solvers such as Gurobi [16]. One major difference between IntSat's conflict analysis procedure and a clausal conflict analysis procedure, is that in the former the analysis can fail to derive a new constraint that compactly describes the current conflict. Yet, the empirical evaluation shows that IntSat is effective despite this fact.

The effectiveness of cutting planes reasoning inspired our work with the question "How can CP solvers incorporate cutting planes reasoning?". As IntSat is heavily inspired by CDCL, this indicates that cutting planes reasoning could be incorporated in constraint programming similar to how LCG includes propositional CDCL. The major difference would be how high-level constraint inference is explained to the learning procedure, as the clausal explanation by LCG solvers are not applicable. We further observe that it may not be possible to generate a linear constraint as a reason for propagation without introducing new variables. This is in contrast to clausal explanations, where creating additional variables is supported but not required.

A CP solver that would come close to this idea is HaifaCSP [40], as it can do cutting planes reasoning to derive linear inequalities. However, it cannot explain propagations by arbitrary constraints as linear inequalities. As a result, as soon as a linear inequality interacts with another arbitrary constraint, the solver resorts to clausal learning. As we show in our experiments, the more clauses are present in the solver, the higher the chance that conflict analysis cannot derive a new linear inequality. To maximize the impact of the more general learning, we want the learning procedure to deal with linear inequalities as much as possible.

We present lazy linear constraint generation (LLG), an approach to use cutting planes reasoning within constraint programming. Our approach explains propagations with linear constraints, allowing propagations by arbitrary propagators to be used in the cutting planes conflict analysis procedure. To this end, we modify the conflict analysis procedure from the IntSat solver. We devise explanations for the integer multiplication, absolute value, truncating division, maximum, linear not equals, and element propagators. Our IntSat-based conflict analysis procedure combines the explanations to learn new linear constraints.

To determine whether the theoretical benefits of learning linear constraints have an impact in practice, we ran an empirical evaluation over 952 MiniZinc instances. In 443 of those, the solver terminates and learns at least one linear constraint. Among those instances, the conflict count is reduced, sometimes significantly. However, there are also many instances where the conflict analysis fails for the same reason as the IntSat conflict analysis fails. When we do learn a linear constraint, we show that it is generally more impactful than the clause that would be learned in that same situation. Additionally, we show that the CP propagators can be more effective with LLG than encoding the problem to linear constraints and solving with IntSat. Lastly, we show that resorting to clausal learning when no linear constraint can be learned remains essential, as omitting this step increases the number of conflicts significantly.

The rest of this paper is organized as follows: We start out with some background in Section 2, followed by a discussion of the related work in Section 3. Then, we will describe our contributions in Section 4. After that, we present our empirical evaluation in Section 5. Finally, we give our conclusions and outline ideas for future work in Section 6.

2 Background

2.1 Constraint Satisfaction Problem

A *constraint satisfaction problem* (CSP) is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where:

- $x \in \mathcal{X}$ is a *decision variable*,
- $\mathcal{D}(x) \in \mathcal{D}$ with $x \in \mathcal{X}$ is the *domain* of x, i.e. the set of values x can be assigned to,
- and $C \in \mathcal{C}$ is a *constraint*: a predicate over the variables that is either satisfied or violated.

We use range notation when the domain is a uninterrupted sequence of integers: $[l, u] = \{i | l \leq i \leq u\}$.

An *assignment* is a total function θ that maps every variable $x \in \mathcal{X}$ to a set $V \subseteq \mathcal{D}(x)$. If $|\theta(x_i)| > 1$, i.e. there is more than one possible value for x_i, then the assignment is referred to as a *partial assignment*. Otherwise, the assignment is called *total*. In this paper, unless we explicitly use the term "partial assignment", we refer to a total assignment. We abuse notation to say that $\theta(x) = v$ with $v \in \mathbb{Z}$ to mean that $\theta(x) = \{x\}$. If the assignment θ satisfies all the constraints in $\mathcal{C}$, then θ is called a *solution*. In this paper, we restrict ourselves to integer decision variables, i.e. $\forall x \in \mathcal{X} : \mathcal{D}(x) \subset \mathbb{Z}$, and we assume the domains are finite.

2.2 Constraint Programming

Constraint Programming (CP) is a paradigm for solving CSPs. CP solvers combine inference and search to find solutions to a CSP. The inference prunes the domain based on the constraints in the problem, and once no more inference can be done, the search splits the problem into subproblems to be solved independently. A *conflict* happens when there exists a variable $x \in \mathcal{X}$ such that $\mathcal{D}(x) = \emptyset$.

In a CP solver, constraints are enforced by propagators. A *propagator* is a function $p : \mathcal{D} \mapsto \mathcal{D}$ that takes the domain and removes values that do not exist in a solution. This means that $p(\mathcal{D}) \sqsubseteq \mathcal{D}$, where $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$ denotes that $\mathcal{D}_1$ is *stronger* than $\mathcal{D}_2$, i.e. for every $x \in \mathcal{X}$ it is the case that $\mathcal{D}_1(x) \subseteq \mathcal{D}_2(x)$. We highlight two types of constraints and their propagators.

- A *clause*, which is a disjunction of Boolean variables. It has the form $l_1 \vee \cdots \vee l_n$, where every l_i has a Boolean domain $\mathcal{D}(l_i) = \{0, 1\}$. This constraint requires at least one l_i to be 1. The propagator for a clause waits until $n - 1$ variables are fixed to 0, and then sets the final variable to 1.
- A *linear inequality* (also referred to as linear constraint in this paper) has the form $\sum w_i x_i \leq c$, where $w_i \in \mathbb{Z}$ and $c \in \mathbb{Z}$ are constants, and $x_i \in \mathcal{X}$ are decision variables. The propagator for this constraint performs bound propagation [5], as shown in Example 1.

▶ **Example 1.** Let x, y be integer decision variables with domains $\mathcal{D}(x) = [1, 5]$ and $\mathcal{D}(y) = [0, 2]$. The propagator for the constraint $x + 2y \geq 7$ can remove the values 1 and 2 from $\mathcal{D}(x)$ and the value 0 from $\mathcal{D}(y)$, because there are no solutions where $x = 2$ or $y = 0$.

2.3 Integer Linear Programs

An *Integer Linear Program* (ILP) is a CSP in which all constraints are linear inequalities. Given two linear constraints $A : a_1x_1 + \cdots + a_nx_n \leq a_0$ and $B : b_1x_1 + \cdots + b_nx_n \leq b_0$, their *linear combination* results in a new linear constraint $C : c_1x_1 + \ldots c_nx_n \leq c_0$ with $c_i = \alpha a_i + \beta b_i$ that is implied by $A \wedge B$. In the case where $c_i = 0$, we say that x_i has been *eliminated*. Note that we can always pick an α and β such that $c_i = 0$ when $a_i b_i < 0$, i.e., when the coefficients of the variable to be eliminated x_i have opposite signs in A and B.

The combination rule can be used by solvers like IntSat [27], which are inspired by Conflict-Driven Clause Learning (CDCL) [22]. IntSat keeps a trail of bounds in the form $\langle x \diamond v \rangle$, with $\diamond \in \{\leq, \geq\}$, that iteratively tighten the domain of the variables.

Algorithm 1 presents the pseudo-code for the conflict analysis procedure. Just like propositional CDCL, the trail gets traversed backwards. For every entry, it is checked whether reason constraint RC can be combined with the conflicting constraint CC to eliminate propagated variable V (lines 5 and 6). This elimination is feasible only if $V \in CC$ and the signs of the coefficients of V in CC and RC are opposite. If these conditions are met, the reason constraint RC is combined with the conflicting constraint CC (line 7). This process is repeated until the resulting constraint CC can propagate at an earlier decision level (*asserting*), or the previous decision is reached.

A notable difference with propositional CDCL, is that Algorithm 1 may fail to derive a linear constraint that is asserting at a prior decision level. In such cases, the solver resorts to performing resolution on bounds as a fallback strategy, similar to LCG conflict analysis (see next section), which is the clausal counterpart of linear combinations.

Algorithm 1 IntSat conflict analysis as described in [27].

Input: a set of linear constraints $\mathcal{C}$ and a trail T containing tuples (V, RC) representing a propagation of variable V by linear constraint RC
Output: a learned linear constraint or learned clause and corresponding backtrack level

1: $CC \leftarrow$ currently conflicting constraint in $\mathcal{C}$
2: ▷ ***Invariant**: CC is conflicting with the current assignment*
3: **while** top of trail is not a decision **do**
4: $(V, RC) \leftarrow$ PopTrail() ▷ Remove the last trail entry
5: **if** $V \notin CC$ **then continue end if** ▷ Variable not relevant
6: **if** $CC[V] \cdot RC[V] \geq 0$ **then continue end if** ▷ Signs equal, bound not relevant
7: $CC \leftarrow$ combination of CC and RC eliminating V
8: **for** backtrack level $\in$ 0..current decision level $-$ 1 **do**
9: **if** CC propagates a new bound at backtrack level **then**
10: **return** CC, backtrack level ▷ Early Backjump
11: **end if**
12: **end for**
13: **end while**
14: **return** Resolution-fallback$(\mathcal{C}, T)$

The reason for the inability to construct an asserting linear constraint is the *rounding problem*, which is illustrated by Example 2 (adapted from [27, Example 3.1]). As a consequence, linear conflict analysis can generate implied constraints that are not conflicting under the current partial assignment, even though the constraint that identified the conflict initially was conflicting with this assignment.

▶ **Example 2.** Consider constraints $c_1 : x + 2y \leq 2$ and $c_2 : x - 2y \leq 0$, with $\mathcal{D}(x) = [1, 3]$ and $\mathcal{D}(y) = [-5, 5]$. x's lower bound causes c_1 to propagate $2y \leq 1$, which after rounding becomes $y \leq 0$. This leads to a conflict with c_2. Combining c_2 and c_1, eliminating y, produces $2x \leq 2$, or $x \leq 1$. This new constraint is *not* conflicting with the current assignment, as $1 \in \mathcal{D}(x)$.

A CDCL-inspired ILP solver has to deal with the situation from Example 2 in some way. In Section 3 we highlight different approaches taken by various solvers.

2.4 Lazy Clause Generation

Lazy Clause Generation [28, 38] (LCG) is an approach to solving CSPs within the CP paradigm. It combines the domain propagation capabilities of CP solvers with the clause learning capabilities of SAT solvers.

There are two main parts to the integration. The first is the representation of the decision variables in a propositional formula. This is done by creating Boolean variables that map to unary constraints of the following form: $\langle x \diamond v \rangle$, where $x \in \mathcal{X}$, $\diamond \in \{\leq, \geq, \neq, =\}$, and $v \in \mathbb{Z}$. Such a unary constraint is called an *atomic constraint*. Every domain reduction in an LCG solver can be expressed by setting one or more atomic constraints to true.

The second part of the integration allows propagations done by propagators to be used during conflict analysis. Every propagation is explained by an implication $\bigwedge l_i \implies p$, where l_i and p are atomic constraints.

▶ **Example 3.** The explanations for the propagations in Example 1 are $\langle x \leq 5 \rangle \implies \langle y \geq 1 \rangle$ and $\langle y \leq 2 \rangle \implies \langle x \geq 3 \rangle$.

These explanations can be treated as clauses to integrate into the CDCL procedure, allowing the solver to learn clauses based on propagations done by propagators.

3 Related Work

It has long been established that learning can be beneficial to backtracking search algorithms [9]. The inclusion into CP solvers became popular when g-nogoods were introduced [20], which are a conjunction or disjunction of $\langle x \neq v \rangle$ constraints. LCG [29, 38] improves the conciseness of the explanations by introducing $\langle x \leq v \rangle$, $\langle x \geq v \rangle$, and $\langle x = v \rangle$ constraints, although the expressiveness of the two approaches is the same. G-nogoods are further generalized to c-nogoods [21, Chapter 5], implemented by Moore [24, Chapter 5]. A c-nogood can combine arbitrary constraints, not just atomic constraints. Finally, Veksler and Strichman [40] introduced the HaifaCSP solver, which is capable of learning constraints that are not clauses or conjunctions.

Of the CP learning approaches, LCG is the most widespread. Since its introduction, much work has been done to increase the impact that the constraint learning has on the search. The learned constraints can be minimized taking into account the semantics of atomic constraint [14], additional Boolean variables can be introduced [7], and explanations can be fine-tuned to improve the quality of learned constraints [33, 13].

Solvers that are specialized in specific constraint types can also deal with different forms of conflict analysis. Pseudo-Boolean solvers, which solve ILPs where all variables are Boolean, use conflict analysis [12, 17] to great effect. In this special case of ILPs, the rounding problem as described in Subsection 2.3 can be handled systematically. CutSat [18] and CutSat++ [6], which are CDCL-inspired solvers for general ILPs, restrict the search to side-step the rounding problem by only assigning variables to their lower- or upper-bound. Last, the IntSat [26, 27] solver accepts that conflict analysis can fail and does not always learn a new constraint.

There is an interesting comparison between IntSat and HaifaCSP. In HaifaCSP, constraints are combined to derive new constraints according to pre-specified rules. The rule for combining two linear constraints is identical to how IntSat operates. This means that, given a problem with only linear constraints, HaifaCSP is essentially an extension of IntSat. It is an extension, because, unlike IntSat, HaifaCSP will fall back to learning a clause if the derived linear constraint is not propagating.

To make effective use of specialized ILP solvers, much work focuses on translating arbitrary constraints into linear inequalities. The MiniZinc [25] toolchain can convert CP models to efficiently solvable ILPs [4]. These translations form a basis of the explanations we introduce in this paper. Much work has also been done to explain propagations in pseudo-Boolean equations [23], although the aim there is correctness in a proof, rather than propagation impact during search.

4 Our Contribution: Lazy Linear Generation (LLG)

We propose the Lazy Linear Generation (LLG) algorithm, an extension of LCG that learns asserting linear inequalities alongside clauses. A learned clause captures a propositional relationship between *atomic constraints*. LLG enhances clausal conflict analysis by incorporating cutting-planes analysis, allowing learned constraints to capture linear relationships between *variables* – something that cannot be expressed compactly using clauses.

Extending LCG to perform conflict analysis using linear inequalities aligns closely with the challenges addressed by IntSat [27] and HaifaCSP [40]. Similar to IntSat, LLG adapts the CDCL resolution algorithm to iteratively apply linear combinations to the explanations for conflicts or propagations, deriving asserting linear constraints that are added to the model. Additionally, LLG also employs resolution – specifically LCG – when the derived conflicting constraint does not conflict with the current assignment any longer.

The key distinction between our LLG approach and previous works lies in LLG's ability to linearly explain propagations and conflicts originating from arbitrary propagators, rather than being restricted solely to linear propagators. Our method leverages the advantages of both CP and a linear formulation of the problem, combining the propagation strength of CP propagators with more powerful linear conflict analysis. An example of a stronger CP propagator compared to its linear decomposition is shown in Example 4.

▶ **Example 4.** Consider the multiplication $a \cdot b = c$ constraint, with initial domains $a \in \{2,3\}, b \in \{1,4\}, c \in \{2,7\}$. A possible linear decomposition introduces auxiliary binary variables p_{a2}, p_{a3} and represents the constraint using the following constraints (normally expressed linearly using the big-M formulation):

$$p_{a2} + p_{a3} = 1 \quad \wedge \quad a = 2 \cdot p_{a2} + 3 \cdot p_{a3} \quad \wedge \quad p_{a2} \rightarrow 2b = c \quad \wedge \quad p_{a3} \rightarrow 3b = c$$

Under the current partial assignment, a CP propagator can infer, based on the upper bounds of a and c, that $b \leq 3$. However, since the auxiliary variables p_{a2} and p_{a3} remain unfixed, no further propagation can be achieved using the linear decomposition. The CP propagation can be explained using the expression $\langle a \geq 2 \rangle \wedge \langle c \geq 0 \rangle \rightarrow 2b \leq c$, which can be transformed into a linear inequality by introducing auxiliary variables for the conditions and employing the big-M formulation.

This new analysis, however, introduces additional challenges. First, it requires defining explanations for every propagation and conflict. Second, these explanations may require the dynamic introduction of auxiliary variables to accurately capture certain propagations

or conflicts. Finally, the learning procedure may generate new constraints that are not conflicting with the current assignment. This issue, which was previously caused only by rounding problems, can now also arise in cases where propagations or conflicts cannot be directly expressed as linear inequalities or when weak explanations are encountered.

The remainder of this section will outline LLG in more detail. Section 4.1 provides a concise overview of the algorithm, while Section 4.2 discusses methods for constructing explanations. Implementation details of the auxiliary variables are presented in Section 4.3, and Section 4.4 describes examples of linear explanations.

4.1 Conflict Analysis algorithm

The conflict analysis algorithm employed by LLG closely resembles the one presented in Algorithm 1, with several key modifications introduced in this section. The most notable distinction is that constraints are no longer exclusively linear. While the conflicting constraint CC and the reason constraint RC were previously guaranteed to be linear, conflicts and propagations must now be *explained* linearly for them to participate in conflict analysis. Unlike IntSat, there is no guarantee that CC or RC can actually be explained linearly, as certain linear explanations for propagations and conflicts may either be impractical or too expensive to create.

For example, we consciously do *not* convert clauses to linear inequalities, even though that is possible. Any clause $l_1 \vee \cdots \vee l_n$ with l_i being atomic constraints can be turned into the linear inequality $l_1 + \cdots + l_n \geq 1$. However, contrary to many other LCG solvers, our implementation does not create 0-1 variables for atomic constraints in clauses. I.e., a clause is a set of atomic constraints rather than a set of Boolean variables. This means that converting the clause to a linear constraint would require creating auxiliary variables for all atomic constraints l_i. As will become clearer in Subsection 4.3, creating auxiliary variables is not cheap, and we would need a lot of them. Future work can explore an implementation where this is feasible.

Subsection 4.4 presents a few linear explanations, and Appendix A gives the complete list of constraints and explanations we implemented. Given that, our conflict analysis differs from IntSat (as displayed in Algorithm 1) in the following areas:

1. First, rather than directly retrieving the currently conflicting constraint from $\mathcal{C}$ (line 1), we attempt to derive a linear explanation for the conflict. Either the conflict is explicitly identified by a propagator and described as a linear inequality, in which case CC is set to that linear inequality. Alternatively, the domain of a variable x became empty. In that case, the linear inequality $Ax \leq b$ explaining the last propagation on x is conflicting, so it is used as the conflicting constraint.
   If we cannot express the conflict with a linear constraint, then we proceed to the resolution fallback (line 14). A common example when this happens is when the conflicting constraint is a clause.
2. Similarly, instead of getting RC directly from the trail (line 4), RC becomes the linear explanation of the propagation. We also introduce an additional check: the combination step cannot proceed if we cannot create a linear explanation to assign to RC.
3. Finally, if we fail to learn an asserting linear constraint, we fall back to resolution just as IntSat does (line 14). The difference here is that IntSat does not remember clauses because it does not have a clausal propagator, so it just propagates the learned clause once and forgets about it. In our CP solver, we have a clausal propagator, so we remember the clause. Effectively, if we cannot learn an asserting linear inequality, the conflict analysis operates as a standard LCG conflict analysis does.

4.2 Linear explanations

A fundamental aspect of LLG is constructing linear inequality explanations for propagations. This section elaborates on several key aspects of constructing these explanations.

Inferred constraints. Firstly, it is important to highlight that the asserting inequality derived during conflict analysis is added to the model as a constraint. Consequently, all linear explanations that lead to the construction of this inequality must also be inequalities inferred from the model. This is the same requirement for a clausal explanation.

Explanation signs. Additionally, it is noteworthy that for a linear inequality to be an explanation, it must constrain the propagated variable in the same direction as the propagation being explained. The direction in which a linear constraint constrains a variable is determined by the sign of this variable in the inequality. More specifically, a linear constraint $5a - 4y \leq 0$ constrains the upper bound of a, and the lower bound of y. Example 5 provides an intuitive justification for this condition. From a technical perspective, this requirement arises directly from the properties of linear constraint elimination: to eliminate a variable by combining two inequalities, the variable must appear with opposite signs in both. If the propagated variable in the explanation has the same sign as in the conflicting constraint, it did not contribute to the conflict and is therefore not taken into account.

▶ **Example 5.** Consider a conflict analysis with current conflicting constraint $CC : -6x+3y \leq 10$. We aim to derive a CC that is asserting earlier in the search. This requires maximizing its left-hand side – achievable by minimizing x or maximizing y. Now, given the trail entry $(V, RC) = (x, -3x - 6y \leq 10)$, we aim to eliminate x. However, since x has a negative coefficient in RC, it constrains x's lower bound, not contributing to the goal of minimizing x.

Conditional explanations. An explanation may incorporate a conditional component to specify the conditions under which the explanation holds. Such conditional statements can be represented using Boolean *auxiliary variables*, which are binary variables indicating the truth value of a condition. For example, consider the conditional explanation $\langle a \leq 2 \rangle \rightarrow \langle b \geq 3 \rangle$. We can encode the condition by introducing a Boolean auxiliary variable p_1, defined as:

$$p_1 = \begin{cases} 1 & \text{if } a \leq 2 \\ 0 & \text{if } a > 2 \end{cases} \tag{1}$$

By ensuring consistent propagation of p_1 (discussed further in Section 4.3), the conditional explanation can be reformulated into a linear inequality: $b \geq 3 - M(1 - p_1)$. This formulation uses the "Big M" transformation to model the cases of $a \leq 2$ and $a > 2$ within a single explanation. Similar linearization techniques are described in more detail by [4].

4.3 Implementing auxiliary variables

The auxiliary variables associated with conditional explanations cannot be constructed before starting the search procedure, as it is unknown which auxiliary variables will be necessary. Thus, they must be introduced and propagated during the search. A key challenge is maintaining a consistent solver state at decision levels where these variables did not exist yet.

Adding auxiliary variables. When a new auxiliary variable is introduced, it often could have propagated at an earlier decision level, had it existed earlier. While we can propagate the auxiliary variable immediately upon its creation, this propagation is discarded upon backtracking. Ideally, we would retroactively introduce auxiliary variables at the start of the search, allowing them to propagate at the correct decision level. However, we argue that this retroactive introduction is unnecessary, as solver correctness does not require propagation at the earliest possible decision level.

Instead, we ensure that auxiliary variables are propagated as soon as possible. Upon backtracking, propagation normally starts by propagating the newly learned constraint, which may then trigger other propagators. LLG, however, prioritizes auxiliary variable propagation at this stage. This approach seeks to update the auxiliary variables – and, by extension, the variables that depend on them – so that they are identical to the state that would have been achieved had the auxiliary variables been present from the beginning of the search. If the solver backtracks further and discards this propagation, the auxiliary variable can again be re-propagated at the earlier level, ensuring correctness without complex trail manipulations. Example 6 illustrates this principle.

▶ **Example 6.** At decision level 10, an explanation introduces an auxiliary variable p, defined as $p \iff 6a + 7b > 3$. We can immediately propagate $\langle p \geq 1 \rangle$. Had p existed earlier, this propagation would have already occurred at level 3. If the solver now backtracks to decision level 5, the propagation $\langle p \geq 1 \rangle$ is discarded. We immediately re-propagate $\langle p \geq 1 \rangle$ at level 5 and subsequently reach the same state as if p had been propagated at decision level 3.

Evaluating auxiliary variables during conflict analysis. A critical aspect of LLG's conflict analysis algorithm is verifying whether a newly learned constraint is asserting at any earlier decision level. The learned constraint may, however, contain auxiliary variables that did not yet exist at a previous decision level. Even though these auxiliary variables are properly propagated once a backtrack is executed, they might not be propagated on the trail when conflict analysis considers them. This can result in incorrectly classifying the learned constraint as non-asserting.

To resolve this, the truth-value of an auxiliary variable at a particular decision level is always computed based on its definition, rather than looked up as for other variables. This allows us to infer the truth value of the auxiliary variable, regardless of whether it has been propagated.

4.4 Examples of explanations

We have formulated linear explanations for several arithmetic constraints and the element global constraint. This section provides a detailed exposition of some of these explanations. A comprehensive overview of all explanations is presented in Appendix A. These explanations can closely resemble a lazy decomposition, where linear inequalities are added to the model lazily. However, when we refer to inequalities as explanations, we explicitly mean that we do not add the inequalities to the model.

▶ **Example 7** (Explanation of $x_i \neq val$ for $Ax \neq b$)**.** The propagation of $x_i \neq val$ implies that either $\langle x_i < val \rangle$ or $\langle x_i > val \rangle$ must hold. This can be captured by introducing an auxiliary variable p defined by $p \iff Ax < b$, leading to two possible explanations: (1) $Ax < b + M(1 - p)$, (2) $Ax > b - Mp$.

Which explanation is chosen depends on whether the propagation of $x_i \neq val$ decreases the upper bound of x_i (explanation 1), increases its lower bound (explanation 2), or introduces a hole in its domain (either explanation is valid, in our implementation we pick explanation 1).

▶ **Example 8** (Explanation of $\langle b \geq 1\rangle \wedge \langle c \geq 0\rangle \rightarrow a \leq \lfloor ub(c)/lb(b)\rfloor$ for $a \times b = c$)**.** To express this propagation linearly, we fix b to its *current* lower bound. Assuming a lower bound of $b_{\min} = 5$, the explanation is given by $b \geq 5 \wedge c \geq 0 \rightarrow 5a \leq c$. We introduce auxiliary variables (1) $p_1 \iff b \geq 5$ and (2) $p_2 \iff c \geq 0$ to represent the conditions. This leads to the linear explanation:

$$5a \leq c + M(1 - p_1) + M(1 - p_2). \tag{2}$$

▶ **Example 9** (Explanation of $\neg c \rightarrow r \leq 0$ for $r \rightarrow c$)**.** A noteworthy constraint is the *half-reified constraint* [15], which defines the implication $r \rightarrow c$ for Boolean variable r and an arbitrary CP constraint c. This constraint ensures that c must hold whenever $r = 1$. If c is conflicting under the current assignment, we can propagate $\langle r \leq 0\rangle$. Let $Ax \leq b$ be the explanation from c for the conflict. Consequently, we incorporate r into the explanation to capture the relationship between r and the conflict:

$$Ax \leq b + M(1 - r). \tag{3}$$

5 Experiments

We implemented our LLG approach in Pumpkin [11]. The initial version of Pumpkin serves as the baseline LCG solver. In the experiments we use the number of conflicts as the main metric to allow us to draw conclusions that are independent of the runtime and the efficiency of the implementation. We believe this fairly shows the potential of our LLG approach. To further ensure that the results are due to the differences in the conflict analysis procedure, and not other factors, we only consider instances for which the branching strategies of LLG and LCG are fixed according to the provided strategy in the instances. We summarize the results:

1. Subsection 5.2: In 25% of the instances *where at least one linear constraint is learned* the number of conflicts is reduced by at least 60%. The median reduction is around 10%, and in the worst 25% of instances there is a slight increase in the number of conflicts.
2. Subsection 5.3: Learned inequalities indeed provide stronger reasoning than clauses.
3. Subsection 5.4: The success rate of LLG analysis is relatively low and decreases with time.
4. Subsection 5.5: Decomposing to linear constraints, to suit the cutting-planes reasoning as much as possible, encounters more conflicts than LLG when aggregated over the entire benchmark set.
5. Subsection 5.6: The presence of clausal propagation reduces the number of encountered conflicts for LLG, indicating it is useful to remember clauses when linear analysis fails.

5.1 Experimental setup

The instances we used to evaluate our LLG implementation are drawn from the MiniZinc Challenges 2008-2024 [36, 34] and the MiniZinc Benchmarks[1]. Models with unbounded or floating-point domains were excluded, as well as models without a specified search heuristic. Instances from the MiniZinc challenge which are not solved by at least one finite-domain solver are also excluded. For every model with more than 10 instances, we sampled 10 instances randomly.

[1] `https://github.com/MiniZinc/minizinc-benchmarks`

Each instance is decomposed into the following constraints: multiplication ($a \times b = c$), truncating division ($a/b = c$), absolute value ($a = |b|$), maximum ($max(A) = b$), not equals ($Ax \neq b$), linear less-than-or-equal-to ($Ax \leq b$), reified ($p \rightarrow constraint$), element ($A[idx] = b$) and clauses. The explanations for these constraints are given in Appendix A.

This results in a dataset of **952** instances based on **223** unique models. These instances result in roughly **50.5M** linear inequalities, **18.4M** reified constraints, **3.7M** not-equals constraints, **1.3M** multiplication constraints, **654k** element constraints, **417k** maximum constraints, only **4.5k** absolute and only **1.4k** division constraints.

All experiments were conducted on the DelftBlue [10] compute cluster. Each run used a single thread of an Intel Xeon E5-6248R 24C 3.0GHz and had access to 8GB of memory. The time limit was set to 1 hour per instance.

5.2 Reduction of conflicts

The first experiment shows how LLG impacts the number of conflicts encountered compared to the LCG baseline. We can break down the instances into three categories:

1. There are 328 instances that timed out or ran out of memory for LLG. This is compared to 289 in the LCG solver. That is an increase, however, the instances for which this happens are different between the two solvers. I.e., the LLG solver times out or errors on different instances than the LCG solver. In many of these instances, the LLG solver does learn linear constraints, but their impact is not big enough to prevent a time-out.
2. Then, there are 181 instances that are solved by the LLG solver, but not a single linear inequality was learned. In these instances, even though LLG has the overhead of cutting-planes reasoning, that overhead did not prevent the solver from terminating within the time-out.
3. The final 443 instances are solved within the time limit, and at least one linear inequality is learned. Figure 1 aggregates the ratio of the number of conflicts for LLG relative to LCG. The lower half of the distribution exhibits significant reductions, with a decrease in conflicts of up to **60%** for a quarter of the instances, even increasing to over **90%** when looking at the top 10% of instances. Then, the Q2 to Q3 quartile indicates that in roughly a quarter of the completed instances, the number of conflicts is reduced only slightly. For these instances, LLG learns very few linear constraints, or learns linear constraints that propagate similar bounds as learned clauses. Lastly, the upper 75% to 90% percentile range highlights cases where the number of conflicts increases marginally. This is due to some learned constraints being weaker compared to the clauses that would have been learned by the LCG solver.

These numbers show that, with the current state of LLG, not all instances benefit from its linear conflict analysis. However, when a linear inequality is derived, it has the potential to massively reduce the number of conflicts. We initially assumed that the top-performing instances would be derived from a limited subset of models that fit LLG especially well. However, upon examining the instances exhibiting at least a 50% reduction in conflicts, we identify 117 instances originating from 52 models ($117/52 \approx$ **2.3** instances per model). In comparison, a total of 574 instances from 171 models successfully complete for both LCG and LLG ($547/165 \approx$ **3.3** instances per model): the top-performing instances are even more varied than the full dataset. Consequently, our assumption was wrong; particular models are not more suitable for LLG than others.

The distribution of constraints across these instances also seems highly variable. The successful instances exhibit a slightly higher percentage of linear inequality constraints, though the difference is within only a few percent.

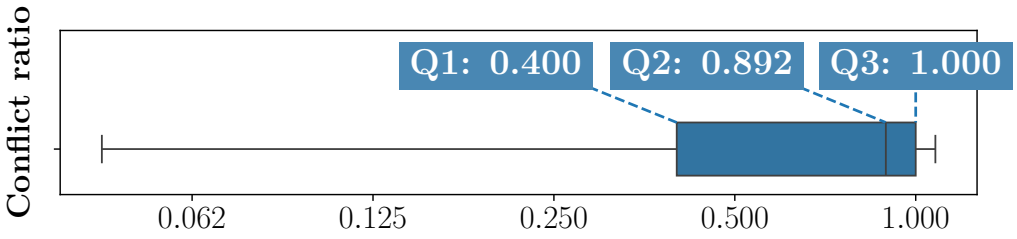


Figure 1 Boxplot showing the ratio of conflicts in LLG over LCG (lower = better). The boxplot displays the Q1, Q2 and Q3 quartiles, with whiskers extending to the 10% and 90% percentiles. A logarithmic scale is used for equal spacing of ratios. Only instances where both LCG and LLG successfully completed, and at least one linear constraint was learned, are considered.

Lastly, although the implementation of the LLG solver can be improved on many fronts, we observe that for the top 25% of instances – each achieving at least a 60% reduction in conflicts – runtime performance already surpasses our baseline LCG, owing to the significant decrease in conflicts of LLG. Specifically, for this subset of instances (excluding those with execution times below 5 seconds), the median runtime improvement is 75%.

5.3 Strength of learned inequalities

The motivation for LLG is based on the proposition that linear constraints may provide stronger reasoning than clauses. With this experiment, we show that the theoretical benefit translates to practice. For the 443 completed instances with at least one learned inequality, we examine whether the learned inequality indeed propagates more often than the clause that would have been learned in its place.

The results indicate that when a linear constraint is learned, it replicates over **91%** of the propagations that the fallback clause would have produced. In contrast, only **37%** of the propagations triggered by learned linear constraints would have been replicated by the clause. This confirms that, in nearly all cases, the linear constraint is at least as strong as the fallback clause. This further suggests that neither conflict analysis method strictly dominates the other, yet learned linear constraints tend to be more general in practice.

We also observe that in a normal LLG execution, the majority of learned clauses do not propagate more than once. In contrast, the median number of propagations per learned learned inequality is **90**. This demonstrates that when an inequality can be learned, it generally has a much greater impact on the search compared to a learned clause.

5.4 Analysis success rate

This experiment aims to illustrate the success rate of LLG analyses as the search progresses, and to identify reasons for failed analyses. Recall that a successful LLG analysis results in learning a linear constraint, whereas a failed analysis reverts to LCG for reasons such as a violated conflicting invariant. Here, it is relevant to compare successful instances with less successful instances. We divide the 624 instances that the LLG solver could finish into three subsets: all instances (Figure 2a), those with at least a 50% reduction in conflicts (Figure 2b), and those with at least a 75% reduction in conflicts (Figure 2c).

Firstly, Figure 2a demonstrates a declining trend in successful analyses over time, accompanied by an increase in conflicts and explanations that cannot be expressed linearly. This trend persists when results are categorized by the number of conflicts (comparing small and

large instances) and by problem type (satisfaction versus optimization). The only correlation we can observe is the one between a decreasing LLG success rate and an increase in failed LLG analyses attributed to encountered clauses. Approximately **38%** of LLG failures can be attributed to these factors. The remaining **62%** consist of failures that occur at a relatively constant rate throughout the search, including invariant violations ($\approx 50\%$), combinations that fully cancel out ($\approx 6\%$), overflows ($\approx 3\%$), and cases where a decision is reached before identifying an asserting constraint ($\approx 3\%$).

In contrast, Figures 2b and 2c exhibit some differences compared to the full dataset. Initially, conflict analysis is highly successful – significantly more so than for the full dataset. However, as the search progresses, analysis performance declines sharply. For these instances, we can see that the number of analysis failures due to invariant violations increases substantially throughout the search. Figures 2a through 2c all illustrate that LLG conflict analysis succeeds in only a relatively small fraction of instances. This experiment demonstrates that even a relatively small number of learned linear constraints can significantly decrease the number of conflicts encountered.

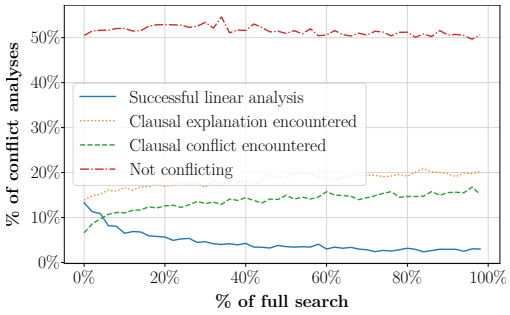
Given the decreasing frequency of conflict analyses leading to newly learned linear constraints, it is worth exploring whether a similar pattern is observed in the propagation of learned constraints. Figure 2d plots the density function (area under the curve equals 1) of all propagations triggered by learned constraints, presented for the same three subsets of instances. The results indicate that, despite the decline in newly learned linear constraints, propagations of learned constraints increase over time. This indicates that, as the search progresses, previously learned linear constraints continue to propagate consistently.

Explanation slacks. Finally, we examined the slack of explanations that resulted in a learned inequality compared to those that did not. In the context of LLG, slack is defined as $\textsc{slack}(Ax \leq b) = b - \textsc{lb}(Ax)$, where LB computes the current lower bound. Notably, a negative slack indicates a conflict.

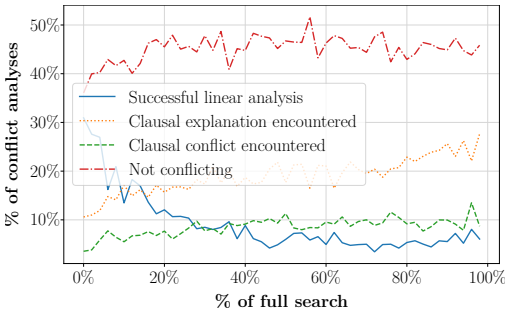
First, we found no clear correlation between the slack of a conflict explanation and the proportion of these explanations that eventually led to a learned constraint, contrary to our expectation that large negative slacks are more likely to succeed. The only necessary condition for a conflict explanation is that is has negative slack. Second, we observed a strong correlation between the slack of a propagation explanation and its likelihood of resulting in a learned constraint: explanations with lower slack demonstrate a higher probability of leading to a learned constraint. This result is sensible, as lower slack values indicate that the explanation is closer to being either asserting or conflicting. In contrast, explanations with high slack may be overly general or involve large big-M coefficients. These results suggest the possibility of selecting certain explanations based on their slack, prioritizing those that are more likely to contribute to the derivation of a learned constraint.

5.5 LLG compared to linear decomposition

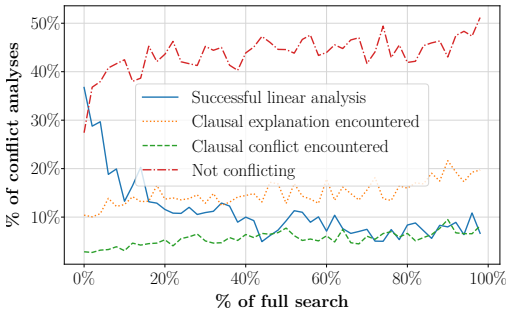
In addition to comparing LLG with LCG, it is also valuable to evaluate Pumpkin's decomposition as described in Section 5.1 against a decomposition consisting only of linear inequalities. This evaluation shows that using CP propagators – even though not all its linear explanations are equally successful – is still beneficial over a fully linear model. For this experiment, all instances from the test set were reformulated into a fully linear model using the MiniZinc Linear Library [4]. Among the 952 instances in the test set, 938 could be transformed into a linear model within a 300-second timeout. The 14 instances that could not be converted likely experienced excessive model growth when represented fully linearly.



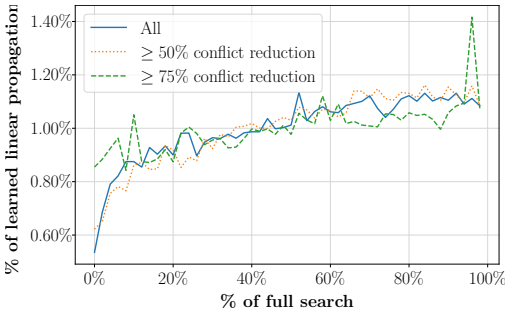
(a) All instances.



(b) Instances with at least 50% conflict reduction over LCG.



(c) Instances with at least 75% conflict reduction over LCG.

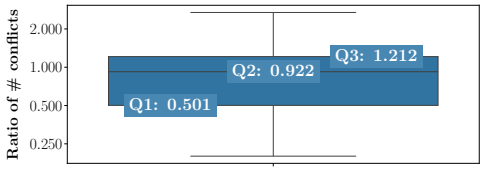


(d) Probability density function showing the distribution of propagations of learned inequalities (y-axis) as a function of search progress, measured by the percentage of total conflicts (x-axis).

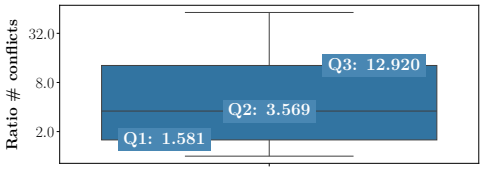
Figure 2 Figures (a) through (c) show the success rate of conflict analyses (y-axis), plotted against the percentage of total conflicts (x-axis). The four most significant outcomes are included. Figure (a) demonstrates a decrease in analysis success as more clauses are encountered. Figures (b) and (c) show a strong initial success, followed by an increase in invariant violations. Figure (d) demonstrates that, although the number of newly learned linear constraints decreases as the search progresses, the tail end exhibits the highest concentration of constraint propagations. This can be explained by the cumulative effect of both newly introduced and previously learned constraints.

For the 938 successfully converted instances, Figure 3a presents the ratio of conflicts between the linear decomposition (baseline) and LLG. The results indicate a substantial reduction in conflicts when using LLG. Specifically, at least 25% of the instances exhibit a conflict reduction of **50%** or more. Furthermore, the first and second quartiles (Q1 to Q2) of the boxplot show a notable decline in conflicts, ranging from **50%** to **8%**. The second to third quartiles (Q2 to Q3) display a mix of reductions, from an **8%** reduction to an increase of **20%**. The remaining instances exhibit a slightly larger increase in conflicts.

These increases in conflicts can be attributed to two primary factors. Firstly, the preprocessing optimizations performed by the linear decomposition can result in the generation of smaller linear problem instances compared to those produced by Pumpkin's decomposition. Secondly, some of Pumpkin's propagators perform propagations that cannot be easily linearly explained, such as *set* or *alldifferent* propagations, whereas the linear decomposition provides alternative propagations that can be linearly explained more effectively. Nevertheless, LLG continues to exhibit a significant performance advantage over the linear decomposition.



(a) Plot showing the conflict ratio between the linear decomposition (baseline) and Pumpkin's decomposition (lower = better). Pumpkin's decomposition encounters significantly fewer conflicts in over 50% of the instances. In 25% of instances, the linear decomposition performs either slightly worse or slightly better. In the remaining 25% of instances, the linear decomposition performs better than Pumpkin's decomposition.



(b) Evaluation of a variation of LLG where learned clauses are propagated but not added to the model. The results show a notable increase in conflicts (note the y-axis labels), highlighting the importance of clausal propagation for LLG.

Figure 3 Boxplots showing the ratio of conflicts in LLG relative to a variation on LLG. The boxplots display the Q1, Q2 and Q3 quartiles, with whiskers extending to the 10% and 90% percentiles. A logarithmic scale is used for equal spacing of ratios. Only instances where both LCG and the variation successfully completed are considered.

5.6 Forgetting learned clauses

Subsection 5.4 shows that as more clauses are encountered in the conflict analysis, the success-rate of the the linear conflict analysis drops. Therefore, we investigate the consequences of only propagating a learned clause once, without storing it in the constraint database. The results, presented in Figure 3b, indicate that omitting clause storage leads to a significant increase in conflicts. This observation underscores that even though encountering clauses likely results in learning fewer linear inequalities, the effectiveness of LLG relies heavily on the ability to store and propagate learned clauses.

We note the clear parallel between this experiment and the standard behavior of IntSat [27]. However, two key distinctions differentiating the two approaches prevent an accurate comparison. First, IntSat operates solely on fully linear models, which inherently provide better linear explanations for propagations than a CP model can, increasing the success rate of IntSat's conflict analysis. Second, IntSat tries to transform learned clauses into linear inequalities when at most one bound of the clause is non-binary. Within our CP problems, such conversions are rarely feasible due to the predominant use of integer variables. This limitation might change if the CP models were reformulated as linear programs.

6 Conclusion & Future Work

The effectiveness of constraint learning in Constraint Programming has been extensively studied, with much of the focus on learning clauses. However, linear constraints have the potential to provide more powerful reasoning capabilities. In response, we propose Lazy Linear Generation (LLG), a conflict analysis algorithm that incorporates concepts from CDCL [22], IntSat [27], HaifaCSP [40], and Lazy Clause Generation [38] to develop a novel learning mechanism for linear constraints. LLG generates explanations for propagations and conflicts for arbitrary CP propagators. To achieve this, the algorithm introduces new auxiliary variables while maintaining a consistent solver state.

Our experimental analysis of LLG shows that, on the one hand, learning linear constraints can lead to a substantial reduction in the number of conflicts over solely learning clauses. Although the success rate of linear conflict analysis is relatively low, when a linear constraint

is derived, the impact of that constraint on overall solver performance is significant. On the other hand, there are many instances where LLG fails to derive any new linear constraint, as the explanations end up being overly general. When compared to a linear decomposition with the same conflict analysis, we observe a reduction in the number of conflicts encountered when aggregating over all instances, suggesting that the strong CP propagation is useful and decomposing to linear constraints is not the preferred approach. Furthermore, our experiments highlight that clausal propagation is still important, even when using conflict analysis based on cutting-planes. To conclude, whilst there are still improvements to be obtained, we believe our approach shows potential to improve the constraint learning capabilities of CP solvers.

There are several promising avenues for further research of the LLG algorithm. First, it would be interesting to consider the impact on more global constraints. Second, it should be investigated whether we can increase the success rate of the cutting-planes conflict analysis. Additionally, we could improve our linear explanations based on decomposition optimisations [4]. Finally, the branching heuristic can likely utilize information from cutting-planes conflict analysis to make the search more dynamic. We believe these research directions offer valuable opportunities to further enhance the effectiveness and applicability of solvers.

References

1 Younes Aalian, Gilles Pesant, and Michel Gamache. Optimization of Short-Term Underground Mine Planning Using Constraint Programming. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2023.6.

2 Tobias Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, March 2007. MAG ID: 1990267895 S2ID: 2446332bd4193d61129622041298218e0f1c7676. doi:10.1016/j.disopt.2006.10.006.

3 Roberto Amadini, Mak Andrlon, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Constraint Programming for Dynamic Symbolic Execution of JavaScript. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 1–19, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-19212-9_1.

4 Gleb Belov, Peter J. Stuckey, Guido Tack, and Mark Wallace. Improved linearization of constraint programming models. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 49–65, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-44953-1_4.

5 A. L. Brearley, G. Mitra, and H. P. Williams. Analysis of mathematical programming problems prior to applying the simplex algorithm. *Mathematical Programming*, 8(1):54–83, December 1975. doi:10.1007/BF01580428.

6 Martin Bromberger, Thomas Sturm, and Christoph Weidenbach. A complete and terminating approach to linear integer solving. *Journal of Symbolic Computation*, 100:102–136, September 2020. MAG ID: 2966563153 S2ID: ded68accef3d475b5e17fa9c1ae9bdfb6bd5968c. doi:10.1016/j.jsc.2019.07.021.

7 Geoffrey Chu and Peter J. Stuckey. Structure Based Extended Resolution for Constraint Programming, June 2013. doi:10.48550/arXiv.1306.4418.

8 Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed: The chuffed cp solver. URL: https://github.com/chuffed/chuffed.

9 Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, January 1990. doi:10.1016/0004-3702(90)90046-3.

10 Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2, 2024.

11 Emir Demirović, Maarten Flippo, Imko Marijnissen, Konstantin Sidorov, and Smits Jeff. Pumpkin: A lazy clause generation constraint solver in rust, 2024. URL: https://github.com/ConSol-Lab/Pumpkin.

12 Donald Chai and Andreas Kuehlmann. A fast pseudo-Boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):305–317, February 2005. MAG ID: 1985602827 S2ID: 6ac45677642ce52faf49a15dc3f1cf3ffdaf504d. doi:10.1109/tcad.2004.842808.

13 Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining alldifferent. In *Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122*, ACSC '12, pages 115–124, AUS, January 2012. Australian Computer Society, Inc. URL: http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV122Downing.html.

14 Thibaut Feydy, Andreas Schutt, and Peter Stuckey. Semantic Learning for Lazy Clause Generation. In *TRICS workshop, held alongside CP*. Citeseer, 2013.

15 Thibaut Feydy, Zoltan Somogyi, and Peter J. Stuckey. Half Reification and Flattening. In Jimmy Lee, editor, *Principles and Practice of Constraint Programming – CP 2011*, volume 6876, pages 286–301. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-23786-7_23.

16 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024. URL: https://www.gurobi.com.

17 Jan Elffers and Jakob Nordström. Divide and Conquer: Towards Faster Pseudo-Boolean Solving. *International Joint Conference on Artificial Intelligence*, pages 1291–1299, July 2018. MAG ID: 2808706008 S2ID: fe4efc3fab6a3a5b01a58f9434bd86c1587fe1d3. doi:10.24963/ijcai.2018/180.

18 Dejan Jovanović and Leonardo de Moura. Cutting to the Chase. *Journal of Automated Reasoning*, 51(1):79–108, June 2013. doi:10.1007/s10817-013-9281-x.

19 Dejan Jovanović and Leonardo de Moura. Cutting to the Chase Solving Linear Integer Arithmetic. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 338–353, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

20 George Katsirelos. Generalized NoGoods in CSPs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 5, pages 390–396, 2005.

21 George Katsirelos. *Nogood Processing in CSPs*. PhD thesis, University of Toronto, January 2009.

22 João Marques-Silva and Karem A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. MAG ID: 2044560939 S2ID: 38be6e613f2c30d21bffe8b468bc0cd46edba0d0. doi:10.1109/12.769433.

23 Matthew McIlree and Ciaran McCreesh. Certifying Bounds Propagation for Integer Multiplication Constraints. In *39th Annual AAAI Conference on Artificial Intelligence (AAAI'25)*, Philadelphia, 2025.

24 Neil C. A. Moore. *Improving the Efficiency of Learning CSP Solvers*. Thesis, University of St Andrews, May 2011.

25 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-74970-7_38.

26 Robert Nieuwenhuis. The IntSat Method for Integer Linear Programming. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, volume 8656, pages 574–589. Springer International Publishing, Cham, 2014. Series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-319-10428-7_42.

27 Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. IntSat: integer linear programming by conflict-driven constraint learning. *Optim. Methods Softw.*, pages 1–28, September 2023. ARXIV_ID: 2402.15522 MAG ID: 4387099028 S2ID: 3843288a76ab7ad11e0d0f664b35c5fd885acc94. doi:10.1080/10556788.2023.2246167.

28 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints - An International Journal*, 14(3):357–391, September 2009. MAG ID: 2059035667 S2ID: 1e00201f51a2f2161c8d7ae0a7843774955659cf. doi:10.1007/s10601-008-9064-x.

29 Peter J. Stuckey Olga Ohrimenko and Michael Codish. Propagation = lazy clause generation. *International Conference on Principles and Practice of Constraint Programming*, 4741:544–558, September 2007. MAG ID: 1546943373 S2ID: 73d784cd29c977ee83874229bcf29a0b143922bb. doi:10.1007/978-3-540-74970-7_39.

30 Laurent Perron and Frédéric Didier. Cp-sat. URL: https://developers.google.com/optimization/cp/cp_solver/.

31 Olivier Ponsini, Claude Michel, and Michel Rueher. Verifying floating-point programs with constraint programming and abstract interpretation techniques. *Automated Software Engineering*, 23(2):191–217, June 2016. doi:10.1007/s10515-014-0154-2.

32 Stéphanie Roussel, Thomas Polacsek, and Anouck Chan. Assembly Line Preliminary Design Optimization for an Aircraft. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2023.32.

33 Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Explaining the cumulative propagator. *Constraints*, 16(3):250–282, July 2011. doi:10.1007/s10601-010-9103-2.

34 Peter J. Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, July 2010. doi:10.1007/s10601-010-9093-0.

35 Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. MiniZinc Challenge 2024 results. URL: https://www.minizinc.org/challenge/2024/results/.

36 Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The MiniZinc Challenge 2008–2013. *AI Magazine*, 35(2):55–60, June 2014. doi:10.1609/aimag.v35i2.2539.

37 Pierre Talbot, Tingting Hu, and Nicolas Navet. Constraint Programming with External Worst-Case Traversal Time Analysis. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2023.34.

38 Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. *International Conference on Principles and Practice of Constraint Programming*, 5732:352–366, September 2009. MAG ID: 2114231362 S2ID: bc1b44d4e041280edc0e26fd55c630e69f4e8163. doi:10.1007/978-3-642-04244-7_29.

39 P. Van Beek, Francesca Rossi, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

40 Michael Veksler and Ofer Strichman. Learning general constraints in CSP. *Artificial Intelligence*, 238:135–153, September 2016. doi:10.1016/j.artint.2016.06.002.

41 Sameela Suharshani Wijesundara, Maria Garcia de la Banda, and Guido Tack. Addressing Problem Drift in UNHCR Fund Allocation. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:18, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2023.37.

A List of explanations

Table 1 contains a comprehensive overview of all explanations used for LLG. In case several explanations are possible for a propagation, the right one is chosen based on whether a lower or upper bound is propagated. The explanations have undergone multiple manual reviews but have not been formally verified for correctness.

Table 1 Table containing all explanations used for LLG.

Constraint	Propagation / conflict	Conditions	Explanation	Auxiliaries
$Ax \leq b$	$x_i \leq v$	$ub(x_i) > (b - lb(Ax) + lb(x_i))$	$Ax \leq b$	-
	Conflict	$lb(Ax) \geq b$	$Ax \leq b$	
$Ax \neq b$	$x_i \neq v$	x_i is only unfixed var in x	• $Ax \leq b - 1 + M(1-p)$ • $Ax \geq b + 1 - Mp$	$p \iff Ax < b$
	Conflict	$lb(Ax) = ub(Ax) = b$	$Ax \geq b + 1 - Mp$	
$a = \lvert b \rvert$	$a \geq 0$	-	$a \geq 0$	
	$a \geq lb(b)$	$lb(b) > 0$	$a \geq b$	
	$b \geq -ub(a)$	-		
	$a \geq \lvert ub(b) \rvert$	$ub(b) < 0$	$a \geq -b$	• $p_1 \iff b \leq 0$ • $p_2 \iff b \geq 0$
	$b \leq ub(a)$	-		
	$a \leq \max(\lvert lb(b) \rvert, \lvert ub(b) \rvert)$	-		
	$b \leq -lb(a)$	$ub(b) \leq 0$	• $a + b \leq M(1-p_1)$ • $a - b \leq M(1-p_2)$	
	$b \geq lb(a)$	$lb(b) \geq 0$		
$max(A) = b$	$a_i \leq ub(b)$	-	$a_i \leq b$	
	$b \geq \max_{a_i \in A}(lb(a_i))$	-		• $\forall i \in \lvert A \rvert$: auxiliary p_i • $\sum p_i = 1$
	$b \leq \max_{a_i \in A}(ub(a_i))$	-		
	$a_i \geq lb(b)$	a_i only for $ub(a_i) \geq lb(b)$	$rhs \leq a_i + M(1-p_i)$	
	$b \leq ub(a_i)$	$p_i = 1$		
	$a_i \geq lb(b)$	$p_i = 1$		
$a \times b = c$	Sign propagations, for instance: $c \geq 0$	$lb(a) \geq 0 \wedge lb(b) \geq 0$	$c \geq 0 - Mn_a - Mn_b$	• $n_a \iff a < 0$ • $n_b \iff b < 0$ • $n_c \iff c < 0$ • $b_{lb} \iff b \geq lb(b)$ • $b_{ub} \iff b \leq ub(b)$
	a/b upper bound propagations, for instance: $a \leq \lfloor ub(c)/lb(b) \rfloor$	$lb(b) \geq 1 \wedge lb(c) \geq 0 \wedge ub(c) \geq 1$	$lb(b) \cdot a \leq c + M(1 - b_{lb}) + Mn_c$	
	$c \geq lb(a) \times lb(b)$	$lb(a) \geq 0 \wedge lb(b) \geq 0$		
	a/b lower bound propagations, for instance: $a \geq \lceil lb(c)/ub(b) \rceil$	$lb(b) \geq 0 \wedge ub(b) \geq 1 \wedge lb(c) \geq 1$	$ub(b) \cdot a \geq c + M(1 - b_{ub}) - Mn_b - Mn_c$	
	$c \leq ub(a) \times ub(b)$	$lb(a) \geq 0 \wedge lb(b) \geq 0$		
$num \mathbin{//} den = rhs$	Sign propagations, for instance: $rhs \geq 0$	$lb(num) \geq 0 \wedge lb(den) \geq 0$	$rhs \geq 0 - Mn_{num} - Mn_{den}$	• $n_{num} \iff num < 0$ • $n_{den} \iff den < 0$ • $p_{den} \iff den > 0$ • $rhs_{lb} \iff rhs \geq lb(rhs)$ • $den_{ub} \iff den \leq ub(den)$
	$rhs \leq ub(num)/lb(den)$	$ub(num) \geq 0 \wedge ub(den) \geq 0$		
	$den \leq ub(num)/lb(rhs)$	$lb(rhs) \geq 0 \wedge ub(num) \geq 0$	$lb(rhs) \cdot den \leq num + M(1 - rhs_{lb}) + Mn_{num}$ Swap rhs and den for alternative	
	$num \geq lb(rhs) \cdot lb(den)$	$lb(rhs) \geq 0 \wedge ub(den) \geq 0$		
	$rhs \geq \lfloor lb(num)/ub(den) \rfloor$	$lb(num) \geq 0 \wedge ub(den) \geq 0$	$ub(den) \cdot rhs + den - 1 \geq num - M(1 - den_{ub}) - Mn_{num} - M(1 - p_{den})$	
	$num \leq (ub(rhs) + 1) \cdot ub(den) - 1$	$ub(den) \geq 0 \wedge ub(rhs) \geq 0$	Swap rhs and den for alternative	
$A[idx] = b$	$b \leq \max_{a_i \in A if idx}(ub(a_i))$	-	$b \leq a_i + M(1-p_i)$	• $\forall i \in \lvert A \rvert$: auxiliary p_i • $\sum p_i = 1$ • $\sum i \cdot p_i = idx$
	$a_i \leq ub(b)$	$idx = i$		
	$b \geq \min_{a_i \in A if idx}(lb(a_i))$	-	$b \geq a_i - M(1-p_i)$	
	$a_i \geq lb(b)$	$idx = i$		
	$p_i \geq 1$	$idx = i$	$\sum i \cdot p_i = idx$	
	$p_i \leq 0$	$idx \neq i$		
$p \rightarrow cond$	$p \leq 0$	$\neg cond$	$expl(cond) \rightarrow Ax \leq b$	-
	Conflict	$p \geq 1 \wedge \neg cond$	$Ax \leq b + M(1-p)$	

Transition Dominance in Domain-Independent Dynamic Programming

J. Christopher Beck
Department of Mechanical and Industrial Engineering, University of Toronto, Canada

Ryo Kuroiwa
National Institute of Informatics, Tokyo, Japan
The Graduate School of Advanced Studies, SOKENDAI, Tokyo, Japan

Jimmy H. M. Lee
Department of Computer Science and Engineering,
The Chinese University of Hong Kong, Hong Kong

Peter J. Stuckey
Department of Data Science and AI, Monash University, Clayton, Australia
ARC Training Centre OPTIMA, Melbourne, Australia

Allen Z. Zhong
Department of Data Science and AI, Monash University, Clayton, Australia
ARC Training Centre OPTIMA, Melbourne, Australia

Abstract

Domain-independent dynamic programming (DIDP) is a model-based paradigm for dynamic programming (DP) that enables users to define DP models based on a state transition system. Heuristic search-based solvers have demonstrated strong performance in solving combinatorial optimization problems. In this paper, we formally define *transition dominance* in DIDP, where one transition consistently leads to better solutions than another, allowing the search process to safely ignore dominated transitions. To facilitate the efficient use of transition dominance, we introduce an interface for defining transition dominance and propose the use of *state functions* to cache values, thereby avoiding redundant computations when verifying transition dominance. Experimental results on DP models across multiple problem classes indicate that incorporating transition dominance and state functions yields a 5 to 10 times speed-up on average for different search algorithms within the DIDP framework compared to the baseline.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization

Keywords and phrases Dominance, Dynamic Programming, Combinatorial Optimization

Digital Object Identifier 10.4230/LIPIcs.CP.2025.5

Supplementary Material *Software*: `https://github.com/domain-independent-dp/didp-rs/releases/tag/transition-dominance-cp25`
archived at `swh:1:dir:ec877b961363d47d3f015af89a623b7a571bc736`

Funding This work was supported by the Australian Research Council OPTIMA ITTC IC200100009, a General Research Fund (CUHK14206321) by the Hong Kong University Grants Committee, a Direct Grant by CUHK, and the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

Dynamic programming [2] is a classical approach to solving combinatorial optimization problems. While it is often the most efficient way of solving many problems, traditionally dynamic programming solutions are hard coded for each problem of interest. *Domain-Independent Dynamic Programming* (DIDP) [13] is a new paradigm to allow the statement of the Bellman equations defining a combinatorial opmisation problem, independent of the

© J. Christopher Beck, Ryo Kuroiwa, Jimmy H. M. Lee, Peter J. Stuckey, and Allen Z. Zhong;
licensed under Creative Commons License CC-BY 4.0

techniques used to solve these equations, separating the specification of the model from the solving, as in other complete solving approaches such as constraint programming (CP) and mixed integer programming (MIP). This separation enables easy experimentation with different search algorithms for the same problem domain and enables research on different DP models for a given problem. Empirical evaluation demonstrates the framework outperforms CP and MIP on multiple problem classes, closing a number of open problem instances [14].

The *Dynamic Programming Description Language* (DyPDL) [13] is a powerful modelling language for DIDP that facilitates the expression of dynamic programming (DP) models. Beyond the problem components, DyPDL also supports the modelling of redundant information, such as state dominance and dual bounds, to improve the performance of various solving approaches. In this paper, we extend this framework by formally defining *transition dominance*, redundant information that indicates that certain transitions can be safely ignored if a state satisfies a certain condition. By incorporating transition dominance into DIDP, we make these advanced techniques accessible in a solver-independent manner. Deriving dominances in different problem domains is a challenging task that requires a deep understanding of problem structures. Our case studies demonstrate that the insights of identifying new dominances can be applied to similar problem domains, enabling the discovery of transition dominance from common substructures. Our key contributions are as follows:

1. We formally define the concept of *transition dominance* for dynamic programming models.
2. We introduce a new component into DyPDL to allow users to model transition dominance effectively and *state functions* to avoid redundant computations.
3. We present new transition dominances for several combinatorial optimization problems.
4. We demonstrate that using transition dominance and state functions substantially improve the performance of various search algorithms with 5 to 10 times speed-ups.

2 Background

A DyPDL model is a tuple $\langle \mathcal{V}, S^0, \mathcal{T}, \mathcal{B}, \mathcal{C} \rangle$, where $\mathcal{V}$ is a set of *state variables*, S^0 is the *target state*, $\mathcal{T}$ is a set of *transitions*, $\mathcal{B}$ is a set of *base cases*, and $\mathcal{C}$ is a set of *state constraints*. A state variable can be an *element*, *set*, or *numeric variable*. A state $S \in \mathcal{D}$ is a tuple that assigns values to state variables in $\mathcal{V}$, where the value of a variable $v \in \mathcal{V}$ is denoted by $S[v]$.

Expressions are used in transitions, base cases, and state constraints to describe the computation of a value using state variables and constants. When an expression e is evaluated given a state S, it returns a value $e(S)$. A *numeric expression* returns a numeric value $e(S) \in \mathbb{Q}$. It can refer to a numeric constant or variable, use arithmetic operations, and take the cardinality of a set expression. An *element expression* returns a nonnegative integer $e(S) \in \mathbb{Z}_0^+$, while a *set expression* returns a set $e(S) \in 2^{\mathbb{Z}_0^+}$. A condition *cond* is a function mapping a state S to a Boolean value $c \in \{\top, \bot\}$. We say $S \models \mathit{cond}$ if $c = \top$, and $S \models C$ for a set of conditions C if $S \models \mathit{cond}$ for all $\mathit{cond} \in C$. A condition can refer to a Boolean constant, compare two elements or numeric expressions, or check whether an element is included in a set. The conjunction and disjunction of two conditions are also conditions. Base cases in $\mathcal{B}$ and state constraints in $\mathcal{C}$ are conditions. When a base case $B \in \mathcal{B}$ is satisfied by a state, the state is called a base state, and the set of all base states is denoted as $\mathcal{S}_\mathcal{B}$. We assume that a function $\mathsf{base_cost} : \mathcal{S}_\mathcal{B} \mapsto \mathbb{Q}$ is defined, which returns a numeric value $\mathsf{base_cost}(S)$ given a base state S.

A transition $\tau \in \mathcal{T}$ has *effect* $\mathrm{eff}_\tau : \mathcal{D} \mapsto \mathcal{D}$ which maps a state S to another state $S[\![\tau]\!]$, and *cost* $\mathsf{cost}_\tau : \mathbb{R} \cup \{\infty\} \times \mathcal{D} \mapsto \mathbb{R} \cup \{\infty\}$ which maps a real value r and a state S to a value $\mathrm{cost}_\tau(r, S)$. *Preconditions* pre_τ are conditions on state variables, and τ is *applicable* in a state S only if all preconditions are satisfied, denoted by $S \models \mathrm{pre}_\tau$.

Solving a DyPDL model requires finding a sequence of transitions with the optimal cost to transform the target state S^0 into a base state. Let $\sigma = \langle \sigma_1, \ldots, \sigma_m \rangle$ be a sequence of transitions applicable from S, i.e., $S \models \mathrm{pre}_{\sigma_1}$ and $S^i \models \mathrm{pre}_{\sigma_{i+1}}$ where $S^1 = S[\![\sigma_1]\!]$ and $S^{i+1} = S^i[\![\sigma_{i+1}]\!]$. Then, σ is an S-solution if S^m is a base state, S and each S^i with $i = 1, ..., m$ satisfy state constraints, and S and S^i with $i < m$ are not base states. If S is a base state, we assume that an empty sequence $\langle\rangle$ is an S-solution. The cost of an S-solution σ is defined recursively as

- if $\sigma = \langle\rangle$, $\mathsf{solution_cost}(\sigma, S) = \mathsf{base_cost}(S)$
- otherwise, $\mathsf{solution_cost}(\sigma, S) = \mathsf{cost}_{\sigma_1}(\mathsf{solution_cost}(\langle \sigma_2, ..., \sigma_m \rangle, S[\![\sigma_1]\!]), S)$

An *optimal solution* for minimization is an S-solution whose cost is less than or equal to the cost of any S-solution. Let V be a function of a state S that returns ∞ if there are no S-solution or the cost of an optimal S-solution otherwise.

In this paper, we assume that a DyPDL model is both *finite* and *acylic*, and the cost expression satisfies the Principle of Optimality [2]. Under these assumptions, $V(S)$ can be computed by Bellman equation [15]:

$$V(S) = \mathsf{base_cost}(S), \qquad \text{if } S \in \mathcal{S}_{\mathcal{B}} \tag{1a}$$

$$V(S) = \infty, \qquad \text{else if } S \not\models \mathcal{C} \tag{1b}$$

$$V(S) = \min_{\tau \in \mathcal{T}(S)} \mathsf{cost}_\tau(V(S[\![\tau]\!]), S) \qquad \text{else} \tag{1c}$$

State dominance is a type of redundant information useful for improving the solving efficiency of DyPDL models.

▶ **Definition 1.** *A state S* dominates *another state S', i.e. $S \preceq S'$, if and only if there exists an S-solution σ for any S'-solution σ' such that $cost_\sigma(S) \leq cost_{\sigma'}(S')$ and $|\sigma| \leq |\sigma'|$.*

In practice, it is challenging to detect and exploit state dominance, and therefore the DyPDL formulation focuses on its approximation defined by *resource state variables* with additional *preference*. A state S is *preferred over* (or approximately dominates) another state S', denoted as $S \preceq_a S'$, iff $S[v] \geq S'[v]$ for resource variables where greater is preferred, $S[v] \leq S'[v]$ for resource variables where less is preferred, and $S[v] = S[v']$ for non-resource variables.

3 Transition Dominance in DIDP

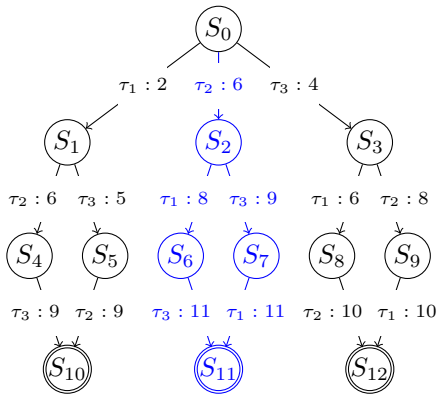
In this paper, we enhance the framework of DIDP by introducing *transition dominance*. State dominance falls into the category of *memory-based* dominance rules [22], where an unexpanded state is compared with previously generated states and pruned if it is shown to be dominated. In contrast, transition dominance is a type of *non-memory-based dominance* that implies the existence of a better solution without requiring additional memory.

▶ **Definition 2.** *Transition dominance at a state S is a relation defined over $\mathcal{T}(S)$ such that τ dominates τ' in S, denoted as $\tau \preceq^S \tau'$, implies that $cost_\tau(V(S[\![\tau]\!]), S) \leq cost_{\tau'}(V(S[\![\tau']\!]), S)$.*

It is straightforward to show that transition dominance is a transitive relation. When a transition dominance relation in a state S is *irreflexive*, i.e., $\tau \not\preceq^S \tau$, we denote the dominance relation by $\prec^S$. If transition dominance relations in all states are irreflexive, removing dominated transitions for all states preserves the optimal cost for a DyPDL model.

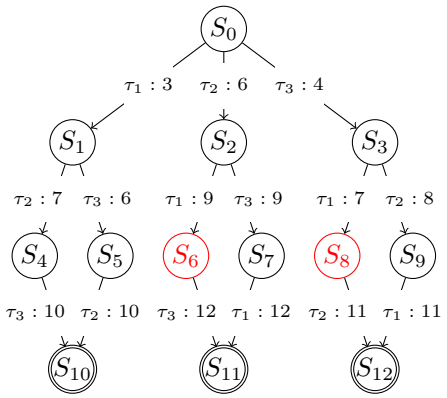
▶ **Proposition 3.** *Let $\mathcal{T}^*(S) = \{\tau' \in \mathcal{T}(S) \mid \nexists \tau \in \mathcal{T}(S), \tau \prec^S \tau'\}$ be the set of non-dominated transitions for a state S where $\prec^S$ is an irreflexive transition dominance relation in a state S. The optimal cost of the DyPDL model $\langle \mathcal{V}, \mathcal{S}^0, \mathcal{T}, \mathcal{B}, \mathcal{C} \rangle$ is equivalent to that of $\langle \mathcal{V}, \mathcal{S}^0, \mathcal{T}^*, \mathcal{B}, \mathcal{C} \rangle$.*

Job	Release(r_i)	Processing(p_i)
1	0	2
2	2	4
3	1	3



(a) Example with transition dominance.

Job	Release(r_i)	Processing(p_i)
1	0	**3**
2	2	4
3	1	3



(b) Example with state dominance.

Figure 1 Example job sequencing instances with transition dominance and state dominance.

The proof, inspired by Chu and Stuckey [4], is in Appendix A.

Note that τ dominates τ' in state S, is different from state dominance $S[\![\tau]\!] \preceq S[\![\tau']\!]$. Transition dominance $\tau \preceq^S \tau'$ holds as long as $\mathsf{cost}_\tau(V(S[\![\tau]\!]), S) \leq \mathsf{cost}_{\tau'}(V(S[\![\tau']\!]), S)$, whereas state dominance between $S[\![\tau]\!]$ and $S[\![\tau']\!]$ requires $V(S[\![\tau]\!]) \leq V(S[\![\tau']\!])$. These two inequalities do not necessarily imply each other, as their relationship depends on the specific cost expressions. In practice, state dominance is approximately determined by comparing the values of state variables between two states, whereas transition dominance depends only on the values of variables within a single state.

▶ **Example 4.** Consider a sequencing problem with three jobs, where each job has a release time and a processing time, and can only be processed after its release time. The objective is to find a sequence of jobs that minimizes the total completion time. We can model this problem using a set variable R to represent the remaining jobs and an integer resource variable t to represent the current time. A transition τ_i for each job i schedules the job next, with the precondition $i \in R$. The optimal value can be computed as follows:

$$V(R,t) = 0, \qquad \text{if } R = \emptyset$$
$$V(R,t) = \min_{i \in R}\{\max(t, r_i) + p_i + V(R \setminus \{i\}, \max(t, r_i) + p_i)\} \qquad \text{else}$$

where $\max(t, r_i) + p_i$ is the completion time for job i given that the current time is t. Figure 1 gives two example instances and their complete state transition graphs. Each edge in the transition graph is labeled with a transition and its corresponding value of $\max(t, r_i) + p_i$. Base states are represented with a double-circle.

Figure 1a illustrates the effect of *transition dominance*, where pruned solutions are highlighted in blue. Transition τ_1 dominates τ_2 in state S_0 because the release time of job 2 is not earlier than the time when job 1 can be fully processed. Thus, directly selecting τ_1 from S_0 is always preferable, leading to the pruning of all S_0-solutions that start with τ_2. Note that transition dominance concerns solutions originating from a single state but differing in their initial transitions.

Figure 1b illustrates the effect of state dominance, where the processing time of job 1 is modified to 3, and so τ_1 no longer dominates τ_2. Note that t is a resource variable where a smaller value is preferred, which defines the state dominance criterion: if a state S has the same set of remaining jobs as another state S', but a smaller or equal current time, i.e., $S[R] = S'[R]$ and $S[t] \leq S'[t]$, then S dominates S'. A close examination reveals that S_4 dominates S_6, S_5 dominates S_8, and S_9 dominates S_7. Assuming that the state graph is explored using depth-first search, proceeding from left to right, solutions passing through S_6 and S_8, highlighted in red, are pruned by state dominance in this example. As shown, pruning by state dominance and by transition dominance operate differently. ⌟

4 Modelling Transition Dominance in DIDP

Theoretically, transition dominance is defined for each state, but transition dominance rules that are valid for a set of states satisfying a certain condition are more interesting in practice. In this section, we introduce two constructs to facilitate modelling transition dominance.

4.1 An Interface for Transition Dominance

Exploiting transition dominance avoids the generation of successor states by dominated transitions. A direct way to specify transition dominance is to add additional preconditions for transitions. Modelers first identify a condition $c_{\tau \preceq \tau'}$ on states such that if a state S satisfies it, then τ dominates τ' in S, i.e., $S \models c_{\tau \preceq \tau'} \rightarrow \tau \preceq^S \tau'$. Then, the negation $\neg(c_{\tau \preceq \tau'} \wedge pre_\tau)$ is added to the preconditions for the dominated transition τ', which states that τ' is applicable only if either transition τ is not applicable or the condition for transition dominance $\tau \preceq^S \tau'$ is not satisfied. Otherwise, τ' can be pruned since τ is a better applicable transition. In DP-based tools with non-declarative APIs like ddo [10] and CODD [20], transition dominance can be implemented by manually checking these preconditions during successor generation.

▶ **Example 5.** Consider the transition dominance in Example 4 again. If there are two jobs i and j such that the time after scheduling i and j consecutively is no greater than that after scheduling j only in any state, then scheduling i first is no worse than scheduling j first. Formally, for any state S, if $precede(i, j) = \max(S[t], r_i) + p_i \leq r_j$ is true, then taking τ_i at state S is no worse than taking τ_j. To fully utilize the power of transition dominance, each transition τ_j should be compared against all other transitions to determine whether there is a transition τ_i which dominates τ_j. Therefore, we need to augment the precondition of τ_j with $\wedge_{i \neq j} \neg\{precede(i, j) \wedge (i \in R)\}$. ⌟

Directly modelling transition dominance through preconditions introduces two key challenges. First, it conflates the redundant information of transition dominance with the actual preconditions of transitions, making it difficult for solvers to treat these distinct model elements differently. Second, multiple conditions for dominance require careful handling of tie-breaking for symmetric transitions that dominate each other in a state.

▶ **Example 6.** Consider again the job sequencing problem in Example 4. If the current time t is greater than the release times for two jobs i and j and $p_i \geq p_j$, then transition τ_i dominates another transition τ_j. This is a special case of Proposition 16. However, while the dominance is valid if the jobs have *equal* processing time, adding preconditions for both jobs may make a satisfiable instance unsatisfiable as both transitions become inapplicable.

Multiple conditions for dominance may form reflexive transition dominance relations in some states, violating the requirement in Proposition 3 to preserve the optimality. To resolve this issue, modelers would need to carefully implement tie-breaking between symmetric transitions, adding unnecessary complexity to the model.

To address these challenges, we introduce a new function in DIDPPy, a Python interface for using DIDP, that allows users to explicitly define transition dominance. The function `model.add_transition` returns a unique identifier for the newly added transition that can later be used to retrieve the transition and define transition dominance relationships. The function `model.add_transition_dominance` is provided for this purpose and takes three arguments: the identifiers of the dominating and dominated transitions, and the condition for transition dominance. The following shows a model for the problem in Example 5.

Listing 1 Sample model for job sequencing with transition dominance.

```
import didppy as dp, itertools
r = [0, 2, 1]
p = [2, 4, 3]
all_jobs = [0, 1, 2]
ids=[]

model = dp.Model()
jobs = model.add_object_type(number=3)
R = model.add_set_var(target=all_jobs, object_type=jobs)
t = model.add_int_resource_var(target=0, less_is_better=True)
model.add_base_case([R.is_empty()])
for i in all_jobs:
    tran = dp.Transition(
        name="schedule␣job{}".format(i),
        cost=(dp.max(t, r[i]) + p[i]) + dp.IntExpr.state_cost(),
        effects=[(R, R.remove(i)), (t, dp.max(t, r[i]) + p[i])],
        preconditions=[R.contains(i)],
    )
    id = model.add_transition(tran)
    ids.append(id)

for i, j in itertools.permutations(all_jobs, 2):
    if i != j:
        model.add_transition_dominance(
            ids[i],  ids[j], [dp.max(t, r[i]) + p[i] <= r[j]]
        )
```

We formalize the entity defined with our interface as a *transition dominance rule*.

▶ **Definition 7.** *A transition dominance rule is a triple* (τ, τ', c) *such that* $S \models c \rightarrow \mathsf{cost}_\tau(V(S[\![\tau]\!]), S) \leq \mathsf{cost}_{\tau'}(V(S[\![\tau']\!]), S)$.

Suppose D *is a set of transition dominance rules. The transition dominance graph* $G(S) = (N, E)$ *for a state* S *is a directed graph where the set of nodes is* $N = \mathcal{T}(S)$ *and a directed edge* $(\tau, \tau') \in E$ *exists if* $\exists(\tau, \tau', c) \in D$ *with* $S \models c$.

A transition dominance rule provides a partial description of transition dominance relations across all states. Using the transition dominance graph $G(S)$, we consider the transitive closure of the binary relation induced by a given set of transition dominance rules. Concretely, the binary relation $\preceq^S$ over $\mathcal{T}(S)$ *derived from* $G(S)$ is defined such that there is a path from τ to τ'.

▶ **Proposition 8.** *The relation $\preceq^S$ derived from a transition dominance graph is a transition dominance relation for S.*

Proof. For a pair of transitions such that $\tau \preceq^S \tau'$, there exists a path $(\tau_1, ..., \tau_m)$ in the transition dominance graph where $\tau = \tau_1$ and $\tau' = \tau_m$. By Definition 7, $\mathsf{cost}_{\tau_i}(V(S[\![\tau_i]\!]), S) \leq \mathsf{cost}_{\tau_{i+1}}(V(S[\![\tau_{i+1}]\!]), S)$ for $i = 1, .., m-1$. Thus, $\mathsf{cost}_{\tau}(V(S[\![\tau]\!]), S) \leq \mathsf{cost}_{\tau_2}(V(S[\![\tau_2]\!]), S) \leq ... \leq \mathsf{cost}_{\tau'}(V(S[\![\tau']\!]), S)$, which satisfy the requirement of a transition dominance relation. ◀

While irreflexivity is required to ensure there exists at least one optimal solution as shown in Proposition 3, a user may define symmetric transition dominance rules, e.g., (τ, τ', c) and (τ', τ, c'), such that there exists a state S where $S \models c \wedge c'$. To enforce irreflexivity in the transition dominance relation derived from the transition dominance graph while pruning as many dominated transitions as possible, we identify all strongly connected components (SCCs) and construct a *contracted transition dominance graph* as follows.

▶ **Definition 9.** *Given the transition dominance graph $G(S)$ for a state S, a directed graph $G'(S)$ is a contracted transition dominance graph if the set of nodes is $\mathcal{T}(S)$, and edges are constructed with the following procedure:*

1. *In each SCC of $G(S)$, select a representative node r, and add edge (r, v) to $G'(S)$ for each node $v \neq r$ in the SCC.*
2. *For each edge (u, v) in $G(S)$ such that u and v are in different SCCs, add an edge (r, r') to $G'(S)$ where r and r' are the representative nodes of the SCCs to which u and v belong.*

▶ **Proposition 10.** *The binary relation $\preceq^S$ derived from a contracted transition dominance graph is an irreflexive transition dominance relation.*

Proof. Let $G(S)$ be the transition dominance graph for a state S, and let $G'(S)$ be the contracted transition dominance graph. We show that if $G'(S)$ has a path from τ to τ', then $G(S)$ also has a path from τ to τ'. Then, by Proposition 8, the binary relation derived from $G'(S)$ is a transition dominance relation. Consider an arbitrary edge (u, v) on a path from τ to τ' in $G'(S)$. We show that $G(S)$ has a path from u to v, and thus, by concatenating such paths, we can find a path from τ to τ' in $G(S)$. If u and v belong to the same SCC in $G(S)$, then there is a path from u to v in $G(S)$. If u and v belong to different SCCs in $G(S)$, then by the construction of $G'(S)$, they represent their respective SCCs. Since $G'(S)$ contains the edge (u, v), there exists an edge (u', v') in $G(S)$ where u' belongs to the SCC of u and v' belongs to the SCC of v. Moreover, $G(S)$ has a path from u to u' and a path from v' to v. Thus, a path from u to v exists in G that includes (u', v').

Next, we show that $G'(S)$ is acyclic, which implies that the derived binary relation over $\mathcal{T}(S)$ is irreflexive. Assume for contradiction that $G'(S)$ contains a cycle, i.e., there exists a path from u to v and a path from v to u in $G'(S)$. By the previous argument, $G(S)$ must then contain a path from u to v and a path from v to u, meaning that u and v belong to the same SCC in $G(S)$. However, since u and v have outgoing edges in $G'(S)$, they must be representative nodes of different SCCs, contradicting the fact that they are in the same SCC. Thus, $G'(S)$ cannot contain a cycle. ◀

Finally, we show the maximality of a contracted transition dominance graph, i.e., we cannot use more transition dominance rules while keeping irreflexivity.

▶ **Proposition 11.** *Let $G(S)$ be the transition dominance graph for a state S and $G'(S)$ be a contracted transition dominance graph. Then, there does not exist an acyclic graph $G''(S)$ with the set of nodes $\mathcal{T}(S)$ satisfying all of the following properties:*

1. *If $G''(S)$ has a path from τ to τ', then $G(S)$ also has a path from τ to τ'.*
2. *If a node τ has an incoming edge in $G'(S)$, then τ does so also in $G''(S)$.*
3. *There exists a node τ that has an incoming edge in $G''(S)$ but not in $G'(S)$.*

Proof. Assume that an acyclic graph $G''(S)$ with the described properties exists. Based on the third property, let τ be a node that has an incoming edge in $G''(S)$ but not in $G'(S)$. By construction of $G'(S)$, τ must be the representative node of an SCC in $G(S)$. Let N' be the set of nodes in this SCC. In $G(S)$, each node $v \in N'$ does not have incoming edges from nodes in different SCCs; otherwise, an edge from a representative node in a different SCC to τ should have been added to G'.

In $G'(S)$, since τ is the representative node of N', each node $v \in N' \setminus \{\tau\}$ has incoming edge (τ, v). By the second property, in $G''(S)$, each node $v \in N' \setminus \{\tau\}$ has an incoming edge. Since τ also has an incoming edge in $G''(S)$, all nodes in N' have incoming edges in $G''(S)$. By the previous paragraph, an incoming edge to each $v \in N'$ must be from a node in N'. In $G''(S)$, since each node in N' has an incoming edge from another node in N', there must be a cycle, contradicting our assumption. ◀

Note that if two transitions dominate each other, the choice of which dominance to enforce can impact search efficiency. However, by definition, our model provides no basis for preferring one over the other. In practice, we use Tarjan's algorithm to detect all SCCs and keep the first node visited by depth-first search as the representative node of each SCC.

Note that using the transition dominance interface and preconditions offer different advantages in terms of computational efficiency. Suppose that a transition τ is dominated by τ' if $S \models c_{\tau' \preceq \tau}$ and by τ'' if $S \models c_{\tau'' \preceq \tau}$. Given a state S with $\mathcal{T}(S) = \{\tau, \tau', \tau''\}$, the transition dominance interface requires evaluating $c_{\tau' \preceq \tau}$ and $c_{\tau'' \preceq \tau}$ and performing SCC extraction to construct a contracted transition dominance graph. However, when transition dominance is modeled with preconditions, we can avoid evaluating $c_{\tau'' \preceq \tau}$ once we detect $S \models c_{\tau' \preceq \tau}$. In contrast, using preconditions requires restricting conditions for transition dominance for tie-breaking, which may result in lost opportunities to exploit certain transition dominances within a state. Therefore, selecting the appropriate method depends on the trade-off between computational overhead and the effectiveness of transition dominance exploitation.

4.2 Avoiding Redundant Computation using State Functions

Modelling transition dominance typically requires pairwise comparisons of all applicable transitions of a state. To prevent the generation of dominated solutions, the condition $c_{\tau \preceq \tau'}$ should be evaluated for all pairs (τ, τ') of transitions where τ potentially dominates τ'. However, evaluating these conditions often results in redundant computations.

▶ **Example 12.** Consider the problem in Example 5 and its corresponding model in Listing 1. The expression $\max(t, r_i) + p_i$ for all transitions, which is represented by the `next_time` array in the model, appears in the cost expression, the effects of the transition τ_i for job i, and the condition for transition dominance when comparing τ_i with other transitions. Suppose there are n jobs in total. The expression $\max(t, r_i) + p_i$ needs to be evaluated $O(n)$ times when generating applicable transitions and removing dominated transitions. By caching the values of such redundant evaluations, the number of times to evaluate such expression can be reduced to $O(1)$ for each transition. ⌟

To address this observation, we introduce *state functions* in DyPDL. State functions are defined by expressions using state variables and can be used in preconditions, effects, and cost expressions, just as state variables. During the solving process, state functions are lazily computed and cached. The following shows the usage in the job sequencing example.

```
next_time = [dp.max(t, r[i]) + p[i] for i in all_jobs]
next_time_sf = [dp.add_int_state_fun(next_time[i]) for i in all_jobs]
```

To use state functions in Listing 1, we define state functions `next_time_sf` for all transitions τ_i and replace all occurrences of `dp.max(t, r[i]) + p[i]` with `next_time_sf[i]`. When checking if the condition to test whether τ_0 dominates τ_1, the expression `next_time[0]` is computed and cached, and for all other comparison between τ_0 and τ_j for $j = 2, \ldots, n$, the value is reused and therefore redundant computation of `next_time[0]` is avoided. State functions are similar to axioms in PDDL [28], which are derived predicates whose truth is inferred from values of some basic predicates, while results of state functions can be Boolean, integer, or set values.

As we will show in the experimental evaluation, using state functions is sometimes crucial in exploiting transition dominance efficiently. Additionally, state functions also avoid duplicate recomputation in the original models for problems such as Talent Scheduling [25] and OPTW [11]. More detailed descriptions of problems and models are in Appendix B.

5 Case Studies

In this section, we demonstrate the applicability of transition dominance across various optimization problems. For each problem, we describe its DyPDL model and the identified transition dominance. All proofs of propositions in this section are in Appendix A.

5.1 Aircraft Landing

The aircraft landing problem [18] involves scheduling the landing of a set of aircraft on multiple runways to minimize the delay from the target landing times. The aircraft are partitioned into multiple classes, and it is assumed that each class follows a first-come-first-serve sequence according to the target landing time. The decision at a state, then, is to determine which class of aircraft should land next on each runway. The model uses state $S = (\vec{n}, \vec{l}, \vec{c})$, where $n_i \in \vec{n}$ is the index of the next aircraft for each class i, $l_j \in \vec{l}$ is the most recent landing time for runway j, and $c_j \in \vec{c}$ is the class of the most recently landed aircraft for runway j. The target and latest landing times of the next aircraft of class i are t_{i,n_i} and p_{i,n_i}, respectively.

Landing two aircrafts of class i and i' consecutively on the same runway must respect the minimum separation time $sep_{i,i'}$. A transition $\tau_{i,j}$ lands the next aircraft of class i on runway j with the delay $d(S, i, j) = (l_j + \mathrm{sep}_{c_j,i} - t_{i,n_i})^+$, where $(x)^+$ is 0 if $x < 0$ and x otherwise. This transition is applicable only when $n_i > 0$ and the actual landing time is no later than p_{i,n_i}. The optimal value is computed as:

$$V(S) = 0, \qquad \text{if } n_i = 0, \forall i$$
$$V(S) = \min_{\tau_{i,j} \in \mathcal{T}(S)} \{d(S, i, j) + V(S[\![\tau_{i,j}]\!])\} \qquad \text{else}$$

where transition $\tau_{i,j}$ updates n_i to $n_i - 1$, l_j to $\max(l_j + \mathrm{sep}_{c_j,i}, t_{i,n_i})$, and c_j to i.

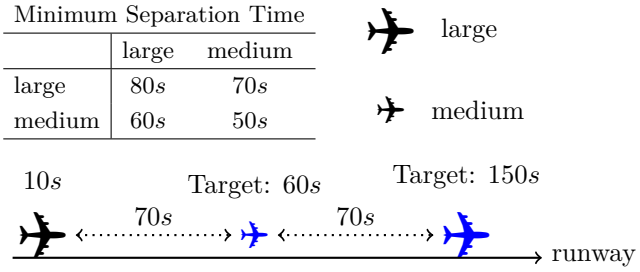


Figure 2 An example instance of aircraft landing.

We follow Coppé et al. [6] to add state dominance that $\vec{l}$ are integer resource variables where less is preferred. We define transition dominance where landing the aircraft for class i is better than landing another class i' on the same runway if the former can be landed before $t_{i',n_{i'}} - \text{sep}_{i,i'}$. Figure 2 shows an example with two aircraft classes. A large aircraft has landed on the runway at $10s$, and the target landing times for the next medium and large aircraft are $60s$ and $150s$ respectively. The target landing time of the next large aircraft is so late that the next medium aircraft can land first without delaying the large aircraft.

▶ **Proposition 13.** *Suppose minimum separation times satisfy the triangle inequality:* $\text{sep}_{i,j} + \text{sep}_{j,k} \geq \text{sep}_{i,k}$ *for any classes* i, j, k*. If* $\tau_{i,j}, \tau_{i',j} \in \mathcal{T}(S)$ *satisfy*

$$\max(l_j + \text{sep}_{c_j,i}, t_{i,n_i}) \leq t_{i',n_{i'}} - \text{sep}_{i,i'}$$

at the current state S*, then* $cost_{\tau_{i,j}}(V(S[\![\tau_{i,j}]\!]), S) \leq cost_{\tau_{i',j}}(V(S[\![\tau_{i',j}]\!]), S)$. ⌟

The transition dominance described above schedules a task, such as landing an aircraft, before the start of another task. This type of transition dominance also applies to other problems, such as the problem in Example 4, the Orienteering Problem with Time Windows (OPTW) and the Travelling Salesman Problem with Time Windows and Makespan Objective (TSPTW-M). Detailed problem descriptions are provided in Appendix B.

5.2 Graph-Clear

The Graph-Clear problem arises from multi-robot surveillance tasks [12]. The problem aims to find a sequence of steps to decontaminate ("sweep") nodes in an undirected graph (N, E) with all nodes being initially contaminated. The objective is to minimize the maximum number of robots used over all steps in a given indoor environment modelled as a graph. Each node $i \in N$ can be swept using a_i robots, and each edge $(i, j) \in E$ can be blocked using b_{ij} robots. If we sweep a node $i \in N$ in one step, we must use robots for sweeping i, blocking edges connecting i, and blocking edges connecting all swept and unswept nodes to avoid recontamination. Figure 3 shows an example instance where the numbers on nodes and edges represent the number of robots required for node sweeping and edge blockage, respectively.

The DyPDL model proposed by Kuroiwa and Beck [13] uses a set state variable C to represent swept nodes. A transition τ_i for each node $i \in N$ is defined to add a node i into C and has the precondition $i \notin C$. The number of robots used to sweep i at a state C is $R(i, C) = a_i + \sum_{j \in N} b_{ij} + \sum_{j \in \overline{C} \setminus \{i\}} \sum_{k \in C} b_{jk}$, and an optimal solution minimizes the maximum number of robots used at any step. The optimal value is computed as follows:

$$V(C) = 0, \qquad \text{if } C = N$$

$$V(C) = \min_{i \in \overline{C}} \max\{R(i, C), V(C \cup \{i\})\} \qquad \text{else}$$

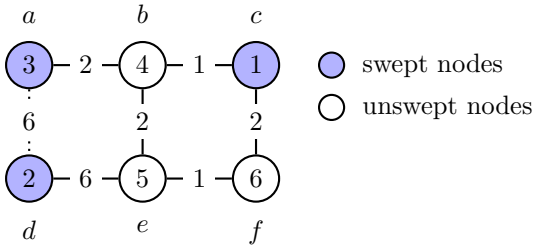


Figure 3 An example instance of Graph-Clear.

We define a transition dominance for the Graph-Clear problem inspired by the customer search model for the Minimization of Open Stacks Problem (MOSP) [4]. In Graph-Clear, observe that in any sequence, a transition τ_i always requires the same number of robots to sweep a node i and block the edges connecting i to its neighbors. The difference arises in the number of robots required to block the *cutting edges* that connect swept nodes in C and unswept nodes in $\overline{C} \setminus \{i\}$. If a transition τ_i reduces the weight of the cutting edges, sweeping any other unswept node i' after applying τ_i requires fewer robots compared to sweeping i' in the current state. Moreover, if τ_i uses fewer robots compared to transition $\tau_{i'}$, then the sequence beginning with transition τ_i then $\tau_{i'}$ uses fewer robots at all steps compared to the corresponding sequence beginning with $\tau_{i'}$.

Consider the instance in Figure 3. Sweeping node b in the next step is preferable to sweeping node e. First, the weight of the cutting edges after sweeping b becomes $(6{+}2{+}2) = 10$, which is smaller than the current state $(2 + 1 + 6 + 2) = 11$. Sweeping e and f afterwards will require fewer robots. Second, sweeping b requires only $4 + (2 + 2 + 1) + (6 + 2) = 17$ robots, which is fewer than the $5 + (2 + 6 + 1) + (2 + 1 + 2) = 19$ robots needed to sweep e. We can conclude that sweeping b then e is always better than sweeping e from the current state. This can be formulated as transition dominance in the model.

▶ **Proposition 14.** *Suppose $i, i' \in \overline{C}$ for a state in the DyPDL model of the Graph-Clear problem. If we have $i \neq i'$ and*

$$a_i + \sum_{j \in \overline{C}} b_{ij} \leq a_{i'} + \sum_{j \in \overline{C}} b_{i'j} \tag{2a}$$

$$\sum_{j \in \overline{C}} b_{ij} \leq \sum_{j \in C} b_{ij} \tag{2b}$$

at a state S, then $cost_{\tau_i}(V(S[\![\tau_i]\!]), S) \leq cost_{\tau_{i'}}(V(S[\![\tau_{i'}]\!]), S)$. ⌟

5.3 Discrete Lot Sizing

The discrete lot-sizing problem (DLSP) is a production planning problem for items of various types on a single machine [24]. Each type has a set of items, and the j^{th} item of type i must be produced before its due period $d_{i,j}$. A changeover cost $c_{i,i'}$ is incurred when the machine switches from producing an item of type i to an item of type i'. Additionally, the stocking cost is calculated as the product of the unit cost s_i and the difference between the production period and $d_{i,j}$.

We propose a *sequence model* that makes decisions based on the reversed production sequence of item types. The model uses three types of state variables: an integer variable q represents the latest available period for producing items, t represents the type of item

Item Parameters

Type	Item No.	$d_{i,j}$	s_i
1	1	3	2
	2	4	2
	3	6	2
	4	9	2
2	1	7	3
	2	9	3

Changeover Costs

	Type 1	Type 2
Type 1	0	3
Type 2	2	0

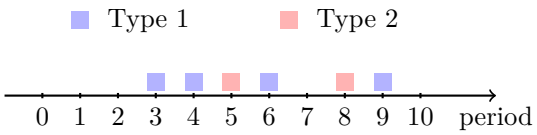


Figure 4 An example instance and solution sequence of DSLP.

chosen in the last decision, and $\vec{r}$ is a vector where $r_i \in \vec{r}$ indicates the remaining demand for each type i. A transition τ_i represents the decision to produce an item of type i. Since backlogging is not allowed, the item is produced at the period $\min(d_{i,r_i}, q)$, and the available period is updated to $\min(d_{i,r_i}, q) - 1$. The total cost of the transition is defined as $c(S, i) = c_{i,t} + s_i \cdot (d_{i,r_i} - q)^+$.

Figure 4 illustrates an example instance and a corresponding solution sequence $\sigma = \langle \tau_1, \tau_2, \tau_1, \tau_2, \tau_1, \tau_1 \rangle$. Initially, $q = 10$ and $\vec{r} = \langle 4, 2 \rangle$. The first transition τ_1 selects an item of type 1 to produce at period 9, which is the latest due period for type 1 items. The state variables are updated as follows: q becomes $\min(d_{1,4}, 10) - 1 = 8$, t is updated to 1, and $\vec{r}$ is updated to $\langle 3, 2 \rangle$. The second transition τ_2 selects an item of type 2 to produce at period 8, incurring a stocking cost of $s_2 \cdot (d_{2,r_2} - q)^+ = 3 \cdot (9 - 8) = 3$ and a changeover cost of $c_{2,1} = 2$. The sequence continues until either all items are produced or the sequence becomes invalid, as the available production period q becomes less than the total number of remaining items.

Formally, the DP model can be expressed using the following Bellman equation:

$$
\begin{aligned}
V(S) &= 0, && \text{if } \forall i, r_i = 0 \\
V(S) &= \infty, && \text{if } \textstyle\sum_i r_i > q \\
V(S) &= \min_{\tau_i \in \mathcal{T}(S)} \{c(S, i) + V(S[\![\tau_i]\!])\}, && \text{otherwise}
\end{aligned}
$$

Observe that in the example sequence in Figure 4, the machine is idle at period 7. An item of type 2 could have been produced at this period to reduce the stocking cost. Additionally, this would save the changeover cost incurred from switching from a type 2 item to a type 1 item at period 5. We could verify whether τ_2 dominates τ_1 after applying the prefix transitions $\langle \tau_1, \tau_2 \rangle$. If τ_2 indeed dominates τ_1, the example sequence can be safely discarded, as it is guaranteed not to be optimal. The following proposition formally characterizes the transition dominance observed in this example.

▶ **Proposition 15.** *Suppose the changeover costs satisfy the triangle inequality. If $\tau_i, \tau_{i'} \in \mathcal{T}(S)$ where $i \neq i'$ satisfy:*

- *i has a later production period: $d_{i',r_{i'}} < \min(d_{i,r_i}, q)$,*
- *the total cost is less: $c_{i',i} + c_{i,t} \leq c_{i',t} + 2s_i$,*

at a state S, then $cost_{\tau_i}(V(S[\![\tau_i]\!]), S) \leq cost_{\tau_{i'}}(V(S[\![\tau_{i'}]\!]), S)$. ⌟

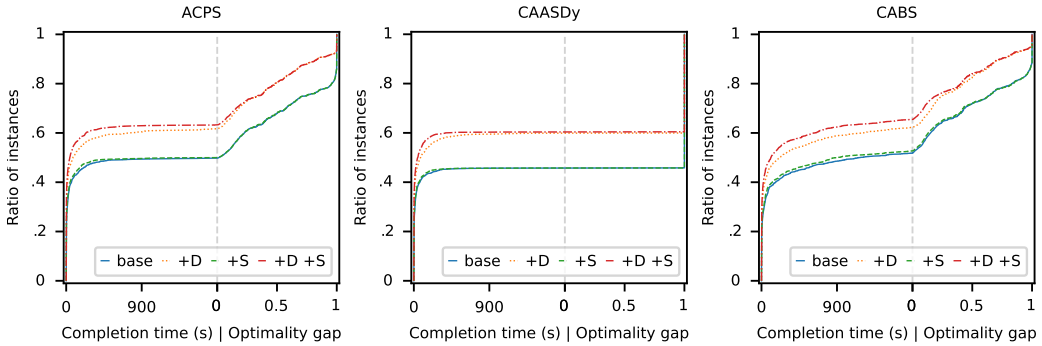


Figure 5 The ratio of instances against time and optimality gap averaged by all problem classes.

Note that Coppé, Gillard, and Schaus [5] propose a similar model which introduces a auxiliary idle transition and makes decisions for each period in reverse, starting from the last period considered in the planning horizon. Preliminary experiments show that solving the sequence model is more efficient than the period model.

6 Experimental Evaluation

In this section, we experimentally evaluate the impact of using transition dominance and state functions on seven combinatorial optimization problems. For Graph-Clear, OPTW, and TSPTW-M, we modify the existing models[1] by incorporating additional state functions and transition dominance. Following Kuroiwa and Beck [13, 14], we use benchmark instances from the literature. Further details on these instances are provided in Appendix C. Additionally, we introduce new DyPDL models for four problems:

- One Machine Scheduling Minimizing Total Weighted Tardiness ($1|r_i|\sum w_iT_i$): We implement the model based on the one for $1||\sum w_iT_i$ from a public repository,[1] and we generate 300 instances where the number of tasks $n \in \{20, 25, 30, 35, 40\}$, $\alpha \in \{0, 0.5, 1, 1.5\}$ and $\beta \in \{0.05, 0.25, 0.5\}$ control the distribution of release and due dates of tasks. We implement transition dominance following Akturk and Ozdemir [1].
- Aircraft Landing (ALP): transition dominance is implemented based on the DP presented by Lieder, Briskorn and Stolletz [18], and 720 instances from Coppé et al. [6] are used.
- Talent Scheduling (Talent-Sched): the double-ended DP model and the dominance from Qin et al. [25] are implemented in DyPDL, and 1000 instances are sampled from those by Garcia de la Banda and Stuckey [7] in the same way as Kuroiwa and Beck [13].
- Discrete Lot Sizing Problem (DSLP): we implement the sequence model and generate 360 instances following Coppé et al. [5] with the number of items in $\{4, 6, 8, 10\}$, the number of periods in $\{100, 120, 140, 160, 180, 200\}$, and the density in $\{0.7, 0.8, 0.9\}$. The stocking costs and changeover costs are sampled uniformly in $[10, 50]$ and $[20, 40]$ respectively.

The transition dominance interface and state functions are implemented in didp-rs v0.7.3[2] using Rust 1.78.0, and all models are implemented in Python 3.9.16 using DIDPPy, a Python interface for didp-rs. All experiments are run on an Intel Xeon-Gold 6150 processor with a single thread, an 8 GB memory limit, and a time limit of 1800 seconds.

[1] https://github.com/Kurorororo/didp-models
[2] https://didp.ai/

Table 1 Experimental results for co-solved instances.

Problem	Solver	Average Solving Time				Average Expanded Nodes		
		base	+D	+S	+D+S	base	+D	reduction
$1\|r_i\|\sum w_iT_i$	ACPS	23.98	1.99	24.03	**1.97**	1803688.77	183502.68	89.83%
	CABS	153.89	3.38	153.46	**3.22**	14889711.97	374543.79	97.48%
	CAASDy	19.98	1.76	20.39	**1.65**	1298236.12	132313.69	89.81%
ALP	ACPS	69.90	47.71	69.55	**43.44**	5290237.80	3154905.57	40.36%
	CABS	165.68	106.90	159.41	**92.69**	14235715.54	8914310.72	37.38%
	CAASDy	67.66	49.18	68.56	**44.61**	3951671.66	2503272.76	36.65%
Graph-Clear	ACPS	34.25	142.44	**26.17**	30.13	1245620.20	947797.81	23.91%
	CABS	40.12	168.04	**24.04**	28.94	1734871.17	952846.14	45.08%
	CAASDy	8.36	17.51	7.11	**4.35**	225567.74	138467.09	38.61%
Talent-Sched	ACPS	191.24	102.09	116.47	**15.11**	1894764.81	394158.89	79.20%
	CABS	238.81	150.64	142.33	**26.35**	2387816.79	665926.33	72.11%
	CAASDy	16.47	13.62	11.28	**2.83**	205820.52	77316.28	62.44%
OPTW	ACPS	57.08	43.49	42.77	**34.74**	1434287.87	1234402.62	13.94%
	CABS	188.23	146.80	143.97	**114.26**	3877856.20	3352288.70	13.55%
	CAASDy	47.48	40.79	35.98	**31.29**	1365988.53	1175550.17	13.94%
TSPTW-M	ACPS	31.97	29.93	15.23	**14.76**	673951.74	619372.42	8.10%
	CABS	98.45	91.26	53.30	**50.64**	2013448.95	1804993.92	10.35%
	CAASDy	30.80	28.19	15.62	**14.53**	623198.30	557875.92	10.48%
DSLP	ACPS	37.72	13.57	37.59	**12.00**	5664688.02	1142853.46	79.82%
	CABS	249.48	105.06	255.71	**93.90**	27283103.00	6743668.77	75.28%
	CAASDy	27.02	10.62	27.33	**9.38**	3313961.61	783309.53	76.36%

We experiment with three search algorithms: Anytime Column Progressive Search (ACPS), Complete Anytime Beam Search (CABS). and Cost-Algebraic A* Search (CAASDy). These solvers usually solve the most instances subject to the limits and exhibit representative tendencies [13] in the DIDP framework. We compare four models for each problem: the "base" model and the base model plus transition dominance "+D", state functions "+S", and both transition dominance and state functions "+D+S".

We compare four configurations of each solving algorithms based on two metrics: (1) *coverage*, which is the number of instances for which optimality or infeasibility is proven within time and memory limits, and (2) *optimality gap*, which is the relative difference between the primal and dual bounds, with a value between 0 and 1. If no solution is found and the instance is not proven infeasible, we set the optimality gap to be 1. Figure 5 illustrates the cumulative ratio of solved instances with respect to completion time and optimality gap. On the left-hand side of each subfigure, the x-axis represents time in seconds, and the y-axis represents the ratio of coverage achieved within x seconds to the total number of instances. On the right-hand side, the x-axis represents the optimality gap, and the y-axis represents the ratio of instances where the optimality gap is less than or equal to x. A curve that is higher and further to the left indicates better performance, which means more instances are solved within a given time or achieve lower optimality gaps within the time limit.

As shown in Figure 5, "+D" and "+D+S" improve the performance of all search algorithms substantially. Upon inspecting the detailed results, combining both transition dominance and state functions enables ACPS, CAASDy, and CABS to solve 420, 450, and 426 more instances in total, respectively, and reduces the average optimality gap by 0.114, 0.155, and

0.106, respectively, within the limits. Compared with ACPS and CABS, the CAASDy solver exhibits an interesting plateau pattern after initial increases, as its first solution is usually optimal: it is guaranteed for our DP models of $1|r_i|\sum w_iT_i$, ALP, Graph-Clear, Talent-Sched, and DSLP in theory, and it is usually the case with OPTW and TSPTW-M in practice. The optimality gap remains 1 even though the dual bound is improved. Overall, the gaps between "+D+S" and "base" show that transition dominance and state functions enable the algorithms to solve instances completely with less time and reduce the primal-dual gaps for unsolved instances considerably.

The coverage and optimality gap may be affected by the time-out and memory limits used in our experiments. To better analyze speed-ups, we also report the average solving time and the average number of expanded nodes in Table 1 for *co-solved instances* that can be solved by all four configurations of each solver. Since the number of expanded nodes remains unchanged when state functions are used, we omit the results for "+S" and "+D+S" in this metric. The fastest average time for each solver is highlighted, and the percentage reduction in expanded nodes is provided for reference.

From the results, we observe that using transition dominance alone ("+D") reduces the number of expanded nodes across all problems and decreases solving time for all problems except Graph-Clear. The "+S" configuration is particularly beneficial in Talent-Sched and OPTW, where state functions help avoid recomputations in their original formulations. Combining both transition dominance and state functions further reduces the average solving time compared to the "base" setup. Overall, the average speed-ups across all co-solved instances for ACPS, CABS, and CAASDy are 10.10, 8.87, and 5.81, respectively.

In the Graph-Clear problem, evaluating transition dominance involves costly summation computations over sets, which significantly slows down performance when applied alone. A closer examination reveals that transition dominance is more effective in reducing expanded nodes for low-density graphs. Combining state functions with transition dominance is essential to achieve better results. In contrast, the "+S" configuration results in similar or higher average times than "base" in $1|r_i|\sum w_iT_i$, ALP, and DSLP, as state functions mostly involve simple integer or Boolean expressions that do not benefit much from caching. They become slightly more effective in "+D+S" when evaluating transition dominance requires more reuses of the cached values. Exploring the trade-offs in using transition dominance and state functions in different problem domains is an interesting direction for future research.

7 Concluding Remarks

In this paper, we define *transition dominance* within the framework of DIDP and introduce new constructs, the *transition dominance interface* and *state function*, into DyPDL to facilitate effective and efficient modelling of transition dominance. We also demonstrate the broad applicability of transition dominance by presenting several previously unexploited cases and show that the insights gained from transition dominance can be applied across problem domains with common combinatorial substructures. Our experimental results indicate that incorporating transition dominance enhances the solving process of various search algorithms in DIDP, reducing computational time and improving solution quality across a range of combinatorial optimization problems.

An interesting research direction is to study whether transition dominance and state functions can be extracted automatically by analyzing DyPDL models. In the benchmark problems studied in this paper, the dominance rules follow recurring patterns. Specifically, transition dominance applies when one can construct a better solution, and the way we

construct such a better solution is usually by advancing (shifting forward) a transition. The goal is to identify sufficient conditions under which the objective value of the new solution improves. These construction patterns could serve as a basis for a general method to infer automatically transition dominance from a problem model. Previous work has explored automatic generation of dominance-breaking constraints from constraint programming models [16, 17], and similar techniques may be applicable to analyzing DyPDL models to derive state dominance and transition dominance.

State functions capture common subexpressions and can be extracted through the analysis of DyPDL models, but there is a tradeoff between recomputation and caching. Retrieving results from cache can prevent the recomputation of computationally costly expressions. However, for simple expressions such as basic comparisons, recomputation is more efficient than caching in terms of time and space. Analyzing the trade-off is important for extracting state functions automatically.

References

1 M Selim Akturk and Deniz Ozdemir. An exact approach to minimizing total weighted tardiness with release dates. *IIE Transactions*, 32:1091–1101, 2000. doi:10.1023/A:1013741325877.

2 Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

3 TC Cheng, JE Diamond, and BM Lin. Optimal scheduling in film production to minimize talent hold cost. *Journal of Optimization Theory and Applications*, 79(3):479–492, 1993. doi:10.1007/BF00940554.

4 Geoffrey Chu and Peter J. Stuckey. Minimizing the maximum number of open stacks by customer search. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*, pages 242–257. Springer, 2009. doi:10.1007/978-3-642-04244-7_21.

5 Vianney Coppé, Xavier Gillard, and Pierre Schaus. Decision diagram-based branch-and-bound with caching for dominance and suboptimality detection. *INFORMS Journal on Computing*, 36(6):1522–1542, 2024. doi:10.1287/IJOC.2022.0340.

6 Vianney Coppé, Xavier Gillard, and Pierre Schaus. Modeling and exploiting dominance rules for discrete optimization with decision diagrams. In Bistra Dilkina, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research, CPAIOR 2024*, volume 14742 of *Lecture Notes in Computer Science*, pages 226–242. Springer, 2024. doi:10.1007/978-3-031-60597-0_15.

7 Maria Garcia de la Banda, Peter J. Stuckey, and Geoffrey Chu. Solving talent scheduling with dynamic programming. *INFORMS Journal of Computing*, 23(1):120–137, 2011. doi:10.1287/IJOC.1090.0378.

8 Yvan Dumas, Jacques Desrosiers, Éric Gélinas, and Marius M. Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations Research*, 43(2):367–371, 1995. doi:10.1287/OPRE.43.2.367.

9 Michel Gendreau, Alain Hertz, Gilbert Laporte, and Mihnea Stan. A generalized insertion heuristic for the traveling salesman problem with time windows. *Operations Research*, 46(3):330–335, 1998. doi:10.1287/OPRE.46.3.330.

10 Xavier Gillard, Pierre Schaus, and Vianney Coppé. Ddo, a generic and efficient framework for mdd-based optimization. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 5243–5245. International Joint Conferences on Artificial Intelligence Organization, 2020. doi:10.24963/IJCAI.2020/757.

11 Marisa G Kantor and Moshe B Rosenwein. The orienteering problem with time windows. *Journal of the Operational Research Society*, 43(6):629–635, 1992. doi:10.1057/jors.1992.88.

12 Andreas Kolling and Stefano Carpin. The GRAPH-CLEAR problem: definition, theoretical properties and its connections to multirobot aided surveillance. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1003–1008. IEEE, 2007. doi:10.1109/IROS.2007.4399368.

13 Ryo Kuroiwa and J Christopher Beck. Domain-independent dynamic programming: Generic state space search for combinatorial optimization. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 33, pages 236–244, 2023. doi:10.1609/ICAPS.V33I1.27200.

14 Ryo Kuroiwa and J Christopher Beck. Solving domain-independent dynamic programming problems with anytime heuristic search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 33, pages 245–253, 2023. doi:10.1609/ICAPS.V33I1.27201.

15 Ryo Kuroiwa and J Christopher Beck. Domain-independent dynamic programming. *arXiv preprint arXiv:2401.13883*, 2024. doi:10.48550/arXiv.2401.13883.

16 Jimmy H.M. Lee and Allen Z Zhong. Automatic generation of dominance breaking nogoods for a class of constraint optimization problems. *Artificial Intelligence*, 323:103974, 2023. doi:10.1016/j.artint.2023.103974.

17 Jimmy H.M. Lee and Allen Z Zhong. Exploiting functional constraints in automatic dominance breaking for constraint optimization. *Journal of Artificial Intelligence Research*, 78:1–35, 2023. doi:10.1613/jair.1.14714.

18 Alexander Lieder, Dirk Briskorn, and Raik Stolletz. A dynamic programming approach for the aircraft landing problem with aircraft classes. *European Journal of Operational Research*, 243(1):61–69, 2015. doi:10.1016/j.ejor.2014.11.027.

19 Manuel López-Ibáñez, Christian Blum, Jeffrey W Ohlmann, and Barrett W Thomas. The travelling salesman problem with time windows: Adapting algorithms from travel-time to makespan optimization. *Applied Soft Computing*, 13(9):3806–3815, 2013. doi:10.1016/j.asoc.2013.05.009.

20 Laurent Michel and Willem-Jan van Hoeve. CODD: A decision diagram-based solver for combinatorial optimization. In Ulle Endriss, Francisco S. Melo, Kerstin Bach, Alberto José Bugarín Diz, Jose Maria Alonso-Moral, Senén Barro, and Fredrik Heintz, editors, *ECAI 2024 - 27th European Conference on Artificial Intelligence, 19-24 October 2024, Santiago de Compostela, Spain - Including 13th Conference on Prestigious Applications of Intelligent Systems (PAIS 2024)*, volume 392 of *Frontiers in Artificial Intelligence and Applications*, pages 4240–4247. IOS Press, 2024. doi:10.3233/FAIA240997.

21 Roberto Montemanni, Gambardella Luca Maria, et al. An ant colony system for team orienteering problems with time windows. *arXiv preprint arXiv:2305.07305*, 2009. doi:10.48550/arXiv.2305.07305.

22 David R Morrison, Sheldon H Jacobson, Jason J Sauppe, and Edward C Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79–102, 2016. doi:10.1016/j.disopt.2016.01.005.

23 Jeffrey W Ohlmann and Barrett W Thomas. A compressed-annealing heuristic for the traveling salesman problem with time windows. *INFORMS Journal on Computing*, 19(1):80–90, 2007. doi:10.1287/ijoc.1050.0145.

24 Yves Pochet and Laurence A Wolsey. *Production planning by mixed integer programming*, volume 149. Springer, 2006. doi:10.1007/0-387-33477-7.

25 Hu Qin, Zizhen Zhang, Andrew Lim, and Xiaocong Liang. An enhanced branch-and-bound algorithm for the talent scheduling problem. *European Journal of Operational Research*, 250(2):412–426, 2016. doi:10.1016/j.ejor.2015.10.002.

26 Giovanni Righini and Matteo Salani. Dynamic programming for the orienteering problem with time windows. Technical report, Università degli Studi di Milano-Polo Didattico e di Ricerca di Crema, 2006. URL: https://air.unimi.it/handle/2434/6449.

27 Giovanni Righini and Matteo Salani. Decremental state space relaxation strategies and initialization heuristics for solving the orienteering problem with time windows with dynamic programming. *Computers & operations research*, 36(4):1191–1203, 2009. doi:10.1016/j.cor.2008.01.003.

28 Sylvie Thiébaux, Jörg Hoffmann, and Bernhard Nebel. In defense of PDDL axioms. *Artificial Intelligence*, 168(1-2):38–69, 2005. doi:10.1016/j.artint.2005.05.004.

29 Pieter Vansteenwegen, Wouter Souffriau, Greet Vanden Berghe, and Dirk Van Oudheusden. Iterated local search for the team orienteering problem with time windows. *Computers & Operations Research*, 36(12):3281–3290, 2009. doi:10.1016/j.cor.2009.03.008.

A Proofs of Propositions

Proof for Proposition 3. The proposition is trivial if $V(S^0) = \infty$, as removing transitions in any state does not reduce the cost of that state. Let Ω denote the set of S^0-solutions, which contains a finite number of elements by the assumptions of finiteness of a DyPDL model. We aim to show that there exists an optimal solution such that no transition in the solution is dominated. To achieve this, we define a relation $\mathcal{R}$ over Ω. For any two solutions $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_r \rangle$ and $\sigma' = \langle \sigma'_1, \sigma'_2, \ldots, \sigma'_s \rangle \in \Omega$, we say $\sigma \mathcal{R} \sigma'$ if and only if there exists $t \in \mathbb{Z}_{\geq 0}$ such that: (1) $t \leq r$ and $t \leq s$, (2) $\sigma_i = \sigma'_i$ for all $i \leq t$, and (3) either $\sigma_{t+1} \prec^S \sigma'_{t+1}$, where $S = S^0[\![\sigma_{:t}]\!] = S^0[\![\sigma'_{:t}]\!]$, or $t = r$ and $t < s$. Here, we denote the state resulting from applying $\langle \sigma_1, ..., \sigma_t \rangle$ in S^0 by $S^0[\![\sigma_{:t}]\!]$.

It is straightforward to verify that $\mathcal{R}$ is both transitive and irreflexive. Now, suppose an optimal solution σ^0 for S^0 is pruned due to the restriction of $\mathcal{T}$ to $\mathcal{T}^*$. Then, there must exist another solution σ^1 such that $\sigma^1 \mathcal{R} \sigma^0$. By Definition 2 and Principle of Optimality, $\mathsf{solution_cost}(\sigma^1, S^0) \leq \mathsf{solution_cost}(\sigma^0, S^0)$, and σ^1 is also an optimal solution. By repeating this process, we can construct a sequence of optimal solutions $\sigma^0, \sigma^1, \ldots$ such that $\sigma^{i+1} \mathcal{R} \sigma^i$. Since $\mathcal{R}$ is transitive and irreflexive, and the set of optimal solutions is a finite subset of Ω, the sequence cannot repeat indefinitely. The sequence must terminate at some σ^k, which is optimal and not pruned by replacing $\mathcal{T}$ with $\mathcal{T}^*$. ◀

Proof for Proposition 13. We need to show that for any S-solution $\langle \tau_{i',j}; \sigma \rangle$ beginning with $\tau_{i',j}$, there exists a better solution beginning with $\tau_{i,j}$. Suppose $\tau_{i,k} \in \sigma$ where aircraft i lands on runway k. We can always construct another S-solution $\langle \tau_{i,j}, \tau_{i',j}; \sigma' \rangle$, where σ' is σ excluding $\tau_{i,k}$. The landing times of all aircraft on runway j will not change since the landing time of the first aircraft i' is still $t_{i',n_{i'}} = \max(\max(\mathrm{sep}_{c_j,i} + l_j, t_{i,n_i}) + \mathrm{sep}_{i,i'}, t_{i',n_{i'}})$. The landing time of all aircraft on runway k cannot be any later since we are removing an aircraft and the minimum separation times satisfy the triangle inequality. ◀

Proof for Proposition 14. For any S-solution $\langle \tau_{i'}; \sigma \rangle$ beginning with $\tau_{i'}$, τ_i must be in σ since $\tau_i \in \mathcal{T}(S)$ and $i \notin C$. We can construct another solution $\langle \tau_i; \tau_{i'}; \sigma' \rangle$, where σ' is σ excluding τ_i. We now prove that the constructed solution uses an equal or lower number of robots at each step. First, the number of robots required to sweep node i does not exceed that required to sweep node i' at state C:

$$\begin{aligned} R(i, C) &= a_i + \sum_{j \in N} b_{ij} + \sum_{j \in \overline{C} \setminus \{i\}} \sum_{k \in C} b_{jk} \\ &= a_i + \sum_{j \in N} b_{ij} - \sum_{k \in C} b_{ik} + \sum_{j \in \overline{C}} \sum_{k \in C} b_{jk} \end{aligned}$$

$$
\begin{aligned}
&= a_i + \sum_{j \in N \setminus C} b_{ij} + \sum_{j \in \overline{C}} \sum_{k \in C} b_{jk} \\
&\leq a_{i'} + \sum_{j \in N \setminus C} b_{i'j} + \sum_{j \in \overline{C}} \sum_{k \in C} b_{jk} \\
&= a_{i'} + \sum_{j \in N} b_{i'j} + \sum_{j \in \overline{C} \setminus \{i'\}} \sum_{k \in C} b_{jk} \\
&= R(i', C)
\end{aligned}
$$

The inequality above is from (2a). Also, the number of robots to sweep other nodes does not increase since for any $l \in \overline{C} \setminus \{i\}$ at a state $C' = C \cup \{i\}$ after applying τ_i

$$
\begin{aligned}
R(l, C') =& a_l + \sum_{j \in N} b_{lj} + \sum_{j \in \overline{C'} \setminus \{l\}} \sum_{k \in C'} b_{jk} \\
=& a_l + \sum_{j \in N} b_{lj} + \sum_{j \in \overline{C} \setminus \{l\}} \sum_{k \in C} b_{jk} + \sum_{j \in \overline{C'} \setminus \{l\}} b_{ji} - \sum_{k \in C} b_{ik} \\
\leq& a_l + \sum_{j \in N} b_{lj} + \sum_{j \in \overline{C} \setminus \{l\}} \sum_{k \in C} b_{jk} + \sum_{j \in \overline{C} \setminus \{l\}} b_{ji} - \sum_{k \in C} b_{ik} \\
\leq& a_l + \sum_{j \in N} b_{lj} + \sum_{j \in \overline{C} \setminus \{l\}} \sum_{k \in C} b_{jk} \\
=& R(l, C)
\end{aligned}
$$

The first inequality is because $C' \supset C$, and the second inequality is due to (2b). ◀

Proof for Proposition 15. Suppose there exists a dominated sequence of the form $\langle \tau_{i'}, \sigma \rangle$. We can always construct a dominating sequence $\langle \tau_i, \tau_{i'}, \sigma' \rangle$, where the first occurrence of τ_i in σ is removed to form σ'. If the dominated sequence is feasible, then the dominating sequence must also be feasible because the due date $d_{i',r_{i'}}$ is less than $\min(d_{i,r_i}, q)$. Postponing the production of the next item of type i and removing τ_i from σ does not cause the production periods of any items to be pushed earlier.

Next, we prove that the total cost of the dominating sequence is less than that of the dominated sequence. We first analyze the difference in changeover costs. In the dominated sequence, an item of type i' is produced, followed by an item of type t, incurring a changeover cost of $c_{i',t}$. If we insert the production of an item of type i between them, the changeover cost becomes $c_{i',i} + c_{i,t}$. The difference is $c_{i',i} + c_{i,t} - c_{i',t}$. Since the changeover costs satisfy the triangle inequality, removing the first τ_i from σ does not increase the changeover costs.

As for the stocking costs, the only difference lies in the cost of the next item of type i. In the dominated sequence, the item is not produced at period $\min(d_{i,r_i}, q)$ or $d_{i',r_{i'}}$. Thus, its production period must be postponed by at least two periods, increasing the total cost by $c_{i',i} + c_{i,t} - c_{i',t} - 2s_i \leq 0$. ◀

B Additional Problem Descriptions

B.1 One Machine Scheduling Minimizing Total Weighted Tardiness

We consider one machine scheduling for a set of jobs N, where each job $i \in N$ has the processing time p_i, the release dates r_i, the deadline d_i, and the weight w_i, all of which are nonnegative. The objective is to schedule all jobs while minimizing the sum of weighted tardiness, i.e. the different between d_i and the completion time of job i.

We formulate a DyPDL model where one job is scheduled at each step. Let F be a set variable representing the set of scheduled jobs, and t be the current time, which are initially an empty set and 0 respectively. The current time t is an integer resource variable where less is preferred. The completion of a job i when it is scheduled after time t is $C(t,i) = \max(r_i, t)+p_i$, and the tardiness is represented as a numeric expression $T(t,i) = \max(0, C_i(t) - d_i)$. The optimal value of a state $S = (F,t)$ is computed as:

$$V(S) = 0, \qquad \text{if } F = N$$
$$V(S) = \min_{i \in \overline{F}} T(t,i) + V(F \setminus \{i\}, C_i(t)) \qquad \text{else}$$

We implement two dominance rules proposed by Akturk and Ozdemir [1].

▶ **Proposition 16.** *Suppose $i, i' \in \overline{F}$ for a state in the DyPDL model of the $1|r_i|\sum w_i T_i$ problem. If $\tau_i, \tau_{i'} \in \mathcal{T}(S)$ satisfy (1) $p_i \geq p_{i'}$, (2) $d_i \leq d_{i'}$, (3) $w_i \geq w_{i'}$, and (4) $C(t,i) \leq C(t,i')$ at a state S, then $cost_{\tau_i}(V(S[\![\tau_i]\!]), S) \leq cost_{\tau_{i'}}(V(S[\![\tau_{i'}]\!]), S)$.* ⌟

▶ **Proposition 17.** *Suppose $i \in \overline{F}$ for a state in the DyPDL model of the $1|r_i|\sum w_i T_i$ problem. If $r_i \leq t$, and for all $i' \in \overline{F}$, τ_i satisfies (1) $p_i \leq p_{i'}$, (2) $d_i \leq d_{i'}$, (3) $w_i \geq w_{i'}$ at a state S, then $cost_{\tau_i}(V(S[\![\tau_i]\!]), S) \leq cost_{\tau_{i'}}(V(S[\![\tau_{i'}]\!]), S)$ for all transitions $\tau_{i'} \neq \tau_i$.* ⌟

Note that the second dominance rule can be implemented using forced transitions in DyPDL, a transition such that all other transitions are not applicable when it is applicable, a special case of transition dominance.

B.2 Talent Scheduling Problem

The talent scheduling problem [3] is to find a sequence of scenes to shoot to minimize the total cost of a film. In this problem, a set of actors A and a set of scenes N are given. In a scene $s \in N$, a set of actors $A_s \subseteq A$ plays for d_s days. For convenience, let $A(S)$ denote the set of all actors in all scenes $s \in S$, i.e. $A(S) = \cup_{s \in S} A_s$. An actor a incurs the cost c_a for each day they are on location. If an actor plays on days i and j, they are on location on days $i, i+1, ..., j$ even if they do not play on day $i+1$ to $j-1$. The objective is to find a sequence of scenes such that the total cost is minimized.

We use the double-ended search model proposed by Garcia de la Banda et al. [7] and implement the dominance proposed by Qin et al. [25]. Let B and E be two set variables representing the scenes at the beginning and at the end of the schedule, which are empty sets initially, and $R = N \setminus (B \cup E)$ be the set of remaining scenes. At each step, a scene s to shoot is selected from R, and τ_s append s to B. There are two types of actors:

- **Type 1**: If a is neither in $A(B) \cap A(E)$ nor in $A(s)$ but is still present on location during the days of shooting scene s. In other words, $a \notin A(B) \cap A(E)$, $a \notin A(s)$, and $a \in A(B) \cap A(R \setminus \{s\})$. Actor a must be paid for the shooting days of scene s.
- **Type 2**: If a is not included in $A(B) \cap A(E)$ but is included in $A(E)$, and scene s is their first involved scene, then actor a must be paid for all shooting days for scenes in $R \setminus \{s\}$.

Let $\mathrm{T}_1(s,B,E) = (A_s \cup (A(B) \cap A(R \setminus \{s\}))) \setminus (A(B) \cap A(E))$ and $\mathrm{T}_2(s,B,E) = (A_s \cap A(E)) \setminus A(B)$. Therefore, The cost per day to shoot s is

$$c(s,B,E) = d_s \times \sum_{a \in \mathrm{T}_1(s,B,E)} c_a + \sum_{s' \in R \setminus \{s\}} d_{s'} \times \sum_{a \in \mathrm{T}_2(s,B,E)} c_a$$

Overall, we have the following DyPDL model.

$$V(B,E) = \begin{cases} 0 & \text{if } B \cup E = N \\ c(s,B,E) + V(E, B \cup \{s\}) & \text{else} \end{cases}$$

We implement the dual bound where the remaining cost must be at least the total cost of actors times the shooting days they must present, i.e.

$$\eta(S) = \sum_{a \in A(R)} c_a \times \sum_{s \in \{s' | a \in A_{s'}\}} d_s$$

We follow Qin et al. [25] to implement the following dominance rule as transition dominance.

▶ **Proposition 18.** *Suppose two scenes $s, s' \in R$ are two unscheduled scenes. Let $o(B, s) = A(B) \cap \overline{A(B)} \cup A_s$ and $o(E, s) = A(E) \cap \overline{A(E)} \cup A_s$. If τ_s and $\tau_{s'}$ satisfy*

- *$o(B, s) \supseteq o(B, s') \wedge o(E, s) \subset o(E, s')$, or*
- *$o(B, s) \subset o(B, s') \wedge o(E, s) \subseteq o(E, s')$,*

then $cost_{\tau_s}(V(S[\![\tau_s]\!]), S) \leq cost_{\tau_{s'}}(V(S[\![\tau_{s'}]\!]), S)$ at the current state.

B.3 Orienteering Problem with Time Window

The OPTW problem [11] asks for a schedule to visit a set of customers $N = \{1, \ldots, n-1\}$ starting from the depot 0. Visiting customer j from i incurs travel time $c_{i,j} > 0$ while producing the profit $p_i \geq 0$. Each customer i has a service window $[a_i, b_i]$ and can be visited only within the window. The vehicle needs to wait until a_i upon earlier arrival. The objective is to maximize the total profit while returning to the depot before the deadline b_0.

The DyPDL model we implement is similar to the DP model by Righini and Salani [27]. The model uses a set variable U to represent the set of customers to visit, an element variable loc to represent the current location, and a numeric resource variable t to represent the current time. We visit customers one by one using transitions. Customer j can be visited next if it can be visited and the depot can be reached by the deadline after visiting j. Let $c^*_{i,j}$ be the shortest travel time from i to j. Then, the set of customers that can be visited next is $X(U, loc, t) = \{j \in U \mid t + c_{loc,j} \leq b_j \wedge t + c_{loc,j} + c^*_{j,0} \leq b_0\}$. The optimal value of a state can be computed as follows:

$$\begin{aligned}
V(S) &= 0, && \text{if } U = \emptyset \\
V(S) &= -\infty && \text{else if } t + c_{loc,0} > b_0 \\
V(S) &= \max_{j \in X(U,loc,t)} p_j + V(S[\![\tau_j]\!]) && \text{else}
\end{aligned}$$

In the actual implementation, we also add additional forced transitions to remove nodes from U that cannot be reached without violating the time limit b_0.

Transition dominance in this problem is similar to that of ALP: if customers i and i' can be visited consecutively without reaching $a_{i'}$ starting from the current position and the current time, then taking $\tau_{i'}$ must not be optimal.

▶ **Proposition 19.** *Suppose the travel times satisfy the triangle inequality, and customers $i, i' \in U$ are unvisited. If $\tau_i, \tau_{i'}$ satisfy that visiting i and i' consecutively does not reach the start time of i', i.e., $\max(a_i, t + c_{loc,i}) + c_{i,i'} < a_{i'}$, then $cost_{\tau_i}(V(S[\![\tau_i]\!]), S) \leq cost_{\tau_{i'}}(V(S[\![\tau_{i'}]\!]), S)$.* ⌟

Proof. By definition of $X(U, loc, t)$ we can infer that $X(U, loc, t_1) \subseteq X(U, loc, t_2)$ if $t_1 \leq t_2$. Let the current state be $S = (U, loc, t)$. We show that for any S-solution $\langle \tau_{i'}; \sigma \rangle$ starting with $\tau_{i'}$, we can construct a dominating S-solution starting with τ_i then $\tau_{i'}$ which has at least the same profits.

Suppose that σ does not contain $\tau_{i'}$, we can construct an S-solution $\langle \tau_i, \tau_{i'}; \sigma \rangle$. The state after applying transition $\tau_{i'}$ directly and after applying transitions τ_i and $\tau_{i'}$ are:

$$S_1 = (U \setminus \{i'\}, i', \max(a_{i'}, t + c_{loc,i'}))$$
$$S_2 = (U \setminus \{i, i'\}, i', \max(a_{i'}, \max(a_i, t + c_{loc,i}) + c_{i,i'}) = (U \setminus \{i, i'\}, i', a_{i'})$$

respectively. The simplification of S_2 is due to the condition in the proposition.

Consider the first transition τ_j in σ. Since $j \neq i$, $j \in X(U \setminus \{i'\}, \mathrm{loc}, t)$ implies $j \in X(U \setminus \{i, i'\}, \mathrm{loc}, t)$. After applying transition τ_j to S_1 and S_2, the current time becomes $\max(a_j, \max(a_{i'}, t + c_{\mathrm{loc},i'}) + c_{i',j})$ and $\max(a_j, a_{i'} + c_{i',j})$, respectively, with the latter term still being less than or equal to the former. Inductively, if σ is an S_1-solution, then it must also be a feasible S_2-solution with the same profit. According to the Bellman equation, the S-solution $\langle \tau_{i'}; \sigma \rangle$ has less profit than $\langle \tau_i, \tau_{i'}; \sigma \rangle$ due to the additional transition τ_i.

Now, suppose the S_1-solution does visit customer i, and let σ' be the sequence of transitions obtained by removing τ_i from σ. We claim that σ' is a feasible S_2-solution. The arguments for the applicability of any transition τ_j before τ_i in σ are similar to those above. Skipping τ_i in the solution does not increase the current time due to the triangle inequality of travel times. After transition τ_i, the set of unvisited customers becomes the same. Therefore, if σ is a feasible S_1-solution, then σ' is a feasible S_2-solution. The difference in the objective between σ and σ' is p_i, and the solutions $\langle \tau_{i'}; \sigma \rangle$ and $\langle \tau_i, \tau_{i'}; \sigma \rangle$ have the same objective. ◀

B.4 Travelling Salesman Problem with Time Windows

In the travelling salesperson problem with time windows and makespan objective [19], a set of customers $N = \{0, ..., n-1\}$ is given. A solution is a tour starting from the depot (index 0), visiting each customer exactly once, and returning to the depot. Visiting customer j from i incurs the travel time $c_{i,j} > 0$. In the beginning, $t = 0$. The visit to customer i must be within a time window $[a_i, b_i]$. Upon earlier arrival, waiting until a_i is required. The objective we consider is to minimize the total makespan where the cost of visiting customer j from the current location i with time t is $\max\{c_{i,j}, a_j - t\}$. Let $c^*_{i,j}$ be the shortest travel time from i to j. Similar to OPTW, the model uses a set variable U represents the set of customers to visit, an element variable loc represents the current location, and a numeric resource variable t represents the current time. We visit customers one by one using transitions. For simplicity, let $X(U, loc, t) = \{j \mid t + c^*_{loc,j} \leq b_j\}$ and $d(t, loc, j) = \max\{c_{loc,j}, a_j - t\}$.

The optimal value of a state S can be computed as follows:

$$V(S) = c_{loc,0}, \qquad \text{if } U = \emptyset$$
$$V(S) = \infty \qquad \text{else if } \exists j \in U, t + c^*_{ij} > b_j$$
$$V(S) = \max_{j \in X(U,i,t)} d(t, i, j) \quad + V(S[\![\tau_j]\!]) \qquad \text{otherwise}$$

▶ **Proposition 20.** *Suppose the travel times satisfy the triangle inequality, and customers $i, i' \in U$ are unvisited. If visiting i and i' consecutively does not reach the start time of i', i.e. $\max(a_i, t + d(t, loc, i)) + c_{i,i'} < a_{i'}$, then $cost_{\tau_i}(V(S[\![\tau_i]\!]), S) \leq cost_{\tau_{i'}}(V(S[\![\tau_{i'}]\!]), S)$.* ⌟

The proof is similar to that of OPTW.

C Experiment Instances

- **Graph-Clear**: We use 135 instances generated by Kuroiwa and Beck [15], where each graph consists of 20, 30, or 40 nodes.

- **OPTW**: We use 144 instances from Righini and Salani [26], Montemanni and Gambardella [21], and Vansteenwegen et al. [29]. The original instances are defined on a geometric plane. However, rounding distances between locations in the literature may lead to violations of the triangle inequality. To correct this, we update the distance between locations i and j to $d_{ik} + d_{kj}$ whenever there exists a location k such that $d_{ij} > d_{ik} + d_{kj}$.
- **TSPTW-M**: For TSPTW, we use 290 instances from Dumas et al. [8], Gendreau et al. [9], and Ohlmann and Thomas [23]. Similar to OPTW, rounding integer travel times may result in violations of the triangle inequality. We apply the same correction to ensure that the inequality holds for all travel times.

Modeling and Explaining an Industrial Workforce Allocation and Scheduling Problem

Ignace Bleukx
Department of Computer Science, KU Leuven, Belgium

Ryma Boumazouza
Airbus SAS, Institut de Recherche Technologique Saint-Exupery, Toulouse, France

Tias Guns
Department of Computer Science, KU Leuven, Belgium

Nadine Laage
Airbus Aerostructures GmbH, Hamburg, Germany

Guillaume Poveda
Airbus SAS, Toulouse, France

Abstract

We present an industrial case on workforce allocation and scheduling in the aircraft manufacturing industry, where available teams need to be assigned to logistical operations. This application presents several challenges such as the scale of the problem, the need for fair workload distribution, and the need for methods for mitigating unforeseen disruptions due to technical malfunctions or incompatible weather conditions. We compare different Constraint Programming (CP) models for the allocation and scheduling problems, with extra focus on modeling the workload balancing component. Additionally, we investigate different techniques for explaining infeasibility of a disrupted schedule, such as conflict computation using Minimal Unsatisfiable Subsets (MUSes) and feasibility restoration using Minimal Correction Subsets (MCSes) or constraint relaxations. Our experimental results show that by using appropriate modeling techniques, the problem can be solved in reasonable time, thereby producing fair schedules. Additionally, we show how invalidated schedules can be explained and restored efficiently to help human operators in solving disruptions to the schedule.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling; Theory of computation → Constraint and logic programming; Human-centered computing → User centered design

Keywords and phrases modeling, scheduling, fairness, explanations, feasibility restoration

Digital Object Identifier 10.4230/LIPIcs.CP.2025.6

Supplementary Material *Model (Source Code and Data)*: `https://github.com/ML-KULeuven/Explainable-Workforce-Scheduling`
Software (Source Code and Data): `https://github.com/airbus/discrete-optimization`

Funding This work was partially supported by the Europe Research and Innovation program TUPLES (101070149). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

1 Introduction

The assignment of workers to different tasks is a recurring challenge in many industries. Effective allocation of workers directly impacts key business metrics such as productivity, cost management, and overall organizational performance. Solving an industrial workforce allocation problem requires considering several factors such as such as employee availability, required qualifications of workers for a task, regulatory requirements, and task priorities.

© Ignace Bleukx, Ryma Boumazouza, Tias Guns, Nadine Laage, and Guillaume Poveda;
licensed under Creative Commons License CC-BY 4.0

In this paper, we focus on a workforce-allocation problem in the aircraft manufacturing industry, solving instances involving hundreds of tasks on a daily basis. Each of these tasks consists of a logistical operation such as transporting parts from one place to another or loading and unloading cargo. Transportation tasks are executed using trucks, trailers, or specialized transportation platforms, which require trained workers. Timely execution of these tasks is critical for the production and assembly of aircraft. Currently, given an order book of tasks to be executed by a given deadline, a team of operational planners manually schedules and allocates the tasks to available workers. To alleviate some of the workload of this planning task, we investigate (partially) automating the planning and allocation of tasks by formulating the allocation and scheduling problem as a CP model.

Furthermore, we aim to offer decision support in case of disrupted schedules due to unforeseen circumstances. Examples of such disruptions can be the delay of a part delivery, human or logistical resource disruptions, adversarial weather conditions, etc. We investigate several methods for detecting and explaining the infeasibility of schedules, as well as methods for automatically resolving disruptions using techniques from feasibility restoration [30]. This allows the planners to 1) understand the implications of the disruption on the given schedule and 2) review alternatives provided by an automated system to restore the schedule to a feasible state while accounting for the disruption(s).

1.1 Use case description

We address the problem of workforce allocation for logistical tasks among worker teams at an aircraft manufacturing site. These logistical tasks are interleaved with production activities on assembly lines [4], and need to be aligned with timings of higher-level logistics involving deliveries of raw materials or components provided by external parties via trucks, ships, or cargo aircraft. These components then have to be transported to internal hangars at the right time, where they will be assembled with other aircraft parts. We incorporate these operational requirements into the allocation and scheduling problem as part of the input data. More specifically, each task has a predefined release time, duration, and deadline, and certain tasks follow precedence constraints, such as when two parts must be delivered in sequence before assembly. Apart from explicit precedence constraints, there are no requirements on the task ordering, as the duration of any task includes returning to the home base of the manufacturing plant. As a result, transition time between the end location of one task, and the starting location of the next is included in the duration of each task. This differs from the traditional definition of a workforce-scheduling and routing problem (WSRP) [5] and allows to formulate the problem as a pure scheduling problem instead. In our setting, the goal is to find an allocation of tasks to working teams, such that the total number of required teams is minimized. Additionally, the workload should be balanced fairly among the teams [12]. That is, the allocation of tasks should be done such that all teams have a similar amount of work.

Currently, transportation tasks are manually allocated and scheduled upfront by a team of planners. These planners also coordinate the fine-grained synchronization among teams, to ensure the precedences of tasks are satisfied. During the working day, they continuously monitor real-time updates and adjust schedules as needed to handle unforeseen events or disruptions. These disruptions can arise for multiple reasons, including:

- **Weather-related disruptions:** Extreme wind or rain prevents the use of some hangars.
- **External logistics disruptions:** Delays or cancellations from third-party suppliers impact the arrival time of essential parts.
- **Human resource disruptions:** Unexpected worker absences, team meetings or fire-drills in a hangar temporarily reduces workforce availability.

- **Material availability disruptions:** Vehicles like trucks or trailers may become temporarily unavailable due to technical breakdowns, accidents or maintenance operations.

Large disruptions can impact the predefined plan in such a way that the set of disrupted tasks cannot be reallocated to other teams. This requires the planners to revise the plan, which may involve multiple re-allocation and/or re-scheduling operations. When disruption occurs, given than individual activity execution times range from 15 to 60 minutes, planners have a limited window to adjust schedules. As this is tedious and time-consuming to do by hand, planners often drop disrupted tasks from the current planning horizon and move it to a next one. In order to help the planners revise the allocation in a more efficient way, we propose to automatically repair disrupted schedules and show several alternatives to the planners. This allows them to further adapt or merge proposed alternatives based on their experience or other tacit knowledge.

2 Background

A Constraint Satisfaction Problem (CSP) is defined as triple $CSP(\mathcal{X}, \mathcal{D}, \mathcal{C})$[27]. Here, $\mathcal{X}$ is set of discrete variables with each an associated set of values (i.e., a domain) in $\mathcal{D}$. The domain of variable x is written as $\mathcal{D}[x]$. A variable can be finite-domain integer ($\mathcal{D}[x] \subset \mathbb{Z}$) or Boolean ($\mathcal{D}[x] = \{True, False\}$). An assignment α maps each variable x to a value in its domain $\mathcal{D}[x]$. $\mathcal{C}$ is a set of constraints, where each constraint maps an assignment α to True or False. An assignment that is mapped to True by all constraints in the CSP is called a *solution*. We write $\alpha(x)$ for the value of the variable x in solution α. A CSP where no α satisfies all constraints is *unsatisfiable* or *UNSAT* for short.

A Constraint Optimization Problem (COP) extends the definition of a CSP with an objective function f that maps solutions to a numerical value. A solution α is an *optimal* solution if no other solution α' has a better objective value than α.

A Multi-Objective Constraint Optimization Problem further extents the definition of a COP by allowing multiple objective functions to be optimized simultaneously [28, 7, 8]. In case there is a strict priority on the objectives, we refer to it as a Lexicographic Multi-objective Constraint Optimization Problem (LMOCOP). Such problems can be solved by decomposing the LMOCOP into individual COPs and solving these in sequence, and where the objective value of previous COPs is fixed. Example 1 illustrates this process, which can be implemented efficiently with incremental (CP-)solvers.

▶ **Example 1** (Solving a LMOCOP). Suppose we have a lexicographic optimization problem with objectives $f_1 \prec f_2 \prec \ldots \prec f_n$. We can find an optimal solution by first optimizing $COP(\mathcal{X}, \mathcal{D}, \mathcal{C}, f_1)$ to find solution α_{f_1}. Then, we add constraint $f(\mathcal{X}) \leq f(\alpha_{f_1})$ and re-solve the problem with objective function f_2 to find α_{f_2}. This process is repeated until we find α_{f_n} where each objective f_i is optimal, given a value for objective f_{i-1}.

2.1 Scheduling-problems in CP

We consider problems involving scheduling and allocating sets of tasks. The CP-community has developed several *global constraints* for such problems, and in this paper we use the NoOverlap (also called Disjunctive) and Cumulative global constraints. These global constrains reason over uninterruptible tasks, which are represented as *intervals* using an Interval(s, d) variable with start s and duration d. The NoOverlap(I) constraint ensures intervals I are scheduled such that they do not overlap. Cumulative(I, D, c) is a generalization of this constraint where a set of intervals with a corresponding set of resource requirements D are scheduled in such as manner that the total capacity c of the resource is never exceeded [29, 32, 33].

An extension to interval variables was introduced by Laborie et al. [21], namely *optional* intervals. Such an interval is declared as OptInterval(s, d, p) where the Boolean variable p controls whether the interval (that is the task it represents) is present in the solution.

2.2 Explainable Constraint Solving

For explaining and resolving infeasible schedules due to disruptions, we will rely on techniques from explainable constraint solving. In particular, we compute Minimal Unsatisfiable Subsets (MUS) [23] to explain the user *why* the allocation problem is unsatisfiable after disruptions, and compute Minimal Correction Subsets (MCS) [24] to pinpoint the user to a set of constraints that should be relaxed or removed in order to resolve the conflicts in the problem.

▶ **Definition 2** (Minimal Unsatisfiable Subset [18]). *A Minimal Unsatisfiable Subset (MUS) is an unsatisfiable subset $U \subseteq \mathcal{C}$ for which it holds that each strict subset $U' \subsetneq U$ is satisfiable.*

Note that several MUSes may exist, and one may have a preference on which conflict to compute [20]. E.g., it is common to compute a *smallest* MUS [19], or *optimal* MUS [15].

▶ **Definition 3** (Minimal Correction Subset [24]). *A Minimal Correction Subset (MCS) is a subset $M \subseteq \mathcal{C}$ for which $\mathcal{C} \setminus M$ is satisfiable, and for any $M' \subsetneq M$, $\mathcal{C} \setminus M'$ is unsatisfiable.*

Similar to MUSes, a problem may contain multiple MCSes. To find an optimal MCS, we can reify each constraint and minimize the number of falsified indicator variables.

3 Formal specification of the constraint models

We now introduce the mathematical formulation of the workforce-allocation COP. Section 3.1 focuses on the pure-allocation setting where each task has a fixed start-and end-time. In Section 3.2, we investigate how this formulation can be adapted to a scheduling-and-allocation formulation where each of the tasks can be shifted within a given time-window.

3.1 Allocation

This problem consists of assigning each task to a compatible team. Teams can be (in)compatible with a task due to their working hours, or due to specific technical qualifications required to execute the task. No team can execute two tasks at once, and hence overlapping tasks should be assigned to different teams. Finally, some tasks are required to be assigned to the same team, e.g., when two tasks involve transporting the same part from one place to another.

Input data

- $\mathcal{T}$: a set of n tasks to assign to teams.
- $\mathcal{W}$: a set of m worker-teams to assign tasks to.
- $Comp \in \mathbb{B}^{n \times m}$ where $Comp_{t,w}$ indicates team w and task t are compatible.
- S: the fixed start times of each task, with $s_i \in S$ the start time of task i.
- D: the fixed durations of each task, with $d_i \in D$ the duration of task i.
- $\mathcal{G}$: set of task groups where tasks in group $g \subseteq \mathcal{T}$ should be executed by the same team.

Decision variables and constraints

The variables in this model consist of a $n \times m$ Boolean matrix where each variable $x_{t,w}$ indicates whether team w is assigned to task t.

Each task is allocated to a team:

$$\sum_{w \in \mathcal{W}} x_{t,w} = 1 \qquad \forall t \in \mathcal{T} \quad (1)$$

Tasks are only assigned to compatible teams:

$$\neg Comp_{t,w} \Rightarrow \neg x_{t,w} \qquad \forall t \in \mathcal{T}, \forall w \in \mathcal{W} \quad (2)$$

A given group of tasks g should be assigned to the same team:

$$\textsc{AllEqual}(\{x_{t,w} \mid t \in g\}) \qquad \forall g \in \mathcal{G}, \forall w \in \mathcal{W} \quad (3)$$

A team cannot be allocated to overlapping tasks:

$$\sum_{t' \in \mathcal{T}}^{n} x_{w,t'} \cdot (s_t \leq s_{t'} \wedge s_t + d_t > s_{t'}) \leq 1 \qquad \forall t \in \mathcal{T}, \forall w \in \mathcal{W} \quad (4)$$

Auxiliary variables and objectives

We introduce auxiliary variables $used_w \in \mathbb{B}^m$ that indicate whether team w is used, and integer variables $time_w \in \mathbb{N}^m$ that indicate the time worked by each team. These auxiliary variables relate to the decision variables x according to the following constraints:

A team is used if a task is allocated to it:

$$\bigvee_{t \in \mathcal{T}} x_{t,w} \Rightarrow used_w \qquad \forall w \in \mathcal{W} \quad (5)$$

Definition of the time worked by a team, only enforced if the team is used:

$$used_w \Rightarrow \sum_{t \in \mathcal{T}} x_{t,w} \cdot d_t = time_w \qquad \forall w \in \mathcal{W} \quad (6)$$

The objectives of the problem are to minimize the number of teams used: $\sum_{w \in \mathcal{W}} used_w$ and to minimize the time-dispersion as described next.

Modeling improvement: different formulations for time-dispersion

Tasks should be distributed *fairly* among the working teams. That is, we aim to minimize the *dispersion* of the workload which we define as: $max_{w \in \mathcal{W}}(time_w) - min_{w \in \mathcal{W}}(time_w)$ and corresponds to a *spread-based* fairness measure from the literature [6, 12]. However, we noticed that combinatorial solvers are often stuck on proving optimality when using this objective. Therefore, in Sec. 5.1, we compare the following formulations of the time-dispersion:

- `max_minus_min`: exact formulation: $\max_{w \in \mathcal{W}}(time_w) - \min_{w \in \mathcal{W}}(time_w)$
- `max_diff`: exact formulation, with pairwise differences $\max_{w,w' \in \mathcal{W}}(time_w - time_{w'})$
- `proxy_max_min`: maximize the minimum time worked: $-\min_{w \in \mathcal{W}}(time_w)$
- `proxy_min_max`: minimize the maximum time worked: $\max_{w \in \mathcal{W}}(time_w)$
- `proxy_sum`: minimize the sum of absolute differences $\sum_{w \in \mathcal{W}, w' \in \mathcal{W}}(|time_w - time_{w'}|)$

Modeling improvement: symmetry breaking over equivalent teams

Although each team has a specific set of qualifications that result in (in)compatibility with certain tasks, some teams are still equivalent for the given time-horizon of the schedule. Therefore, we can detect and break symmetries between teams. We do this by detecting sets of teams that can be assigned to the exact same set of tasks in the input data. For every consisting set of equivalent teams $\{t_1, t_2, \ldots, t_p\}$, we add the following constraint:

$$used_{t_1} \leq used_{t_2} \leq used_{t_3} \leq \cdots \leq used_{t_p}$$

Comparison to literature

The allocation problem described in this paper is related to the "list coloring" problem, a variant of the well-known graph-coloring problem where each node can only be assigned a subset of colors. However, in addition to minimizing the number of teams (colors) used, we also aim to minimize the time dispersion. This is specific to our use case and, to the best of our knowledge, it is not considered in the literature of similar list-coloring problems. Constraint (4) would implement a "clique" constraint on a list coloring problem. In our case, these cliques can easily be detected due to the structure of the problem.

3.2 Scheduling

In the second variant of the problem, the tasks are not yet fixed in time and should be scheduled in a predefined time-window. That is, each task has a earliest time it can be started (the release time), and a deadline by which it should be completed. Moreover, some tasks depend on one another and therefore share a precedence constraint. Also, each of the worker teams has fixed working-hours and hence, any task assigned to a given team, must start and finish within the working hours of that team.

Additional data

- $\mathcal{R}$: the release time of each task, with $r_t \in \mathcal{R}$ the release time of task t.
- $\mathcal{E}$: the deadline of each task, with $e_t \in \mathcal{E}$ the deadline of task t.
- *Prec*: a set of tuples (t, t') where task t should finish before the start of t'.
- Cal_w: a set of tuples (c_s, c_d) that represent intervals during which team w is off-duty.

Additional variables and constraints

Our CP-model for this problem is inspired by flexible-jobshop problems [1, 9], and the main idea of this model is to duplicate each task for each team using an optional interval variable. Each interval is defined using a new variable $s_t \in \mathbb{N}$ for representing the start time of each task, and we use the Boolean matrix x to control which intervals are indeed present in the schedule, i.e., which intervals (tasks) are allocated to what team.

The constraint model builds further on the "pure allocation" formulation above. That is, we re-use constraints (1), (2) and (3), but replace constraint (4) with constraint (7), which ensures that no tasks assigned to the same team overlap. Additionally, constraint (7) ensures no tasks are assigned during the "off-hours" of a team as defined by their calendar. Finally, we add constraints (8) and (9) to enforce a legal schedule of the tasks.

Teams cannot perform overlapping tasks and can only work when available.

$$\textsc{NoOverlap}(\{\textsc{OptInterval}(s_t, d_t, x_{t,w}) \mid t \in \mathcal{T}\} \cup \{\textsc{Interval}(c_s, c_d) \mid (c_s, c_d) \in Cal_w\}) \quad \forall w \in \mathcal{W} \ (7)$$

Tasks are scheduled within their allowed time time-frame:

$$s_t \geq r_t \wedge s_t + d_t \leq e_t \quad \forall t \in \mathcal{T} \ (8)$$

Enforce precedences between tasks:

$$s_t + d_t \leq s_{t'} \quad \forall (t, t') \in Prec \ (9)$$

Typically, the objective in flexible job-shop scheduling is to minimize the makespan of the schedule. However, in this problem, we re-use the objectives from Section 3.1 and aim to minimize the number of teams used and to minimize the dispersion.

Modeling improvement: computing a lower-bound on the number of required teams

The constraint model above can be solved for large instances by state-of-the art constraint solvers. However, we noticed that when minimizing the number of required teams, the solver is often stuck on proving optimality of the solution. We want to help the solver by enforcing a reasonable lower-bound on the number of teams. To compute this lower-bound, we propose the following relaxation of the problem, where the pool of working teams is modeled as a cumulative resource:

$$\textsc{Cumulative}(\{\textsc{Interval}(s_t, d_t) \mid t \in \mathcal{T}\}, c)$$

In this relaxation, each team is treated as equivalent, and hence the relaxation does not consider task-compatibility requirements, nor the availabilities of each team. As we will see in Section 5.2, computing a solution that minimizes the capacity c, is often much faster than solving the full problem. We add the lower-bound on c found by the solver to the full-problem as a constraint, which can help the solver prove optimality faster.

Comparison to literature

The scheduling problem considered in this paper is most closely related to the Flexible-Jobshop scheduling problem (FJSP), where a set of tasks is to be scheduled and assigned to a set of resources (e.g., machines). However, compared to the flexible-jobshop problem, we also consider the constraint where some tasks have to be assigned to the same team (resource in FJSP). Moreover, in jobshop-scheduling, it is usually the goal to minimize the makespan; this is not the case for our use case. We consider as primary objective to minimize the number of resources (teams) required to complete the schedule. Additionally, we ensure a fair distribution using the time-dispersion objective. While load balancing is considered in FJSP literature, it is more common to minimize the maximum workload of a machine given the time-horizon as not to wear out any machines more than others. This is different from our objective and corresponds to the `proxy_min_max` formulation from Section 3.1.

4 Interactive and explainable allocation

In this section, we outline our efforts to develop an explainable decision-making tool for allocation of tasks to teams, with the potential for generalization to similar use cases. Our primary application of explanation techniques will be to handle disruption scenarios, which invalidate the precomputed schedule and require the planner to re-allocate and/or re-schedule the set of tasks. Section 4.1 outlines several types of disruptions and Section 4.2 investigates what types of explanation techniques can be useful for the planners.

4.1 Disruption scenarios

From the planner's perspective, the scenario is as follows: an initial schedule and allocation is established for upcoming shifts, with an estimate of available teams. This schedule can be computed using the CP-models from the previous section or manually constructed by the planners. As discussed in Section 1.1, various disruptions can affect the plan and possibly make the foreseen allocation infeasible. In response, the planner must first understand the implications of the disruption, and then adapt and reallocate the available resources such that as many tasks as possible can still be allocated, without wasting resources.

To evaluate our explanation methods in a broad range of disruption scenarios, we use a generator with adjustable parameters for simulating disruptions. These generator parameters are designed in coordination with the real planners, to cover a range of realistic and complex scenarios that arise in their daily practice. Table 1 details the parameters used for generating resource disruption scenarios based on three key factors: the number of disruptions, the number of affected teams, and the duration of the disruptions. As an illustration, the `all teams affected` scenario corresponds to a situation where all working teams are unavailable for a given amount of time, which can occur during extreme weather conditions. In less extreme cases, such as the `few disruptions` scenario, some teams are unavailable for a period of time caused by team meetings, transport accidents, or unexpected worker absences. A single disruption can affect multiple teams when they need to share a certain resource in the manufacturing site, e.g., when a hangar has to be evacuated due to a fire-drill. Figure 1 visualizes two instances of disruptions.

Table 1 Parameters for generating disruptions of each scenario type.

Scenario description	No. of disruptions	No. of teams affected	Duration (Minutes)
`few disruptions`	$\{1 \dots 3\}$	$\{1 \dots \frac{\lvert\mathcal{W}\rvert}{2}\}$	$\{15, 30, 60, 120\}$
`long disruptions`	$\{1 \dots 3\}$	1	$\{120, 240, 360\}$
`many disruptions`	$\{5 \dots 10\}$	$\{1 \dots \lvert\mathcal{W}\rvert\}$	$\{15, 30, 60, 120\}$
`many teams affected`	1	$\{\frac{\lvert\mathcal{W}\rvert}{2} \dots \lvert\mathcal{W}\rvert\}$	$\{15, 30, 60, 120\}$
`all teams affected`	1	$\lvert\mathcal{W}\rvert$	$\{15, 30, 60, 120\}$
`one team dropped`	1	1	Horizon
`two teams dropped`	1	2	Horizon
`delay`	-	-	$Poisson(\lambda = 5)$

After a disruption in the schedule has occurred, the planners will try to reallocate tasks to other, free teams. However, in practice, this is often not possible as task-compatible teams are usually already occupied with other tasks. In the next two sections, we explore how to assist the planner in resolving these disruptions.

▶ **Example 4** (Example disruptions to a schedule). Figure 1 illustrates two classes of disruption scenarios. Disruptions are shown as striped boxes, and the affected tasks are colored black.

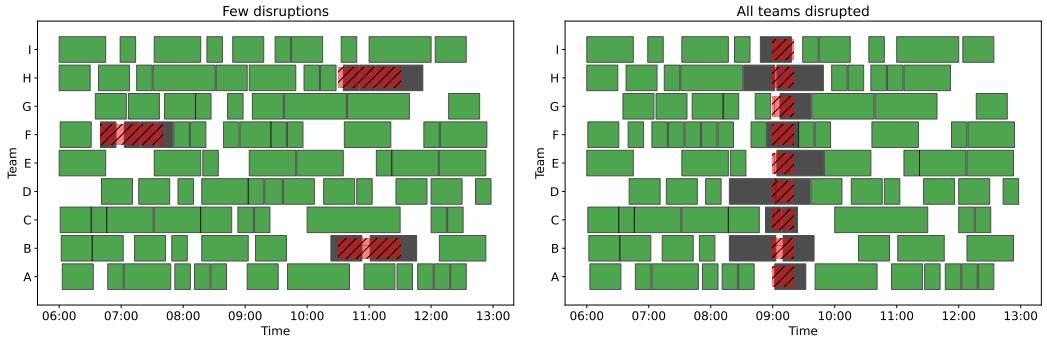


Figure 1 Examples of different disruption scenarios on the same base instance of 1 working shift.

4.2 Conflict detection: explaining why the allocation is unsatisfiable

In this first setting, we aim to explain to the planner *why* it was not possible to reallocate all the tasks affected by the disruption. That is, we compute a minimal *conflict* in the constraints by using a Minimal Unsatisfiable Subset (MUS)-extraction algorithm. As is typical in MUS-extraction, we split the problem constraints into a set of *foreground* and *background* constraints [18]. Foreground constraints are those that are explicitly presented as part of the conflict while background constraints are assumed to always hold. In our setting, we treat constraints that require tasks to be allocated (1), constraints that require tasks to be allocated to the same team (3) and no-overlap constraints (4) as foreground constraints. The remaining constraints are background, in combination with an encoding of the disruptions. As a result, a MUS corresponds to a minimal set of overlapping tasks that cannot be executed given the available teams. Section 6.1 experimentally compares the Shrink and SMUS algorithms for computing MUSes for this setting. SMUS computes the smallest MUS in terms of the number of constraints [19], while Shrink computes an arbitrary MUS.

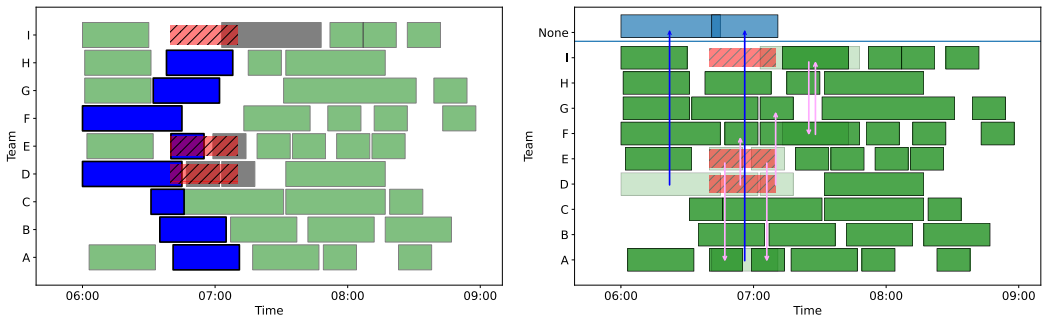
Figure 2a visualizes a set of *task is allocated (3)* constraints that together form a MUS. Indeed, this irreducible set of tasks cannot be allocated given available teams and disruptions.

4.3 Feasibility restoration: explaining how to revise the model

After understanding the conflict in the allocation problem, the next step is to revise the model such that a new schedule can be computed. Model diagnosis [26] can be used to iteratively resolve all conflicts in the problem by sequentially resolving each individual MUS. Indeed, a conflict represented by a MUS is resolved after the removal of one of its constraints. However, by solving each conflict locally, in the end, more constraints may be removed from the problem than strictly necessary. Therefore, we investigate two types of feasibility restoration methods that can globally resolve all conflicts in the model.

4.3.1 Removing constraints with Minimal Correction Subsets

In this first approach to resolving disruptions, we compute a Minimal Correction Subset (MCS). Such a correction subset presents the planner with a minimal set of constraints that should be removed, so that the resulting model will admit a solution. Similar to computing



(a) Visualization of a MUS.

(b) Visualization of a MCS.

Figure 2 Visualization of two explanation methods for the same working shift and disruption.

a MUS, the model is split into foreground and background constraints. Here, the foreground constraints are those that can be removed, while the background constraints are always enforced. In this setting, we only allow to remove tasks from the schedule (1) or remove *same-allocation* constraints (3). All other constraints are considered background. Figure 2b visualizes an MCS and the resulting solution where two tasks (in blue) can be dropped from the schedule in order to resolve the disruption. Section 6.1 experimentally investigates two types of MCS-computation algorithms: we explore a greedy GROW-based algorithm to find an arbitrary MCS, and a reified MAX-CSP formulation for finding an optimal MCS.

4.3.2 Rescheduling using multi-objective feasibility restoration

Using MCS-based feasibility restoration can cause drastic changes to the model [3]. Indeed, as constraints (or tasks) are removed from the model in order to restore feasibility, the resulting solution may drastically change compared to the initial assignment. Moreover, a disruption may only overlap with a task for a short period of time and, instead of removing the task, it could be slightly shifted to a later time. In this section, we investigate such fine-grained alterations to the schedule, which are not possible by computing MCSes on the infeasible allocation model. That is, we convert the original allocation problem to a allocation-and scheduling-formulation as outlined in Section 3.2. Then, based on the original assignment α, which is now infeasible, we define several *similarity* objectives. These objectives serve to ensure the newly computed solution is as close as possible to the original assignment [10, 14, 30]. We define 5 such similarity objectives:

Maximize the amount of tasks that are still allocated to a team (`#done`):

$$\textit{maximize} \quad (\sum_{t\in\mathcal{T}}(\sum_{w\in\mathcal{W}} x_{t,w})) = (\sum_{w\in\mathcal{W}} \alpha(x_{t,w})) \tag{10}$$

Minimize the tasks that are re-allocated to a different team (`#realloc`):

$$\textit{minimize} \quad \sum_{w\in\mathcal{W}}\sum_{t\in\mathcal{T}}(\alpha(x_{t,w}) \neq x_{t,w}) \tag{11}$$

Minimize the number of shifted tasks (`#shift`).

$$\textit{minimize} \quad \sum_{t\in\mathcal{T}}(\alpha(s_t) \neq s_t) \tag{12}$$

Minimize the total shift in time of the tasks (`sum-shift`)

$$\textit{minimize} \quad \sum_{t \in \mathcal{T}} |\alpha(s_t) - s_t| \tag{13}$$

Minimize the maximum shift in time of the tasks (`max-shift`)

$$\textit{minimize} \quad \max_{t \in \mathcal{T}} |\alpha(s_t) - s_t| \tag{14}$$

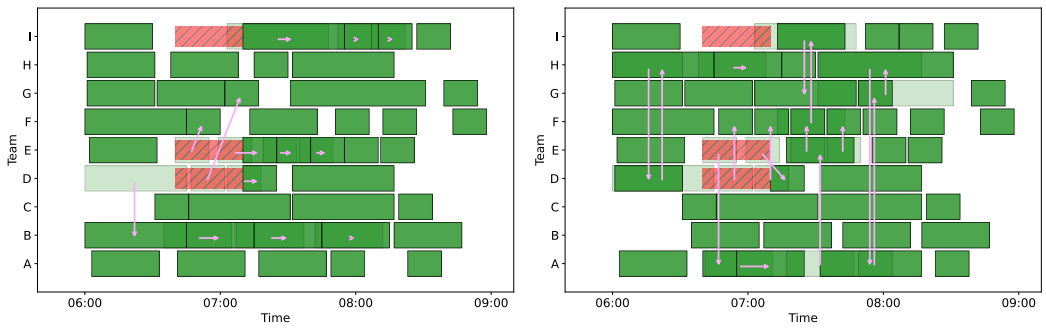
Instead of optimizing only one of these similarity objectives, we propose to combine these into a LMOCOP. This allows us to adjust the ordering of objectives, which, after solving with techniques similar to Example 1, produce differently repaired schedules after disruptions. In Table 2, we propose several orderings of the similarity objectives that can be used as sub-objectives to the LMOCOP.

Table 2 Optimization orders used for feasibility restoration with rescheduling.

Optimization order	Lexicographic ordering
Reallocation first (A)	`#done` $\prec$ `#realloc` $\prec$ `#shift` $\prec$ `sum-shift` $\prec$ `max-shift`
Shift first (B)	`#done` $\prec$ `#shift` $\prec$ `sum-shift` $\prec$ `max-shift` $\prec$ `#realloc`
Sum of shifts first (C)	`#done` $\prec$ `sum-shift` $\prec$ `max-shift` $\prec$ `#shift` $\prec$ `#realloc`
Max-shift first (D)	`#done` $\prec$ `max-shift` $\prec$ `sum-shift` $\prec$ `#shift` $\prec$ `#realloc`

Figure 3 shows two repaired schedules for different orders of the sub-objectives on the same instance as Figure 2. Depending on the ordering of the similarity objectives, a different trade-off between re-allocating and re-scheduling tasks is found by the solver.

By optimizing the LMOCOP for each lexicographic ordering, we can present the decision maker with multiple alternatives on how to resolve the disruption; as can be seen, many of these resolutions go further than shifting or re-allocating single tasks, and repairing the allocation can involve a chain of moves in order to ensure all tasks can be allocated to a non-disrupted team; which is tedious and time-consuming for planners to do by hand.



(a) Using order **A**: `#realloc` = 3, `#shift` = 12. **(b)** Using order **B**:: `#realloc` = 14, `#shift` = 4.

Figure 3 Repaired schedule after disruptions using different optimization orders.

5 Computational results on solving the allocation/scheduling problem

In this section, we investigate the efficiency of the different modeling approaches for solving the allocation/scheduling problem and Section 6 investigates the efficiency of the explanation techniques. More concretely, this section aims to answer the following questions:

EQ1. How do Constraint Programming (CP), Integer Linear Programming (ILP) and Pseudo-Boolean (PB) solvers compare for solving the allocation problem?

EQ2. Which objective function yields the best fairness in terms of time-dispersion?

EQ3. What is the effect of adding a lower-bound from the problem relaxation in the scheduling problem?

We implement the models of Section 3.1 and 3.2 in the CPMpy [17] library version 0.9.23 and use as underlying solvers OR-Tools CP-SAT v.9.11; the ILP-solver Gurobi v12.0.0 and the Pseudo-Boolean solver Exact v2.1.0 [11, 13]. The experiments in this section were ran on a single thread of a Intel(R) Xeon(R) Silver 4514Y and 256GB of RAM. For the allocation problem, we set OR-Tools to not use any sub-solvers to evaluate a pure CP-approach of solving the problem and for the scheduling problem, we allow it to use 8 threads. We consider instances of 8h horizons over a period of 3 months. This corresponds to one shift of the worker-teams and an average number of 67.6 tasks to be scheduled and allocated. All instances and models are made publicly available on `https://www.github.com/ML-KULeuven/Explainable-Workforce-Scheduling`.

5.1 Solving the allocation problem

We first compare different solvers and models for allocating tasks with *fixed* start-times.

Minimizing the number of teams

In Figure 4, left side, we plot the number of instances solved to optimality by each solver, with and without symmetry breaking as described in Section 3.1. We notice that both CP and PB-solving can benefit greatly from breaking symmetries in the model. Indeed, both OR-Tools and Exact can solve ±125 instances when symmetry breaking is disabled. However, by imposing an order on the equivalent teams for the given time horizon, OR-Tools solves ±200 instances, and Exact solves all instances in our dataset. Interestingly, we notice that Gurobi does not benefit from breaking the symmetries in the model, and the performance slightly degrades instead. Still, in both settings, Gurobi is solves all instances in <1 seconds.

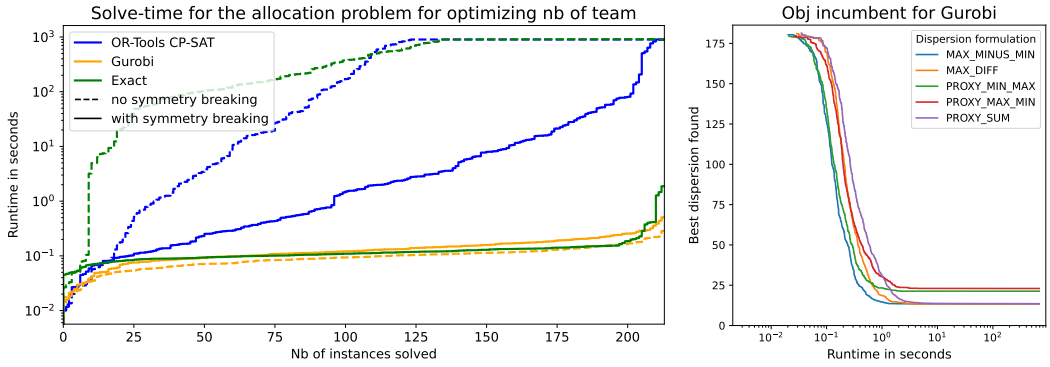


Figure 4 Computational results on the allocation use-case (time-horizon of 8h).

Minimizing time-dispersion

We found that computing optimal solutions for the time-dispersion using the straightforward formulation $\max_{w \in \mathcal{W}}(time_w) - \min_{w \in \mathcal{W}}(time_w)$ is a challenging problem, and solvers are often stuck proving optimality. Indeed, as shown in Section A, no solver was able to prove optimality for more than 30% of the instances. In Figure 4, right side, we plot the *true* dispersion during optimization for each of the dispersion-formulations outlined in Section 3.1 for Gurobi with results for other solvers in Section A. Interestingly, while Gurobi does not prove the optimality of the solution, the time-dispersion drops to $\pm 15min$ on average within 10s of solve-time. Furthermore, although more instances using the proxy-metrics are solved to optimality, this does not actually contribute to a better overall time-dispersion. From this experiment, we can conclude that although optimality is not proven for most instances, exact formulations of the time-dispersion work well for allocating the tasks fairly among the teams.

5.2 Solving the scheduling problem

We now turn to the much more computationally intensive case of *scheduling*, where tasks are not yet fixed in time. We only use OR-Tools CP-SAT as it is the only solver we tested with support for the appropriate global constraints. Figure 5, left side, shows the runtime for optimizing the number of teams in the scheduling problem. While OR-Tools is able to solve only ± 100 instances to optimality in the bare-bones model, adding the lower-bound of the relaxed problem greatly improves the performance of the solver and ± 100 extra instances are solved. The right-side of Figure 5 reveals that the lower-bound found by the solver is close to or equal to the true optimal number of teams required to allocate all tasks and hence, by adding the lower-bound to the model, we alleviate a lot of the work in proving optimality. In 83% of instances, the computed lower-bound is a proven optimal for the relaxed problem.

6 Computational results on explanation-generation

In this section, we investigate how effective the explanation methods of Section 4 are for explaining and correcting infeasible schedules due to disruptions. More precisely, we first investigate the computation time and size of MUSes and MCSes and subsequently, we investigate how infeasible schedules can be repaired using feasibility restoration methods. We use the generator from Section 4.1 to generate 4663 instances, summarized in Section B. All experiments were ran on a 2.8GHz AMD CPU with 16 cores and 64GB of RAM.

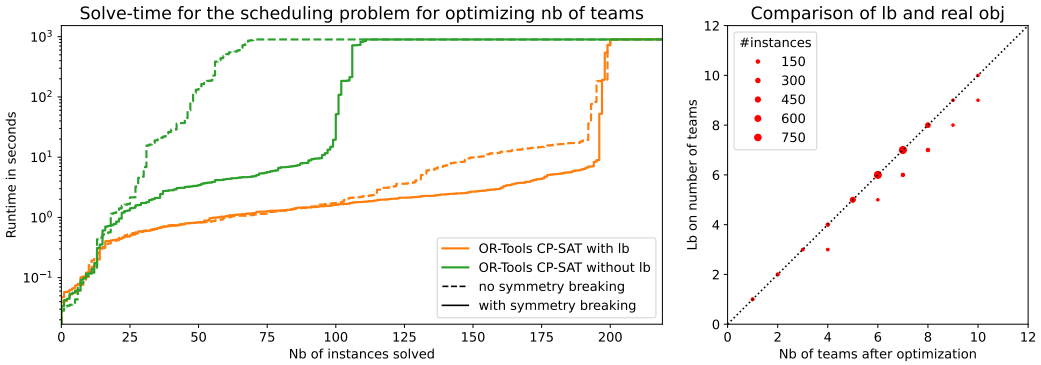


Figure 5 Computational results averaged over 5 seeds on the scheduling use-case.

6.1 Computation of MUS and MCS explanations

The results presented in this section were obtained by running default implementations of MUS and MCS-algorithms as implemented in CPMpy v0.9.23, on the unsatisfiable scenarios listed in Table 7. In particular, we use Shrink [23] for computing *any* MUS, SMUS [19] for computing a smallest MUS, Grow [24, 25] for finding *any* MCS and finally we use a reified Max-CSP formulation for finding an optimal MCS. In all MUS and MCS-experiments we use Exact [11] as solver, as it supports incremental and assumption-based solving [34].

Runtime of explanation-computation

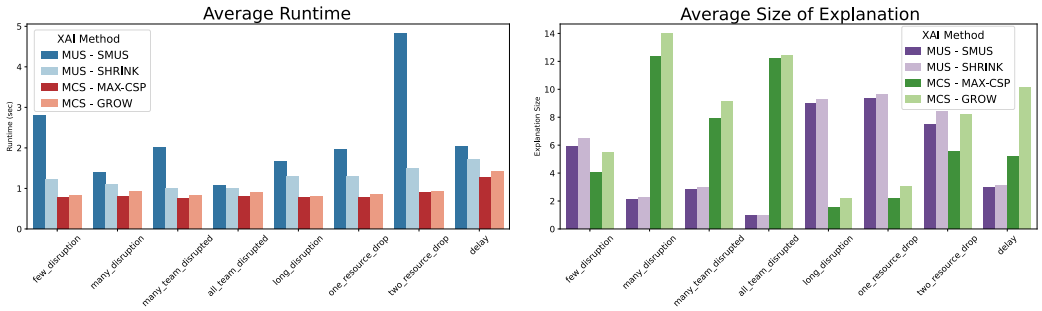
In our tool, we want to be able to quickly compute explanations when disruptions occur, such that the planners can interact with these explanations. For computing MUSes, there is a clear runtime-difference between a smallest MUS, and computing a MUS using Shrink as the runtime is doubled on average. Moreover, computing optimal MUSes times-out in 5.81% of the instances and yields significantly higher runtime for some disruption types such as the `two teams dropped` scenario, shown in Figure 3, left side. This is likely due to the inner workings of the SMUS algorithm which iteratively computes MCSes. In this scenario, many tasks can be re-allocated and hence, many MCSes may be enumerated.

Computing an MCS using Grow and optimal MCSes using Max-CSP shows less difference in runtime, and both methods compute MCSes in under 1 second on average.

So, while computing MUSes and MCSes can be done quickly for this application, the potential time-out for SMUS does not outweigh the slight decrease in explanation size compared to computing MUSes using the simpler Shrink algorithm.

Explanation size

From an application perspective, it is preferable to show small explanations are they are typically easier and quicker to understand by the planner. Figure 6b shows the size of MUSes and MCSes found by various methods for all disruption scenarios. Here, we clearly see a large



(a) Runtime of computing MUSes and MCSes. **(b)** Size of explanation.

Figure 6 Comparison of runtime and explanation quality for different scenarios.

Table 3 Summary of averaged performance metrics for XAI methods on the allocation use-case. The number of constraints requiring a task to be allocated in explanations is shown in brackets.

Method	Time(s)	MUS size	%Timeout	Method	Time(s)	MCS size	%Timeout
SMUS	2.06	**4.42 (3.72)**	5.81	Max-CSP	**0.82**	**6.67 (6.29)**	0.00
Shrink	**1.13**	4.55 (3.99)	0.04	Grow	0.88	7.99 (7.31)	0.00

difference in the size of MUSes depending on the scenarios. For example, when all teams are disrupted, a MUS consists of just a single task that is allocated within that disruption period. Conversely, when one or two teams are dropped from the available set of teams, a MUS contains a set of tasks that fully saturates the original set of available teams, and one of these tasks was allocated to a now dropped team. Additionally, we can observe that in all scenarios, the size of an optimal MUS is close to the size of an arbitrary MUS computed by Shrink. This suggests that all found MUSes have more or less the same size for our setting, given a disruption scenario.

Comparing the size of MCSes, there is overall a larger difference between the optimal MCS found by Max-CSP and one found by Grow. For example, in the `delay` scenario, the Max-CSP approach on average finds a set of 6 constraints to drop, compared to 10 in the grow-based approach. Interestingly, the scenarios where one full team is dropped from the set of available teams do not necessarily correspond to larger MCSes, as only a few tasks need to be dropped from the allocation. This suggests that the full set of teams is only required at a few points in the working-shift, and most tasks can be allocated to different teams by (potentially many) re-allocations, instead of being dropped. Conversely, when many teams are affected by the same disruption, most tasks that overlap with this disruption have to be dropped from the planning horizon.

6.2 Multi-objective feasibility restoration

We now investigate the effectiveness of restoring feasibility by computing fine-grained alterations to the schedule as described in Section 4.3.2. We use OR-Tools CP-SAT with 10 search workers to solve the lexicographic optimization models as introduced in Section 4.3.1, using the sequential solving approach outlined in Section 2. For each of the 5 objectives, we set a time-limit of 5 seconds, such that the total time for feasibility restoration is limited to 25s. Table 4 summarizes the final objective value for each similarity objective for each of the optimization orders and across all scenario types. The number of not-done tasks (`#not-done`) is derived from the `#done` similarity objective.

Table 4 Averaged objective value for feasibility restoration, grouped per lexicographic ordering.

Optimization order	`#not-done`	`#realloc`	`#shift`	`sum-shift`	`max-shift`
Reallocation first (A)	1.00	**8.37**	23.21	289.21	24.04
Shift first (B)	0.96	22.60	17.59	244.75	21.40
Sum Shift first (C)	0.98	22.48	**16.66**	**198.93**	16.15
Max Shift first (D)	0.98	22.95	17.54	199.71	**14.91**

Comparing the number of dropped tasks in the above table to the MCS-results in Table 3, we can clearly see that more-fine grained feasibility restoration leads to less tasks dropped. Indeed, repaired schedules using the optimal MCS drops on average 6.29 tasks, whereas allowing for re-scheduling of tasks only drops a single task on average.

As expected, there is a clear trade-off between shifting tasks and re-allocating tasks after a disruption. In “Reallocation first (A)”, which prioritizes minimization of re-allocated tasks, 8.37 tasks are re-allocated on average, while for the other orders between 22 and 23 tasks are re-allocated. Conversely, prioritizing minimization of shift-related metrics amounts to a lower number of shifted tasks (16-17) at the cost of more re-allocations.

Table 4 shows that by altering the order of the objectives, the resulting repaired schedule can be influenced significantly, and hence, we can use these orderings to propose several alternatives to the planners.

Table 5 Fraction of instances solved to optimality for each sub-objective.

Optimization order	`#not-done`	`#realloc`	`#shift`	`sum-shift`	`max-shift`
Reallocation first (A)	0.97	0.96	0.84	0.90	0.89
Shift first (B)	0.98	0.63	0.60	0.68	0.66
Sum shift first (C)	0.97	0.64	0.43	0.80	0.67
Max shift first (D)	0.97	0.65	0.59	0.88	0.68

Interestingly, optimizing the total magnitude of shifted tasks, results in less tasks that are shifted on average, compared to minimizing `#shift` directly. Table 5 reveals that this is due to `#shift` being a more difficult objective to optimize compared to `sum-shift`, thereby often reaching the time-limit of 5s. Indeed, in only 60% of the instances, OR-Tools was able to provide an optimal solution when optimizing this continuity objective first. In "Sum Shift First (C)", the `sum-shift` continuity objective was proven optimal in 80% of the instances and, by minimizing this simpler objective first, the resulting repaired schedule contains less shifted tasks overall.

7 Conclusions and future work

We have presented an industrial setting of a workforce-allocation and scheduling problem. We have showed that, using modeling techniques such as symmetry breaking, both PB and ILP-solvers can solve the allocation problem efficiently, and produce fair solutions regarding time-dispersion. For the scheduling variant, additional techniques such as adding the lower-bound of a relaxed problem are required in order to find optimal schedules. In future work, we plan to explore allowing for direct transfers from one location to another, which requires extending our formulation to a full allocation-and-routing problem [5].

Additionally, we have identified several disruption scenarios and investigated how to support a decision maker using explainable constraint solving techniques; we have experimentally shown that explanations such as MUSes and feasibility restoration can be computed efficiently. Hence, we deem it feasible to deploy such an interactive system as a decision support system for the planners. We are currently setting up a user-study to evaluate how effective the different generated explanations are in practice. Such user-study will enable to further tune the type of explanations generated by the system and going further, could allow us to learn preferences of the planners in repairing the schedule [16, 2, 31].

A future, alternative setting for dealing with possible disruptions in the schedule, is to ensure the original solution is *robust* [22]. A robust solution remains feasible, or even optimal when the parameters of the constraint model change. A robust schedule can for example introduce larger breaks in between tasks to account for possible delays. That robustness of a solution typically comes at the cost of higher computation times, and the need for specialized solving algorithms, which makes integration with production systems nontrivial.

Finally, we want to explore how we can assist planners by generating explanations for the full scheduling variant as well, similar to the explanations we developed for the allocation problem in this paper. This raises new challenges such as how to efficiently use global constraints in MUS and MCS-computation algorithms.

References

1 J Christopher Beck and Mark S Fox. Constraint-directed techniques for scheduling alternative activities. *Artificial Intelligence*, 121(1-2):211–250, 2000. doi:10.1016/S0004-3702(00)00035-7.

2 Nawal Benabbou and Thibaut Lust. An interactive polyhedral approach for multi-objective combinatorial optimization with incomplete preference information. In *International Conference on Scalable Uncertainty Management*, pages 221–235. Springer, 2019. doi:10.1007/978-3-030-35514-2_17.

3 Ignace Bleukx, Tias Guns, and Dimos Tsouros. Explainable Constraint Solving: A hands-on tutorial, February 2024. doi:10.5281/zenodo.10694140.

4 Nils Boysen, Philipp Schulze, and Armin Scholl. Assembly line balancing: What happened in the last fifteen years? *European Journal of Operational Research*, 301(3):797–814, 2022. doi:10.1016/J.EJOR.2021.11.043.

5 J Arturo Castillo-Salazar, Dario Landa-Silva, and Rong Qu. Workforce scheduling and routing problems: literature survey and computational study. *Annals of Operations Research*, 239:39–67, 2016. doi:10.1007/S10479-014-1687-2.

6 Violet Xinying Chen and JN Hooker. A guide to formulating equity and fairness in an optimization model. *Preprint*, pages 162–174, 2021.

7 João Cortes, Inês Lynce, and Vasco Manquinho. New core-guided and hitting set algorithms for multi-objective combinatorial optimization. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 55–73. Springer, 2023. doi:10.1007/978-3-031-30820-8_7.

8 João Cortes, Inês Lynce, and Vasco Manquinho. Slide&drill, a new approach for multi-objective combinatorial optimization. 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 8:1–8:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CP.2024.8.

9 Giacomo Da Col and Erich C Teppan. Industrial-size job shop scheduling with constraint programming. *Operations Research Perspectives*, 9:100249, 2022.

10 Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, Gerda Janssens, and Marc Denecker. Declarative logic programming. In Michael Kifer and Yanhong Annie Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, chapter Predicate Logic As a Modeling Language: The IDP System, pages 279–323. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2018. doi:10.1145/3191315.3191321.

11 Jo Devriendt. Exact solver, 2023. URL: https://gitlab.com/JoD/exact.

12 Zehranaz Dönmez, Merve Ayyıldız, Benay Uslu, Ozlem Karsu, and Bahar Y Kara. Fairness in humanitarian logistics: State of the art and future directions, 2022.

13 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.

14 Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. Plan stability: Replanning versus plan repair. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*, pages 212–221. AAAI, 2006. URL: http://www.aaai.org/Library/ICAPS/2006/icaps06-022.php.

15 Emilio Gamba, Bart Bogaerts, and Tias Guns. Efficiently explaining csps with unsatisfiable subset optimization. *J. Artif. Intell. Res.*, 78:709–746, 2023. doi:10.1613/JAIR.1.14260.

16 Andreia P. Guerreiro, João Cortes, Daniel Vanderpooten, Cristina Bazgan, Inês Lynce, Vasco Manquinho, and José Rui Figueira. Exact and approximate determination of the pareto front using minimal correction subsets. *Comput. Oper. Res.*, 153:106153, 2023. doi:10.1016/J.COR.2023.106153.

17 Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

18 Sharmi Dev Gupta, Begum Genc, and Barry O'Sullivan. Explanation in constraint satisfaction: A survey. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 4400–4407. ijcai.org, 2021. doi:10.24963/ijcai.2021/601.

19 Alexey Ignatiev, Alessandro Previti, Mark H. Liffiton, and João Marques-Silva. Smallest MUS extraction with minimal hitting set dualization. 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 173–182. Springer, 2015. doi:10.1007/978-3-319-23219-5_13.

20 Ulrich Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, volume 4. Citeseer, 2001.

21 Philippe Laborie, Jerome Rogerie, Paul Shaw, and Petr Vilím. Reasoning with conditional time-intervals. part II: an algebraical model for resources. In H. Chad Lane and Hans W. Guesgen, editors, *Proceedings of the Twenty-Second International Florida Artificial Intelligence Research Society Conference, May 19-21, 2009, Sanibel Island, Florida, USA*. AAAI Press, 2009. URL: http://aaai.org/ocs/index.php/FLAIRS/2009/paper/view/60.

22 Jheisson López, Alejandro Arbelaez, and Laura Climent. Large neighborhood search for robust solutions for constraint satisfaction problems with ordered domains. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming, CP 2022, July 31 to August 8, 2022, Haifa, Israel*, volume 235 of *LIPIcs*, pages 33:1–33:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CP.2022.33.

23 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.

24 João Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 615–622. IJCAI/AAAI, 2013. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6922.

25 Carlos Mencía and Joao Marques-Silva. Efficient relaxations of over-constrained csps. In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, pages 725–732. IEEE, 2014.

26 Raymond Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32(1):57–95, 1987. doi:10.1016/0004-3702(87)90062-2.

27 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. URL: https://www.sciencedirect.com/science/bookseries/15746526/2.

28 Pierre Schaus and Renaud Hartert. Multi-objective large neighborhood search. 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 611–627. Springer, 2013. doi:10.1007/978-3-642-40627-0_46.

29 Andreas Schutt, Thibaut Feydy, and Peter J Stuckey. Explaining time-table-edge-finding propagation for the cumulative resource constraint. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings 10*, pages 234–250. Springer, 2013. doi:10.1007/978-3-642-38171-3_16.

30 Ilankaikone Senthooran, Matthias Klapperstück, Gleb Belov, Tobias Czauderna, Kevin Leo, Mark Wallace, Michael Wybrow, and Maria Garcia de la Banda. Human-centred feasibility restoration in practice. *Constraints An Int. J.*, 28(2):203–243, 2023. doi:10.1007/S10601-023-09344-5.

31 Federico Toffano, Michele Garraffa, Yiqing Lin, Steven Prestwich, Helmut Simonis, and Nic Wilson. A multi-objective supplier selection framework based on user-preferences. *Annals of Operations Research*, pages 1–32, 2022.

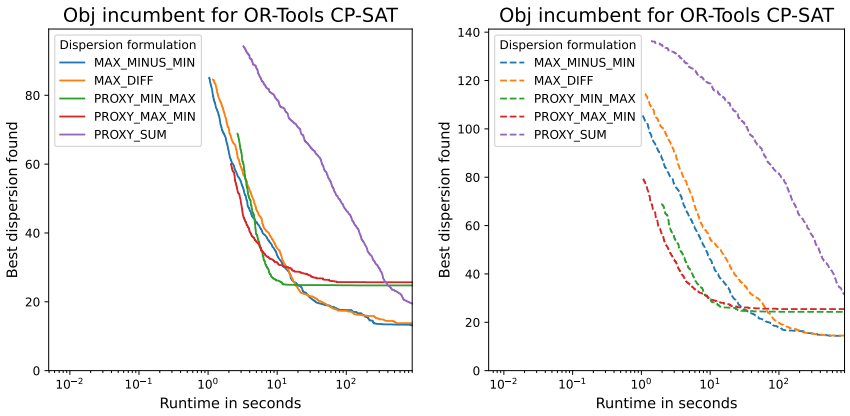
32 Petr Vilím. Edge finding filtering algorithm for discrete cumulative resources in $O(kn \log n)$. In *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 802–816. Springer, 2009. doi:10.1007/978-3-642-04244-7_62.

33 Petr Vilím. Max energy filtering algorithm for discrete cumulative resources. In *CPAIOR*, volume 5547 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2009. doi:10.1007/978-3-642-01929-6_22.

34 Siert Wieringa. *Incremental satisfiability solving and its applications*. PhD thesis, Aalto University School of Science, Department of Computer Science and Engineering, 2014.

A Detailed results on optimizing dispersion

Below, we show the detailed objective incumbent for each of the solver/model configurations. Note that we show the *true* dispersion value according to the exact formulation.

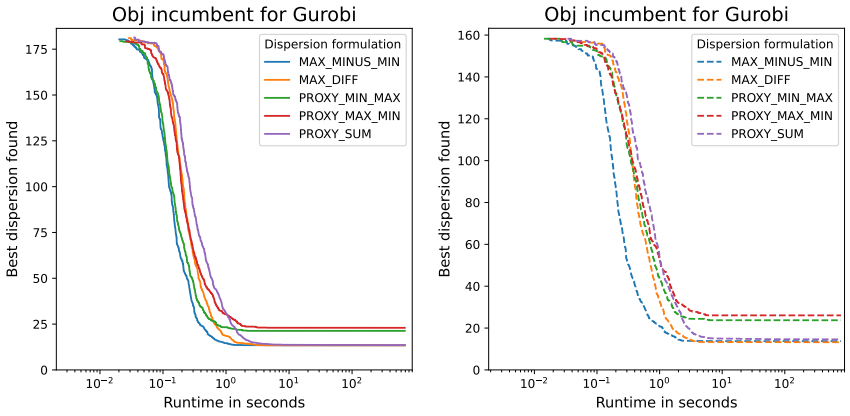


(a) With symmetry breaking. **(b)** Without symmetry breaking.

Figure 7 Dispersion objective incumbent for OR-Tools CP-SAT.

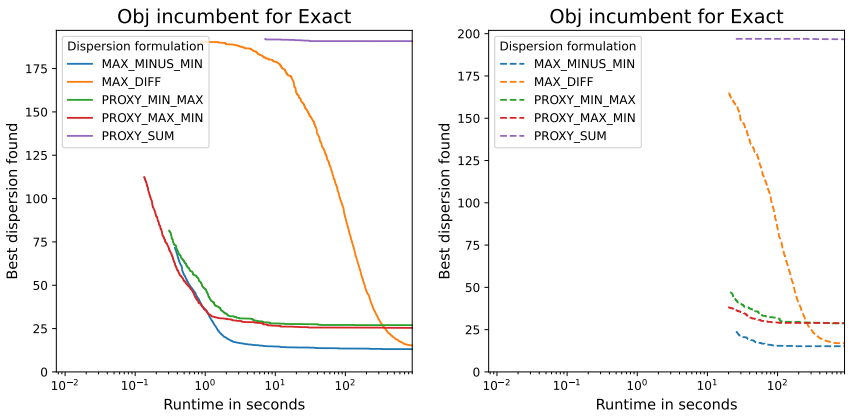
Table 6 Percentage of instances solved to optimality.

Solver	Break symmetries?	Dispersion variant	%solved to optimality
OR-Tools CP-SAT	No	MAX_DIFF	7.41
		MAX_MINUS_MIN	8.52
		PROXY_MAX_MIN	5.93
		PROXY_MIN_MAX	8.52
		PROXY_SUM	7.04
	Yes	MAX_DIFF	9.63
		MAX_MINUS_MIN	9.63
		PROXY_MAX_MIN	5.93
		PROXY_MIN_MAX	8.89
		PROXY_SUM	7.41
Gurobi	No	MAX_DIFF	17.41
		MAX_MINUS_MIN	18.89
		PROXY_MAX_MIN	78.15
		PROXY_MIN_MAX	78.15
		PROXY_SUM	13.33
	Yes	MAX_DIFF	30.37
		MAX_MINUS_MIN	33.33
		PROXY_MAX_MIN	78.52
		PROXY_MIN_MAX	78.52
		PROXY_SUM	15.56
Exact	No	MAX_DIFF	7.41
		MAX_MINUS_MIN	8.89
		PROXY_MAX_MIN	5.19
		PROXY_MIN_MAX	5.56
		PROXY_SUM	4.81
	Yes	MAX_DIFF	10.00
		MAX_MINUS_MIN	14.44
		PROXY_MAX_MIN	5.93
		PROXY_MIN_MAX	11.85
		PROXY_SUM	22.22



(a) With symmetry breaking. **(b)** Without symmetry breaking.

Figure 8 Dispersion objective incumbent for Gurobi.



(a) With symmetry breaking. **(b)** Without symmetry breaking.

Figure 9 Dispersion objective incumbent for Exact.

Figure 10 shows the number of instances and solve-time for finding a lower-bound of the scheduling problem by solving the relaxed problem from Section 3.2.

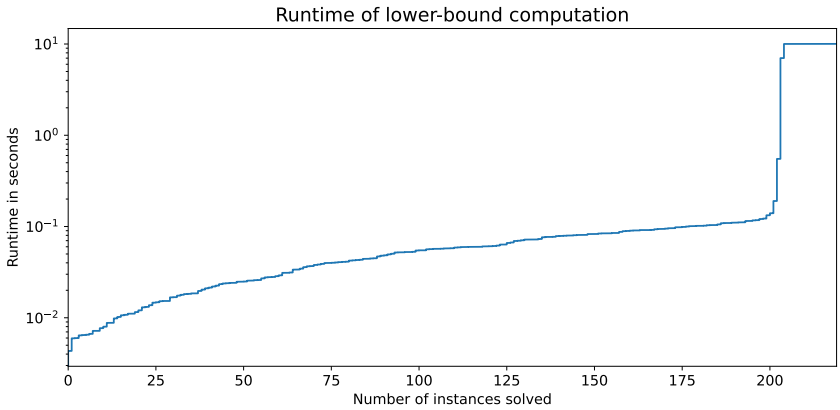


Figure 10 Runtime for solving the relaxed problem to optimality.

B Detailed results on explanation-generation

Table 7 summarizes the instances used for all explanation-generation methods for each scenario.

Table 7 Overview of generated scenarios, relative value to total number of tasks in parenthesis.

Scenario description	#Instances count	#Disr. mean	#Teams(%) mean	$\sum$ duration mean	#Tasks Disr(%) mean
`few disruptions`	1167	2.71	5.66 (0.6)	411.56	11.23 (0.11)
`long disruptions`	411	2.45	2.27 (0.26)	582.19	6.48 (0.06)
`many disruptions`	128	**6.16**	**8.72 (1)**	**1355.74**	**17.88 (0.13)**
`many teams affected`	1049	1	7.83 (0.83)	361.74	9.11 (0.08)
`all teams affected`	974	1	**9.46 (1)**	383.59	10.58 (0.09)
`one team dropped`	334	1	1 (0.11)	744.43	13.27 (0.11)
`two teams dropped`	306	1	2 (0.21)	**1480.78**	**26.48 (0.21)**
`delay`	294	–	–	–	–

Table 8 provides a summary of the number of instances, average runtime, and average explanation size, categorized by scenario type and XAI method. The explanation size corresponds to the number of constraints involved in the generated explanations.

Table 8 Total number of instances, average runtime, and average explanation size, number of timeout, broken down by scenario type and XAI method (MCS vs. MUS).

Scenario description	Explanation type	No. Of instances	Runtime (s) ↓	Size XAI ↓	Timeout
`few disruptions`	MCS	2334	0.80	**4.75**	0
	MUS	2186	1.95	6.23	148
`many disruptions`	MCS	256	0.86	13.18	0
	MUS	254	1.24	**2.16**	2
`many teams affected`	MCS	2098	0.80	8.55	0
	MUS	2072	1.50	**2.89**	26
`all teams affected`	MCS	1948	0.84	12.31	0
	MUS	1948	1.03	**1.00**	0
`long disruptions`	MCS	822	0.79	**1.87**	0
	MUS	821	1.48	9.12	1
`one team dropped`	MCS	668	0.82	**2.63**	0
	MUS	666	1.63	9.49	2
`two teams dropped`	MCS	612	0.92	**6.90**	0
	MUS	518	2.86	8.05	94
`delay`	MCS	588	1.35	7.67	0
	MUS	588	1.87	**3.05**	0

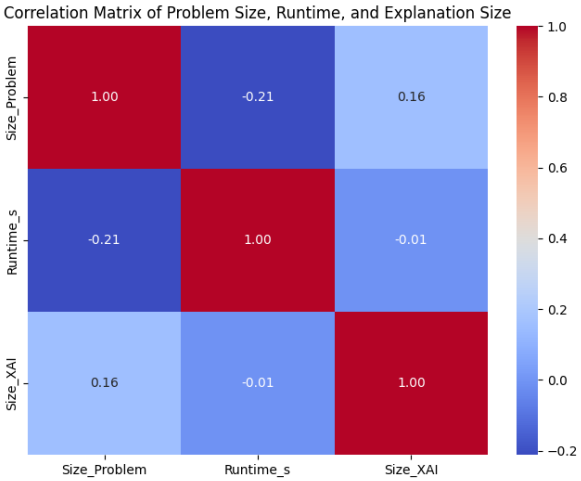


Figure 11 Correlation between problem size, runtime, and explanation size.

C Detailed results on rescheduling per scenario type

Table 9 display the detailed results for each scenario type, using the objective order A. We see that the scenario type with the most task that are dropped is the `all teams affected`. On other continuity objectives, the `two teams dropped` implies more shifts.

Table 9 Rescheduling results per scenario type (only with optimisation order **A**).

Scenario description	#not-done	#realloc	#shift	sum-shift	max-shift
few disruptions	0.21	8.19	26.36	347.48	26.43
many disruptions	1.80	8.81	29.08	392.32	25.85
many teams affected	0.94	5.55	22.83	302.94	25.85
all teams affected	**2.87**	5.15	24.21	305.54	24.85
long disruptions	0.19	8.23	11.62	140.29	21.94
one team dropped	0.42	14.47	13.47	156.26	22.69
two teams dropped	0.67	**28.07**	**36.95**	**507.95**	26.09
delay	0.01	2.00	20.51	57.04	6.97

Table 10 details the optimality proof per scenario type, highlighting notably that the scenario `two teams dropped` is the most difficult.

Table 10 Fraction of optimality proof for each lexico-phase, for each scenario type.

Scenario description	#not-done mean	#realloc mean	#shift mean	sum-shift mean	max-shift mean
few disruptions	0.98	0.54	0.41	0.69	0.53
many disruptions	0.92	0.66	0.58	0.87	0.72
many teams affected	0.95	0.83	0.73	0.93	0.87
all teams affected	0.97	0.86	0.79	0.98	0.89
long disruptions	0.98	0.75	0.58	0.73	0.71
one team dropped	0.99	0.71	0.58	0.77	0.73
two teams dropped	0.96	0.34	0.12	0.44	0.24
delay	0.99	0.99	0.99	0.96	0.99

Optimizing 2D Cutting: A Bin Packing Approach to Minimize Scraps and Maximize Their Reusability

Manuel Chastenay
Lab-Usine – Joint Research Unit for Advanced Manufacturing, Université Laval, Québec, Canada
SOKÏO Industrie, Saint-Augustin-de-Desmaures, Canada

Xavier Zwingmann
Lab-Usine – Joint Research Unit for Advanced Manufacturing, Université Laval, Québec, Canada
SOKÏO Industrie, Saint-Augustin-de-Desmaures, Canada

Claude-Guy Quimper
Lab-Usine – Joint Research Unit for Advanced Manufacturing, Université Laval, Québec, Canada

Jonathan Gaudreault
Lab-Usine – Joint Research Unit for Advanced Manufacturing, Université Laval, Québec, Canada

Abstract

In industrial settings, cutting predefined pieces from one or multiple sheets of material is a common optimization challenge. This problem can be formulated as a variant of the 2D bin packing problem, where the edges of the pieces define the cut lines. This paper presents a constraint programming model developed in collaboration with an industrial partner in construction to minimize scrap waste generated when cutting insulation pieces. The model introduces an objective function designed to maximize the reusability of leftover material. To fully leverage the model's efficiency, an initial process transforms irregular insulation pieces into rectangles using one of four processing methods. A comparative analysis is conducted to evaluate the impact of these methods, as well as to benchmark the model's results against the partner's manual approach.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Combinatorial optimization, constraint programming, 2D bin packing

Digital Object Identifier 10.4230/LIPIcs.CP.2025.7

Funding This research is funded by SOKÏO jointly with the Mitacs Accelerate and NSERC Alliance grants. Manuel Chastenay and Xavier Zwingmann receive a FRQNT Master Scholarship and Xavier Zwingmann also receives an NSERC Graduate Research Scholarship – Master's program.

Acknowledgements We thank SOKÏO for their support and for providing the benchmark.

1 Introduction

Efficient material utilization is a critical concern in modern manufacturing. Minimizing waste directly translates into environmental benefits and helps industries reduce costs. In the construction industry, cutting insulation pieces from material sheets presents a complex but common challenge. At its core, this problem can be seen as a variant of the two-dimensional bin packing problem, in which the goal is to pack irregular shapes onto sheets while minimizing the unused area in said sheets. Unlike classical packing problems where no cutting is done, the unused area here results in waste. Thus, the insulation nesting task further demands that scraps not only be minimized but also have a shape that favors reusability for cutting future pieces. For example, a single rectangular scrap is preferable to many long and thin pieces, even if their combined area is the same.

In practice, insulation pieces are not always rectangles. Instead, they are often made up of multiple different shapes predefined in a way that fits the factory that produces them. Moreover, the pieces can be rotated or flipped to fit better on the sheet, which increases

© Manuel Chastenay, Xavier Zwingmann, Claude-Guy Quimper, and Jonathan Gaudreault;
licensed under Creative Commons License CC-BY 4.0

the combinatorial complexity of the problem. This increased complexity, coupled with the heterogeneous nature of the available sheets (new sheets and recycled scraps), makes the already NP-Hard 2D bin packing problem [14, 15] even more complex and requires the use of sophisticated heuristics and exact methods.

This work presents two novel constraint models that integrate one distinct objective, each. First, the total area of the sheets that are used to nest the pieces is minimized. This reduces the number of sheets used and prioritizes the use of scraps over new sheets. Second, using the sheets found using the first model, the placement of all pieces is optimized to enhance the reusability of any resultant scrap. To manage the inherent computational difficulty, irregular insulation pieces are transformed into rectangles by a preprocessing phase that occurs before the optimization phase. The preprocessing uses one of the four new algorithms that we introduce. These methods aim to reduce the search space of the optimization process, thus improving computational efficiency.

The proposed approach is compared with the manual strategies traditionally used by our collaborating industrial partner. The analysis highlights improvements in material utilization and scrap reusability, which offer a significant step toward automated and sustainable production practices that can be applied to many industries.

2 Preliminary concepts

2.1 Cutting and Packing Problems

Cutting and packing problems [6] are part of a category of problems that have been studied well before computers. The simplicity of these problems, paired with the fact that they arise naturally in a lot of domains, makes them good candidates for research in optimization. Since Gilmore and Gomory [16] laid the foundation of modern computational approaches to the cutting stock problem, the field grew much, and many subproblems were officially classified. Wäscher et al. [27] proposed the (currently) most popular of these classifications, splitting cutting and packing problems into two distinct categories. The first category, **output maximization**, considers problems in which items have to be packed in a single container of fixed size, maximizing the number of items packed (or their value). The second, **input minimization**, considers problems where all pieces must be packed in one or many containers, minimizing the number of containers used. This category contains problems such as the open dimension problem (ODP), also known as the strip packing problem (SPP) [18]. Schutt et al. proposed a great example of the use of the SPP for industrial purposes in [24]. The SPP considers a single container of infinite length, which differentiates it from the cutting stock problem (CSP) [8, 17] and the bin packing problem (BPP) [1, 3, 9, 18, 22]. These two problems consider the use of one or many containers of fixed sizes. Although both subproblems manage the use of homogeneous and heterogeneous containers, their main difference comes from the items to pack. Where the CSP uses predefined cutting patterns to determine how to cut the different pieces, the BPP uses no pattern and instead works with highly heterogeneous items as input. Our problem is a specific implementation of the BPP for which we present a model that acts on a problem that is not well documented in the literature: maximizing the potential reusability of scraps. While minimizing scraps is common, maximizing their reusability is not and justifies the implementation of a new model.

2.2 Constraint Programming

Constraint programming (CP) is a programming paradigm specialized in solving satisfaction and optimization problems, such as those presented in Section 2.1 and in [25]. These problems are defined as mathematical models, where the decision variables are linked together with constraints that restrict the possible assignments to the solution space. Many predefined constraints already exist in popular solvers, often under different names. This disparity (and the fact that solvers are often implemented in different programming languages) led, in 2007, to the creation of MiniZinc [21], which is a CP modeling language that can seamlessly translate the defined problem to work with multiple solvers, such as Chuffed [10] and CP-SAT [23]. The presented MiniZinc models incorporate two important constraints: DiffN and Cumulative.

The DiffN constraint [4, 26] considers a set of two-dimensional rectangles and constrains their relative placements so that they do not overlap. The constraint is formally expressed as: DiffN($[X_1, \ldots, X_n], [Y_1, \ldots, Y_n], [width_1, \ldots, width_n], [height_1, \ldots, height_n]$), where X_i and Y_i represent the positions of the rectangles and $height_i$ and $width_i$ their dimensions. This constraint is especially useful for cutting and packing problems.

The Cumulative constraint [2, 26] is used in scheduling problems to limit the number of tasks that can be executed simultaneously, but it can also be used efficiently in cutting and packing problems. Since it encodes the placement of tasks on a timeline, it can also encode the position of pieces over one dimension. Formally, it is expressed as: Cumulative($[s_1, \ldots, s_n]$, $[d_1, \ldots, d_n], [r_1, \ldots, r_n], B$) , where s_i represents the position of a piece, d_i its length, and r_i its height. B represents the maximum height allowed. It ensures that at any position along the dimension, the total height of the overlapping pieces does not exceed B.

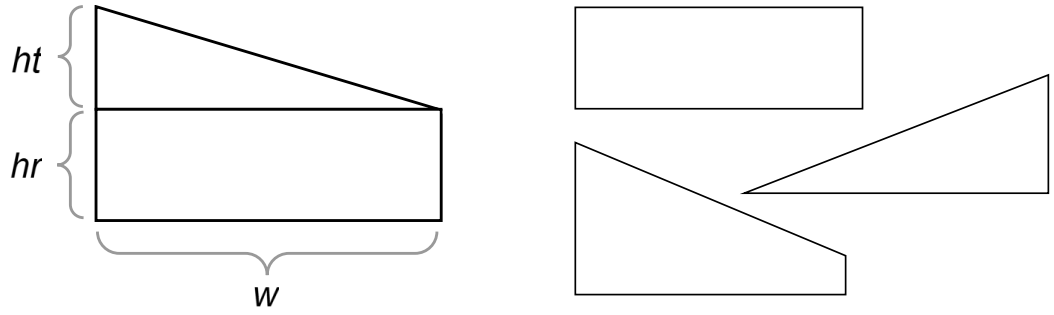
Adding the Cumulative to the DiffN is considered redundant. In our case, this redundancy improves the solver convergence speed by pruning invalid placements early and more efficiently than by using only DiffN [20].

3 Case study

SOKÏO is a wood manufacturer that is developing a highly customizable construction system. Its products are buildings whose components, such as walls, roofs, and floors, are made from cross-laminated timber (CLT) and are assembled directly in the factory in a fully automated way. For example, walls are made from layering the main CLT panel, insulation, doors, windows, and cladding. After being layered, all panels are brought to the site to be assembled, making it possible for the installation to be completed in one or two days at most.

Built with automation in mind, the different systems fueling the factory's production are seamlessly linked together and can communicate in a digital twin (DT) [12] developed in the Unity® game engine. Clients and architects eager to design their own dream building can freely do so in the DT, while the back-end of the app built using constraint programming ensures that the building fully respects all the different constraints of the factory.

The system in question in this paper concerns the nesting of insulation pieces. The shapes and dimensions of these pieces are calculated in real time during the product configuration process [28], and sent to the nesting module. As explained in the next section, this system computes an optimal way to cut the different input pieces into predefined sheets while optimizing key metrics, such as scrap reusability. This process saves a lot of time for SOKÏO, which used to do it by hand. It also saves critical quantities of insulation material by facilitating its reusability and reduces operations and labor costs.



(a) Specifications.

(b) Examples.

Figure 1 Insulation pieces.

4 Problem presentation

To provide context for the proposed algorithms, it is important to understand the basic 2D bin packing problem [18, 22]. This problem involves two components: **i)** a set of rectangular items, and **ii)** a set of rectangular bins in which we need to fit all the items. The bins typically have the same predefined dimensions. The items also have their dimensions set in advance, with the added assumption that they can fit in the bins. These items can rotate 0° (without rotation) or 90°. Using these definitions, the goal is to find the smallest number of bins in which it is possible to fit all the items.

4.1 Insulation nesting

When applying these concepts to the insulation nesting problem, the bins are represented by insulation sheets, and the precomputed insulation pieces are the items. All insulation sheets come in a standard size of 8 feet by 4 feet, and the precision of the tools used to cut by SOKÏO is $\frac{1}{16}$ of an inch. All pieces are cut individually from the sheets, without common or guillotine cuts constraints [22]. We now add to the complexity of this problem by refining it with these new modifications/additions:

1. The shape of the precomputed pieces is now defined by the following constraints, represented in Figure 1: **i)** The piece must have a rectangular component of width $w > 0$ and height $hr \geq 0$. **ii)** It must have a right triangle component, whose width is equal to w and height $ht \geq 0$. **iii)** The sum of heights hr and ht is strictly greater than 0. These shape definitions allow for rectangular, triangular, and trapezoidal insulation pieces.
2. We use two rotational values (0° and 90°) when working with rectangular elements. These rotations offer the possibility for the sides of the items to align in an orthogonal way to the edges of the bins. Although this is not enough to obtain optimality in all cases [7], this is a great compromise between simplicity and results that are more than sufficient for our particular domain. For non-rectangular elements defined by the shape presented in point 1, we are required to use more than two rotations. Since at most one side of the trapezoidal shape has two right angles, two possible rotational values (180° and 270°) are added to ensure that it can correctly align to the bin. Additionally, to account for the possible placements of the right triangle in the sheet, it is possible to flip the piece over. Thus, we go from two possible rotations to eight (four rotations if not flipped, four if flipped) when working with a shape that has a triangle height $ht > 0$.

3. Sheets are not all the same size. For SOKÏO and many other companies, the ability to reuse scraps and smaller sheets has great potential value. As a result, they need to be able to specify sheets that have dimensions smaller than the ones predefined for new sheets. These smaller sheets represent scraps that have been acquired by cutting insulation sheets in the past. The model has to be able to take these scraps into account so that they can be reused in the future. Following the above nomenclature, these different-sized bins are referred to as being **strongly** heterogeneous [27].
4. Although the main objective remains similar, we now aim to minimize the total area of the sheets in which the pieces are nested, rather than only their number. A second objective is also introduced. This is justified by the fact that the "creation" of scraps of sizes large enough to be reused later for nesting insulation pieces is critical, as implied by point 3. This second objective aims to position the insulation pieces in the sheet in a way that maximizes the reusability of its wasted area. This goal is complex to describe as a quantifiable metric due to its subjective nature, and this is part of the reason why the literature on this exact subject is rare, especially when working with constraint programming.

4.2 Instance specification

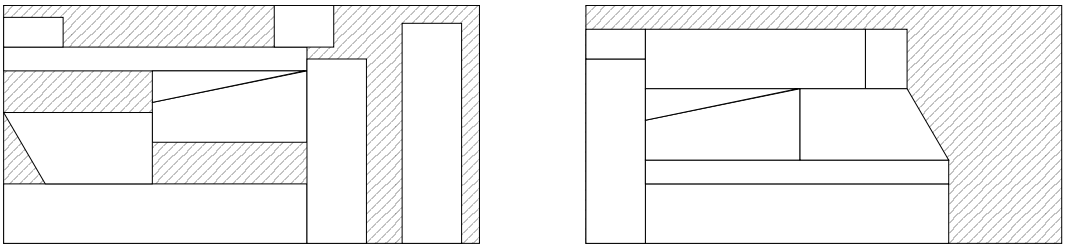
A formal specification of an instance of the described problem is defined as follows: We are given two sets: **i)** Let $\mathcal{P}$ be the set of indices of the insulation pieces. For each $p \in \mathcal{P}$, the piece p is defined by its width w_p, its rectangular height hr_p and its triangular height ht_p. **ii)** The set $\mathcal{R}$ contains the indices of rectangular sheets, which have predefined widths and heights (sw_r and $sh_r \; \forall \; r \in \mathcal{R}$). These are ordered by non-decreasing area $sw_r \cdot sh_r$, and represent old heterogeneous insulation scraps that can be reused as bins. Using these sets, we define $\mathcal{S}$, which contains all the scrap sheets in $\mathcal{R}$ plus $|\mathcal{P}|$ new sheets that are added in sufficient quantities to make the instance feasible. The new sheets all have the same dimensions. Each sheet $s \in \mathcal{S}$ is defined with their width sw_s and height sh_s.

The goal is to assign each piece $p \in \mathcal{P}$ a sheet $s \in \mathcal{S}$, a position in the sheet (considering that the piece's origin is located at its bottom left corner), and a rotation such that none of the pieces overlap, while minimizing the total area of the used sheets. After finding optimality for the area of the sheets that are used, we also want to optimize the scrap reusability, further refining the solution of the problem. All this being said, we can consider the typology brought up by Wäscher et al. [27] to classify the problem as a variant of the residual bin packing problem (RBPP), where the dimensions of the residual pieces are considered an objective during the optimization process. Of course, a quantifiable metric is needed to optimize this value, which will be introduced in Section 5.3.2. Figure 2 shows two possible placements of the same pieces in a sheet. The example in Figure 2a is objectively worse than the one in Figure 2b when comparing the reusability of the scraps.

5 Methodology

5.1 Preprocessing – converting pieces to rectangles

The NP-Hardness of the problem makes it difficult for large instances to be computed to optimality in a reasonable time. First, having to deal with shapes that contain angles non-divisible by 90° adds an incredible layer of complexity, as computing the collisions between the shapes now relies on using trigonometric functions rather than simple edge placement comparisons. The eight possible rotations also make the solver's search tree



(a) Scraps with non-optimal reusability.

(b) Scraps with optimal reusability.

Figure 2 Packing examples: Pieces (white rectangles) are packed within the hatched sheet. Visible hatched areas are the resulting scraps.

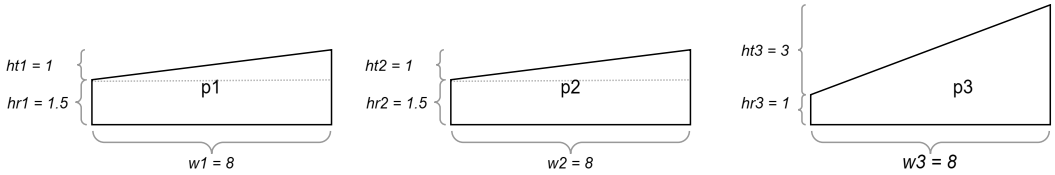
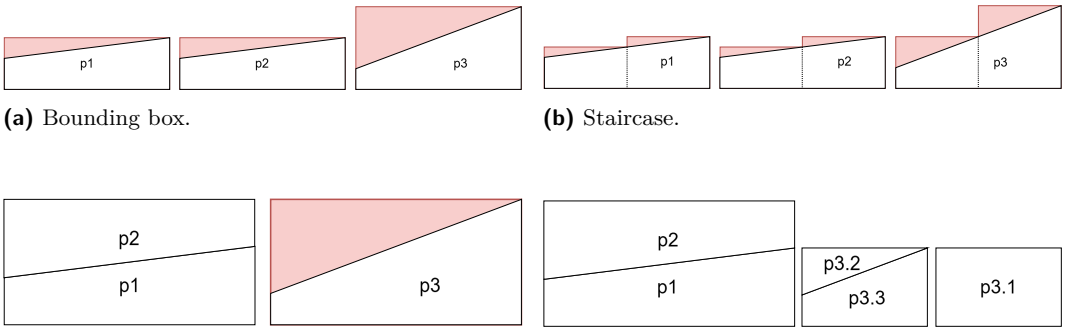


Figure 3 Instance being passed to the different preprocessing algorithms.

exponentially larger than if only two were used. That being said, four different polynomial-time preprocessing algorithms will be introduced and compared. Figure 3 shows an example of a set of pieces, which is sent to the different preprocessing methods to be analyzed in Figure 4. The goal of these algorithms is to reduce the search space by transforming the pieces into rectangles before sending them to the solver. Each preprocessing method receives the same input (assuming that it is working with the same instance), this being the set of pieces $\mathcal{P}$ and their dimensions. They all also return an output of the same form: First, each preprocessing returns two arrays of the same length that contain the dimensions of the newly created rectangular shapes, W and H. Second, they also return an array named SUB, which will be explained in more detail in Section 5.1.2. Note that the preprocessing presented in Section 5.1.4 introduces a way to further decompose predefined insulation pieces, which has significant implications for the definition of the problem in itself. We allow the Slope mapping + preprocessing to decompose the non-rectangular pieces in such a way that they can be represented as a rectangle, reducing the losses induced from the piece's slope to zero. These pieces are subdivided into exactly three new smaller pieces. Subdividing further could allow for slightly better packing, but is not allowed since the factory wants to limit the necessary cutting to a minimum.

5.1.1 Bounding box

The Bounding box preprocessing is depicted in Figure 4a. Each piece $p \in \mathcal{P}$ is transformed into the smallest rectangle that can fully contain it. The width of the new rectangle is equal to w_p, while its height is set to $hr_p + ht_p$. Of course, while being fast and easy to comprehend, this preprocessing generates a substantial amount of scraps when working with trapezoid pieces, since treating them as their bounding rectangles takes up more space than they really occupy. Pieces with a triangular height ht_p of 0 do not create additional losses.



(a) Bounding box. **(b)** Staircase.

(c) Slope mapping. **(d)** Slope mapping +.

Figure 4 Preprocessing algorithms examples.

5.1.2 Staircase

The Staircase preprocessing goes one step further in the same direction as Bounding box did and creates multiple bounding boxes for each piece that has a triangular height component $ht_p > 0$, reducing the amount of wasted space. Figure 4b shows this reduction by highlighting the way it forces the model to consider pieces. Each of these pieces have their width w_p divided into sub subdivisions of equal width $\left\lfloor \frac{w_p}{sub} \right\rfloor$, with any remainder w_p mod sub added to the last segment. The algorithm calculates the bounding box for each subdivision using the original dimensions of the shape. For each subdivision, the starting and ending points relative to the original width of the shape are known. It is possible to use the heights of the piece hr_p and ht_p together with this information to calculate their respective heights. This calculation is shown in the following:

Suppose that the triangular height ht_p of a piece is always pointing upward and that its bottom left corner is at the origin $(0, 0)$. The piece is also flipped so that its upper corners are at coordinates $(0, hr_p)$ and $(w_p, hr_p + ht_p)$. Each subdivision has the x position of their right boundary referred to as rb, which always has a value between 0 and w_p (included). The following expression is then used to calculate the height of each subdivision: $\left\lceil hr_p + \frac{rb}{w_p} \cdot ht_p \right\rceil$. The bounding boxes are always high enough to contain their entire subdivision, as the value of rb is always greater than the value of the x position of the left boundary for each subdivision.

Rectangular pieces that have a triangular height ht_p of 0 are treated differently. These pieces are not subdivided, so their bounding box is the same as the piece itself. The SUB array encodes the subdivision information for the model. In short, SUB is an array of length $|\mathcal{P}|$, where SUB_p represents the number of subdivisions in which the piece p has been split, $\forall p \in \mathcal{P}$. Since the pieces are always divided into the same number of subdivisions, the only two possible values in SUB are sub, if the piece is a triangle or a trapezoid, or 1 if the piece is a rectangle. Also note that an important constraint is added to the model when working with subdivisions. Since the shape is not split into smaller parts but is only represented as a combination of bounding boxes, it is important that these boxes remain connected when being translated or rotated. These connections between the different subdivisions make it so that the piece can be rotated in the eight possible rotations mentioned in Section 4 (each individual subdivision only truly rotates 0 or 90 degrees as they are rectangular, but the connection constraints model the eight feasible rotations).

5.1.3 Slope mapping

The Slope mapping preprocessing acts as an extension of the simple Bounding box approach by attempting to merge pieces that have an identical triangular height and width into one rectangular piece. Figure 4c shows an example of two pieces that are matched together as a singular rectangle, and the other piece, being alone, is instead considered as its bounding box. For each piece $p_1 \in \mathcal{P}$, if $ht_{p_1} \neq 0$, the algorithm searches for another piece p_2 with the same width and triangular height that has not yet been matched. If a match is found between the pieces p_1 and p_2, they are stacked to create one larger rectangle, provided that their combined height $hr_{p_1} + hr_{p_2} + ht_{p_1}$ fits within the predefined dimensions of a new sheet. If no match is found, p_1 is transformed in the same way as it would have been by the Bounding box approach. This approach has the potential to remove the wasted space from two pieces that have the same triangular height component each time a match is found.

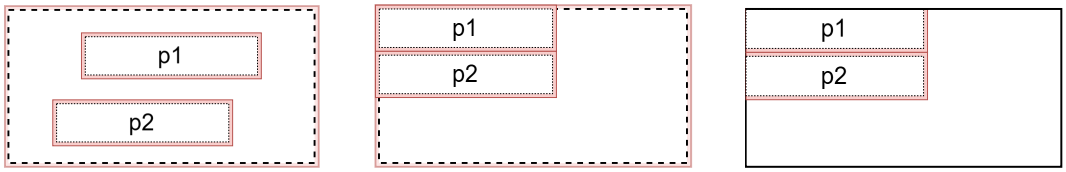
5.1.4 Slope mapping +

The Slope mapping + preprocessing refines the basic Slope mapping approach by further addressing pieces that still inherently create losses when sent to the model, those being pieces with triangular components that cannot be paired with a matching piece. Each piece $p \in \mathcal{P}$ that matches this condition is decomposed by the algorithm into three smaller shapes. In other words, each trapezoidal shape is converted into one rectangle, one triangle, and another triangular or trapezoidal shape. The pieces *p3.1*, *p3.2*, and *p3.3*, as shown in Figure 4d, are the result of the decomposition of piece *p3* in Figure 3.

Applying the same assumptions made for the Staircase cutting phase, the piece can be decomposed in a way that makes it reconfigurable as a rectangle, preventing any left losses. First, two cuts are made, one at the coordinate $(\lfloor \frac{w_p}{2} \rfloor, 0)$ going up to $(\lfloor \frac{w_p}{2} \rfloor, hr_p + \left\lceil \frac{ht_p}{2} \right\rceil)$, and the other at $(\lfloor \frac{w_p}{2} \rfloor, hr_p + \left\lceil \frac{ht_p}{2} \right\rceil)$ going right to $(w_p, hr_p + \left\lceil \frac{ht_p}{2} \right\rceil)$. Following these cuts, the new pieces are considered to be two distinct rectangles. The first rectangle *p3.1* has a width of $\lceil \frac{w_p}{2} \rceil$ and a height of $hr_p + \left\lceil \frac{ht_p}{2} \right\rceil$. The second is made up of the pieces *p3.2* and *p3.3* stacked, creating a rectangle that has the same dimensions as the first. Although it might not be ideal to cut a piece into three smaller pieces, the insulation pieces context allows it. The Slope mapping + preprocessing is the only one among the four methods we propose that leads to the cutting of a piece into smaller ones.

5.2 Preprocessing – Cutting margins

Before sending the data to the first optimization model, the dimensions of the sheets and pieces are increased so that the cutting margins are taken into account. Insulation pieces are cut using a $\frac{1}{8}''$ wide water jet. This is modeled by increasing the width and height of all (now rectangular) insulation pieces and insulation sheets by $\frac{1}{8}''$. All pieces now contain half the width of the jet ($\frac{1}{16}''$) as margins on each of their edges. Two touching pieces will always at least be distanced by their combined margins (equal to the width of the jet), so the water jet can directly cut between them and never impact the true sizes of the pieces. Similarly, pieces on the edge of a sheet benefit from its increased dimensions. The placement finished, the added dimensions of the sheet are removed, effectively removing the margins of the pieces that were placed on its side. Figure 5 shows the effect of the added margins, where 5a shows the addition of the margins to the pieces and sheets, 5b shows two pieces with their margins touching and touching the edges of the resized sheet, and 5c shows the final placement of the pieces, after the removal of the margins of the sheets.



(a) Addition of the margins. **(b)** Effects of the margins. **(c)** Sheet margins removal.

Figure 5 Preprocessing – Adding the margins to the pieces and sheets.

5.3 Objective functions / models

The preprocessing step finished, it is possible to send the dimension values of the pieces and sheets as well as the subdivision data to the first constraint model, thus starting the optimization process. As mentioned earlier, two models are used sequentially. The first minimizes the total area of the used sheets, prioritizing the use of scraps over new sheets. The second optimizes the shape of the scraps. All preprocessing output their data in a way that can be used by the first constraint model. This model, in turn, outputs its data in a way that is usable by the second model.

The use of two sequential models is justified by the computation time. In essence, the solver used by the first model computes the optimal sheets to use before sending them to the next one. The second model then optimizes the reusability of the scraps without having to think about reducing the total area of the used sheets again. Optimizing those two objectives at the same time in a unique model does not work as efficiently because the solver spends a lot of time optimizing the reusability of scraps even before the optimal sheets are found.

5.3.1 Sheets minimization model

From the selected preprocessing output, we receive the following: **i)** The array SUB of length $|\mathcal{P}|$ indicates the number of subdivisions for a piece. For each piece $p \in \mathcal{P}$, we have $SUB_p = sub$ subdivisions if the piece p was subdivided by Staircase and $SUB_p = 1$ if not. We define the set $\mathcal{D} = \{1, \ldots, \sum_{p \in \mathcal{P}} SUB_p\}$ to be the indices of all subdivisions. **ii)** The arrays W and H, each of length $|\mathcal{D}|$, give the width and height of each subdivision. Pieces that were not subdivided are still considered a subdivision, which means that the dimensions of these pieces will also be present in W and H as a single subdivision. We also have **iii)**, the sheet widths SW and the sheet heights SH which are arrays of length $|\mathcal{S}|$. As stated earlier, the sheets in $\mathcal{S}$ are ordered by non-decreasing area, meaning that $\forall s \in \mathcal{S}$ where $s > 1, SW_s \cdot SH_s \geq SW_{s-1} \cdot SH_{s-1}$. The first model is described below and uses the constants and variables presented in tables 1 and 2.

Table 1 Names, values and descriptions of model constants.

Name	Value	Description
SA_s	$SW_s \cdot SH_s, \forall s \in \mathcal{S}$	Area of the sheet s (already ordered by non-decreasing area)
SL_s	$\sum_{i=1}^{s-1} SW_i, \forall s \in \mathcal{S}$	X position of the left edge of sheet s
I_p	$1 + \sum_{i=1}^{p-1} SUB_i, \forall p \in \mathcal{P}$	Index of the first subdivision of the piece p
R	$\lvert\{s \in \mathcal{S} \mid SA_s < \max(SA)\}\rvert$	The number of sheets that are scraps

Table 2 Names, domains and descriptions of model variables.

Name	Domain	Description
$sheet_d$	$\mathcal{S}, \forall d \in \mathcal{D}$	The sheet in which the d subdivision is nested
l_d	$\{0, \ldots, \sum_{s \in \mathcal{S}} SW_s\}, \forall d \in \mathcal{D}$	X position of the left edge of subdivision d
r_d	$\{0, \ldots, \sum_{s \in \mathcal{S}} SW_s\}, \forall d \in \mathcal{D}$	X position of the right edge of subdivision d
b_d	$\{0, \ldots, \max(SH)\}, \forall d \in \mathcal{D}$	Y position of the bottom edge of subdivision d
t_d	$\{0, \ldots, \max(SH)\}, \forall d \in \mathcal{D}$	Y position of the top edge of subdivision d
rw_d	$\{0, \ldots, \max(SW)\}, \forall d \in \mathcal{D}$	Width of subdivision d after rotation
rh_d	$\{0, \ldots, \max(SH)\}, \forall d \in \mathcal{D}$	Height of subdivision d after rotation
rot_d	$\{1, \ldots, 8\}, \forall d \in \mathcal{D}$	Rotation of subdivision d
ta	$\{0, \ldots, \sum_{s \in \mathcal{S}} SA_s\}$	Total area of the used sheets.

5.3.1.1 Constants

We position the sheets on the Cartesian plane, side by side from left to right, in non-decreasing order of area (in the order in which they are in $\mathcal{S}$). For each sheet $s \in \mathcal{S}$, SA_s denotes the area of s, and SL_s holds the x position of its left edge. For each piece $p \in \mathcal{P}$, I_p is the index of the first subdivision belonging to p. The constant R contains the number of sheets that are scraps, i.e. sheets whose area is smaller than the area of the largest sheet. The constants and their values are summarized in Table 1.

5.3.1.2 Variables

For each subdivision $d \in \mathcal{D}$, we have the decision variables l_d and b_d that are the coordinates of the lower left corner of the subdivision and the variable rot_d that encodes one of the eight possible rotations (four if d is flipped, four if not). In addition to the decision variables, there are *functional variables*. The values of these variables can be fully determined once the values of the decision variables are known. These variables are, for each subdivision $d \in \mathcal{D}$, the coordinates (r_d, t_d) of the upper-right corner of the subdivision, the sheet number $sheet_d$ on which lies the subdivision, the width rw_d and height rh_d of the subdivision after being rotated. Furthermore, the variable ta is used to represent the total area of the used sheets in the solution. The variables and their domains are summarized in Table 2.

5.3.1.3 Constraints

The following constraints ensure that subdivisions are fully nested inside a single sheet and that subdivisions do not overlap each other.

Constraints (1) to (3) apply to each piece $p \in \mathcal{P}$ and its subdivisions d. Constraint (1) applies to pieces that have a single subdivision and, therefore, are rectangular. It ensures that the rotation is either 0° or 90°. Constraints (2) and (3) consider the current rotation of subdivision d and fix the width and height of the subdivision.

$$SUB_p = 1 \implies rot_d \leq 2 \qquad \forall p \in \mathcal{P}, d \in \{I_p, \ldots, I_p + SUB_p - 1\} \qquad (1)$$

$$rot_d \bmod 2 = 0 \implies \begin{cases} rw_d = H_d \\ rh_d = W_d \end{cases} \qquad \forall d \in \mathcal{D} \qquad (2)$$

$$rot_d \bmod 2 \neq 0 \implies \begin{cases} rw_d = W_d \\ rh_d = H_d \end{cases} \qquad \forall d \in \mathcal{D} \qquad (3)$$

Constraint (4) links the left-side coordinate, the right-side coordinate and the width of a subdivision together. The same is true for the top and bottom coordinates and the height of the subdivision. This constraint takes into account the rotation of the piece.

$$r_d = l_d + rw_d \wedge t_d = b_d + rh_d \qquad \forall d \in \mathcal{D} \tag{4}$$

Constraints (5) to (7) have two purposes: 1) to force each subdivision $d \in \mathcal{D}$ to be fully contained in one sheet and 2) to assign the sheet in $\mathcal{S}$ on which subdivision d lies to the variable $sheet_d$.

$$l_d \geq SL_{sheet_d} \qquad \forall d \in \mathcal{D} \tag{5}$$

$$r_d \leq SL_{sheet_d} + SW_{sheet_d} \qquad \forall d \in \mathcal{D} \tag{6}$$

$$t_d \leq SH_{sheet_d} \qquad \forall d \in \mathcal{D} \tag{7}$$

Constraint (8) links the subdivisions of a piece $p \in \mathcal{P}$. It ensures that the subdivisions are assigned to the same sheet and the same rotation. Finally, it ensures that the subdivisions are positioned side by side. The right edge of one subdivision coincides with the left edge of the other subdivision, or the top edge of one subdivision coincides with the bottom edge of the other subdivision. The constraint takes into account the eight combinations of rotations and flips.

$$\begin{aligned}
&\forall p \in \mathcal{P}, d \in \{I_p, \ldots, I_p + SUB_p - 2\} : \\
&\quad sheet_d = sheet_{d+1} \wedge rot_d = rot_{d+1} \wedge \\
&\quad \begin{pmatrix} rot_d \bmod 4 = 1 \rightarrow r_d = l_{d+1} \\ rot_d \bmod 4 = 2 \rightarrow t_d = b_{d+1} \\ rot_d \bmod 4 = 3 \rightarrow l_d = r_{d+1} \\ rot_d \bmod 4 = 0 \rightarrow b_d = t_{d+1} \end{pmatrix} \wedge \begin{pmatrix} rot_d \in \{1,7\} \rightarrow b_d = b_{d+1} \\ rot_d \in \{2,8\} \rightarrow r_d = r_{d+1} \\ rot_d \in \{3,5\} \rightarrow t_d = t_{d+1} \\ rot_d \in \{4,6\} \rightarrow l_d = l_{d+1} \end{pmatrix}
\end{aligned} \tag{8}$$

The constraint DiffN ensures that the subdivisions do not overlap on each other. This constraint has in its scope the positions of the subdivisions and their dimensions. We also use the Cumulative constraint. It is redundant but comes with more filtering algorithms that improve the performance of the model. The Cumulative constraint only considers the positions of the subdivisions on the x-axis, as the range defined by the y-axis is not large enough for its filtering algorithms to offer any improvements.

$$\text{DiffN}(l, b, rw, rh) \tag{9}$$

$$\text{Cumulative}(l, rw, rh, \max(SH)) \tag{10}$$

Constraint (11) constrains ta, which is the objective variable to minimize, to be equal to the total area of the sheets used by the solution. The constant R indicates the number of scrap sheets that are available for use. These scrap sheets have smaller areas than the new sheets, they are therefore labeled with indices smaller than or equal to R. The summation in (11) adds the area of the used scrap sheets. The second term computes the area of new sheets. Since all have the same area, we simply multiply the area of new sheets, $\max(SA)$, by the number of new sheets used, $\max(sheet) - R$. Using $\max(sheet) - R$ to encode the number of used new sheets forces the solver to break symmetries by assigning pieces to new sheets with lower indices. We found that using this formulation instead of explicitly specifying a value precedence constraint among the new sheets offered equivalent results while being slightly faster.

$$ta = \left(\sum_{\substack{s \in \{1,\ldots,R\} \\ \exists d \in \mathcal{D}, sheet_d = s}} SA_s \right) + \max(SA) \cdot (\max(sheet) - R) \tag{11}$$

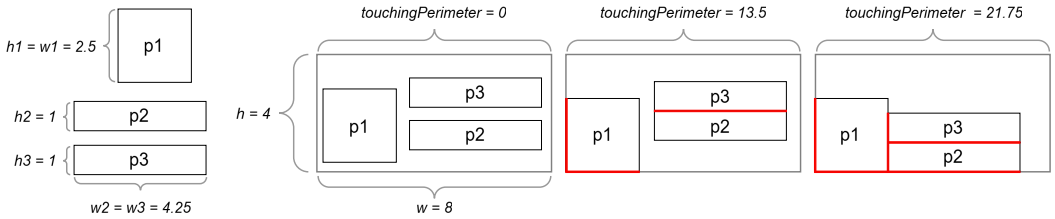


Figure 6 Computing the touching perimeter metric.

Constraint (12) breaks symmetry between similar solutions where the pieces nested in a new sheet can be swapped for the pieces nested in another new sheet. This symmetry occurs less often with scrap sheets, as their area can differ. The constraint states that the first piece can only be nested on the first new sheet, the second piece on the first two sheets, and so on.

$$sheet_d \leq p + R \qquad \forall p \in \mathcal{P}, d \in \{I_p, \ldots, I_p + SUB_p - 1\} \tag{12}$$

We implemented this model in MiniZinc using the Chuffed solver [10], which offers the priority search [13] annotation to model the branching heuristic.

```
solve::priority_search(l,
   [int_search([rot_d, l_d, b_d], input_order, indomain_min) |  d ∈ 𝒟 ],
smallest, complete) minimize ta;
```

The priority search selects the subdivision that can be placed the farthest to the left on the sheet with the smallest index. Once the subdivision is selected, it fixes its rotation and position (in that order). The solver also uses solution-based phase saving [11] as it offers a significant speed increase in the time needed to compute optimality.

Using a contiguous coordinate system for packing the pieces on the sheets instead of the classic method where each sheet has its own coordinate system avoids the need for optional variables, hence simplifying the way the problem is modeled.

5.3.2 Scrap reusability maximization model

The subsequent model uses the same base as the first model (Section 5.3.1). We only change the objective function, some data given as input, and the branching heuristic.

We introduce the new objective function, which is inspired by the *touching perimeter* heuristic introduced by Lodi et al. [19]. The intuition is to maximize the length of the perimeter of the subdivisions that touches either the side of a sheet or the side of another subdivision. Maximizing this value is equivalent to maximizing the density of the pieces in the sheet. This also indirectly concentrates the waste in each sheet, increasing the size of the scraps and their reusability. Where Lodi et al. [19] use a greedy heuristic to try to maximize the touching perimeter, our objective maximizes it directly.

Figure 6 shows three pieces that need to be packed in the most efficient way. In the first sheet, those pieces float in the middle without touching any edges, resulting in an objective value of 0 and a suboptimal packing. In the second and last sheets, their placement is further optimized, and we can clearly see parts of their perimeter that come in contact. In this example, the last sheet shows the best way to place the three pieces, as it offers the best scrap reusability in terms of the pieces SOKÏO often produce.

The objective variable of the total used sheet area ta is discarded and replaced by the touching perimeter variable tp, whose domain is $\{0, \ldots, 2 \cdot \sum_{d \in \mathcal{D}}(W_d + H_d)\}$ and whose value should be maximized. Constraint (13) constrains tp to be equal to the total touching perimeter of the subdivisions where the notation $[\![b]\!]$ returns 1 if b is *true* or 0 otherwise. Specifically, the first summation computes the perimeter of the subdivisions that are in contact with the edges of the sheet in which they are nested, while the second summation computes the perimeter of the subdivisions that are in contact with the edges of other subdivisions. The only pairs of subdivisions that are considered are those that are on the same sheet. Each pair is considered at most once ($d_1 < d_2$), which means that the touching perimeter will only be counted once, when both touching edges should be taken into account. This explains the multiplication by two, which makes it so that the touching edges of the subdivisions have their touching perimeter added correctly.

$$\begin{aligned} tp = \sum_{d \in \mathcal{D}} & \Big([\![l_d = SL_{sheet_d}]\!] \cdot rh_d + [\![r_d = SL_{sheet_d} + SW_{sheet_d}]\!] \cdot rh_d + \\ & [\![b_d = 0]\!] \cdot rw_d + [\![t_d = SH_{sheet_d}]\!] \cdot rw_d \Big) + \\ 2 \times \sum_{\substack{d_1, d_2 \in \mathcal{D} \\ d_1 < d_2 \\ sheet_{d_1} = sheet_{d_2}}} & \Big([\![r_{d_1} = l_{d_2} \vee r_{d_2} = l_{d_1}]\!] \cdot \max(0, \min(t_{d_1}, t_{d_2}) - \max(b_{d_1}, b_{d_2})) + \\ & [\![t_{d_1} = b_{d_2} \vee t_{d_2} = b_{d_1}]\!] \cdot \max(0, \min(r_{d_1}, r_{d_2}) - \max(l_{d_1}, l_{d_2})) \Big) \end{aligned} \tag{13}$$

The input of this model is still a list of subdivisions and sheets, but we only feed this second model with the sheets that were assigned pieces by the first model. This reduces the search space and ensures that the total area of the used sheets remains optimal.

To solve this model, we use the CP-SAT solver [23] using four threads (using six threads did not improve performance in any way, and more than six worsened it). CP-SAT does not offer solution-based phase saving and does not offer a priority search. Despite that, we have found that CP-SAT is faster than Chuffed for this specific model. The second model uses this specific search annotation that selects the variable in the l and b arrays with the smallest value in its domain and branches on that value.

`solve::int_search(` $l \parallel b$`, smallest, indomain_min) maximize` tp`;`

6 Experimentations

6.1 Benchmarks

This section benchmarks both models presented in this paper and compares our solutions with those of SOKÏO, for a particular instance. All experiments were run on a computer with a 6-core Intel Core i7-10750H CPU @ 2.60 GHz and 16 GB of memory.

6.1.1 Sheets minimization model

The experiments consisted of four instances ranging from 15 to 188 pieces, which were analyzed with each of the preprocessing methods. For the Staircase preprocessing, we experimented with 2 and 4 as values for the number of subdivisions *sub*. Each of these tests was also performed with 3 different sets of scrap sheets $\mathcal{R}$. The first one contains no scrap, so only new sheets can be used. The other ones contain 30 and 60 scraps of different sizes.

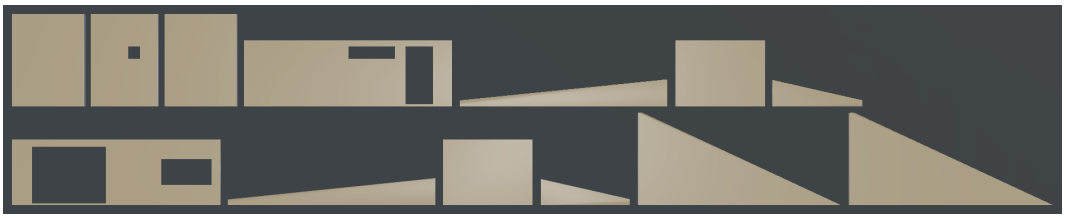


Figure 7 CLT panels breakdown of instance 2.

We use three key metrics to compare the different experiments made with the first model: 1) the number of new sheets used, 2) the waste ratio, expressed as a percentage of the total used sheets area. This is calculated using the following expression:

$$\left(1 - \frac{\text{Total area of the pieces (before the preprocessing)}}{\text{Total area of the used sheets}}\right) \cdot 100\%,$$

as well as 3) the time required to compute the solution, which is by far the most valuable resource for SOKÏO.

Our results will be compared with those obtained by SOKÏO on instance 2, which represents the only structure currently in production. This structure is composed of prefabricated panels, which are shown in Figure 7. SOKÏO, exactly as we now do in an automated way, had to compute the best shapes, sizes, placements, etc. of the insulation pieces so that they would completely cover the structure's panels, excluding the different openings. Their manual computation landed them with a set of pieces that were then nested in a total of 46 new sheets. Note that no scraps were used to create these pieces or kept after the cutting process due to their suboptimal sizes. SOKÏO's waste metric on this specific instance is 21.58%, and their manual nesting took an entire week to complete.

Table 3 shows our results for the instance 2 produced by SOKÏO, as well as for the others. For this model, all instances were subject to a 5-minute timeout. For instance 2, we were able to calculate a solution that is much more efficient than what SOKÏO achieved manually on site, with an optimal solution of 39 new sheets obtained in 2.91 seconds using the Slope mapping + preprocessing. This equates to a material waste of 6.71%. Compared to SOKÏO's solution of 46 new sheets and 21.58% wasted material (which took a week of planning). Looking at these results, it is safe to say that the two objectives that were a priority for SOKÏO, reducing the amount of time needed to obtain the solutions and improving their quality based on the number of new sheets used, have been achieved. Figures 8 and 9 in Appendix A show visualizations of the scrap minimization process for instance 2 using the Bounding box and Slope mapping + preprocessings. All instances in Table 3 are feasible.

Although SOKÏO did not yet work on the other instances, we can assume that the time needed to compute their solution, as well as the number of new sheets used, will be much lower in the results provided by the first model than the manual results SOKÏO would have obtained if they had done so. The models can also work with scraps. This allows SOKÏO to recycle them, which they could not even consider before due to their small sizes in the solution they had. Table 3 shows that while instances that contain scraps take more time to compute to optimality, they offer even better solutions both in terms of the new sheets used and in the amount of waste generated (which is computed on all sheets used, not only new ones). The results also show that the Slope mapping + preprocessing is the best of the four we presented on these metrics. This is easily explained by the fact that the subdivisions generated by Slope mapping + do not contain any wasted area, while those generated by the

Table 3 Model 1 – Performance Metrics Across All Instances and Preprocessing Methods.

Preprocessing	$\lvert\mathcal{D}\rvert$	New Sheets [0/30/60]	Waste (%) [0/30/60]	Time (s)[1] [0/30/60]
Instance 1 (15 Pieces)				
Bounding box	15	9 / 7 / 7	20.67 /13.13/10.51	0.51 / 0.60 / 0.74
Staircase - *sub* = 2	21	9 / 7 / 7	20.67 /13.13/10.51	0.63 / 0.78 / 0.92
Staircase - *sub* = 4	33	8 / 8 / 7	**10.76**/10.76/10.51	0.83 / 1.27 / 1.59
Slope mapping	15	9 / 7 / 7	20.67 /13.13/10.51	0.47 / 0.60 / 0.69
Slope mapping +	21	8 / 6 / 5	**10.76**/ **7.51** / **5.01**	0.71 / 1.01 /10.23
Instance 2 (74 Pieces)				
Bounding box	74	44 / 42 / 39	17.31 /16.42/14.11	2.17 / 4.88 /64.77
Staircase - *sub* = 2	94	43 / 42 / 38	15.38 /15.14/13.86	6.21 / 8.68 / –
Staircase - *sub* = 4	134	43 / 42 / 38	15.38 /14.39/12.10	12.96/19.36/ –
Slope mapping	67	42 / 40 / 36	13.37 /11.73/ 9.29	1.90 / 3.45 /13.80
Slope mapping +	73	39 / 37 / 35	**6.71** / **5.34** / **4.30**	2.91 / – / –
Instance 3 (106 Pieces)				
Bounding box	106	65 / 63 / 60	12.26 /11.32/ 9.46	– / – / –
Staircase - *sub* = 2	114	65 / 62 / 60	12.26 /11.10/ 9.69	– / – / –
Staircase - *sub* = 4	130	65 / 62 / 59	12.26 /10.58/ 9.05	– / – / –
Slope mapping	106	65 / 63 / 57	12.26 /11.32/ 9.46	– / – / –
Slope mapping +	114	63 / 60 / 59	**9.47** / **7.41** / **7.18**	– / – / –
Instance 4 (188 Pieces)				
Bounding box	188	123/120/ 118	10.29 / 8.90 / 8.34	– / – / –
Staircase - *sub* = 2	214	122/119/ 116	9.55 / 8.71 / 8.03	– / – / –
Staircase - *sub* = 4	266	122/119/ 118	9.55 / 8.83 / 8.19	– / – / –
Slope mapping	188	123/119/ 117	10.29 / 8.90 / 8.34	– / – / –
Slope mapping +	214	116/115/ 112	**4.87** / **4.44** / **4.52**	– / – / –

Values in columns represent results for instances where $\mathcal{S}$ contains [0 scraps / 30 scraps / 60 scraps].
Lowest waste percentage per scrap group within each instance highlighted in bold.
[1] "–" indicates the 300-second time limit was reached.

other three preprocessings on triangular and trapezoid pieces do. As mentioned above, Slope mapping + cuts some pieces into three smaller pieces, and the current context allows it. For a problem where this behavior is not allowed, the Slope mapping and Staircase preprocessings also offer great results, especially when the sheet set $\mathcal{S}$ contains reused scraps.

6.1.2 Scrap reusability maximization model

The benchmark for the second model consists of 12 instances obtained by testing each of the 4 base instances from model 1 with the 3 different numbers of scrap sheets, $|\mathcal{R}| \in \{0, 30, 60\}$. These new instances are based on the results provided by the best preprocessing in Table 3. In all cases, the best preprocessing (based on the waste ratio metric) was Slope mapping +, so all 12 runs were performed using the results of this preprocessing. Table 4 shows the results obtained for these 12 experiments after a 30-minute timeout (which was reached in all instances). A higher touching perimeter percentage (TPP) means a higher density of the pieces in each sheet, which translates into better packing and more reusable scraps. A TPP

Table 4 Model 2 – Touching Perimeter Ratio (Solutions from model 1 using Slope mapping +).

Instance	Touching Perimeter (%)[1]		
	0 Scraps	30 Scraps	60 Scraps
Instance 1 (15 Pieces)	81.71	77.68	77.88
Instance 2 (74 Pieces)	78.82	77.43	77.54
Instance 3 (106 Pieces)	83.68	–	–
Instance 4 (188 Pieces)	73.00	–	–

[1] "–" indicates no result was obtained before reaching the 30-minute time limit.

of 100% could only be obtained with a waste ratio of 0%. Bigger instances generally provide a higher TPP, because the larger number of possible combinations yields a higher probability that efficient packing can be achieved. Figure 10 in Appendix A shows visualizations of the scrap reusability maximization process for instance 2, using the result from the Slope mapping + preprocessing in Figure 9. All instances in Table 4 are feasible.

Table 4 shows that the model generates initial solutions quickly (and keeps increasing their quality) for small instances that reuse no scraps, but takes a longer time for larger instances that try to reuse many scraps. Of course, the objective of the second model is way more complex than the one of the first model, and large instances suffer from that.

When time is not an issue, waiting for the solver to output solutions for the second model is clearly beneficial. After 16 hours, the solver achieved a TPP of 88.04% on the 60-scrap version of instance 4, which is by far the largest and most complex. Although 16 hours may seem long, it is still a short amount of time compared to the manual results of SOKÏO. Where SOKÏO took a week to compute their results for instance 2 and did not generate any reusable scraps, we are able to guarantee that our solution for instance 4 (which is more than twice the size of instance 2) obtained in 16 hours will result in scraps with great reusability.

6.2 A word on the dataset

All the different instances of the problem considered in this paper were created using the product configurator mentioned in Section 3. This software was developed using the Unity® game engine, and allows the user to edit the model of a structure to his/her liking before ordering it. This offers a load of possibilities in regards to industry 4.0 parametric product configuration, at the cost of needing to optimize each newly created building instance and its related parts, such as insulation cutting and nesting, as discussed.

7 Conclusion

We introduced four different preprocessing methods that work with trapezoidal shapes in the bin packing problem, using solvers that typically only work with rectangular shapes. We compared the nesting produced by our algorithms with the one manually obtained by SOKÏO. Each preprocessing achieved better results than those provided by SOKÏO. We introduced a new objective function that optimizes scrap reusability. This objective refines the results returned by the sheet minimization model and proposes new solutions that make better use of the space given by the optimal sheets that were found, allowing SOKÏO to create more reusable scraps. Directions for future work include the implementation of a constraint described by Beldiceanu et al. [5] that directly supports trapezoidal shapes. We also aim to further explore the implementation of the touching perimeter objective function to speed up the convergence of the second model. Finally, we want to model scrap sheets of non-rectangular shapes by fixing virtual pieces at locations where the scrap cannot be used.

References

1 Ranga Prasad Abeysooriya. *Cutting patterns for efficient production of irregular-shaped pieces*. PhD thesis, University of Southampton, 2017. URL: https://eprints.soton.ac.uk/414693/1/23._Final_submission_of_thesis.pdf.

2 Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. In Jean-Paul Delahaye, Philippe Devienne, Philippe Mathieu, and Pascal Yim, editors, *JFPL'92, 1ères Journées Francophones de Programmation Logique, 25-27 Mai 1992, Lille, France*, volume 17, page 51, 1992. doi:10.1016/0895-7177(93)90068-A.

3 Brenda S. Baker, Edward G. Coffman Jr., and Ronald L. Rivest. Orthogonal packings in two dimensions. *SIAM J. Comput.*, 9(4):846–855, November 1980. doi:10.1137/0209064.

4 N Beldiceanu and E Contejean. Introducing global constraints in chip. *Mathematical and Computer Modelling*, 20(12):97–123, 1994. doi:10.1016/0895-7177(94)90127-9.

5 Nicolas Beldiceanu, Qi Guo, and Sven Thiel. Non-overlapping constraints between convex polytopes. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*, pages 392–407. Springer, Springer, 2001. doi:10.1007/3-540-45578-7_27.

6 Julia A. Bennell and José Fernando Oliveira. A tutorial in irregular shape packing problems. *J. Oper. Res. Soc.*, 60(S1):S93–S105, 2009. doi:10.1057/jors.2008.169.

7 David W. Cantrell, Erich Friedman, et al. Packing unit squares in squares: A survey and new results. *The Electronic Journal of Combinatorics*, 5(DS7), 1998. URL: https://www.combinatorics.org/files/Surveys/ds7/ds7v1-1998.pdf.

8 CH Cheng, BR Feiring, and TCE Cheng. The cutting stock problem—a survey. *International Journal of Production Economics*, 36(3):291–305, 1994. doi:10.1016/0925-5273(94)00045-X.

9 Henrik I. Christensen, Arindam Khan, Sebastian Pokutta, and Prasad Tetali. Approximation and online algorithms for multidimensional bin packing: A survey. *Comput. Sci. Rev.*, 24:63–79, 2017. doi:10.1016/j.cosrev.2016.12.001.

10 Geoffrey Chu. *Improving combinatorial optimization*. PhD thesis, University of Melbourne, Australia, 2011. URL: https://hdl.handle.net/11343/36679.

11 Emir Demirovic, Geoffrey Chu, and Peter J. Stuckey. Solution-based phase saving for CP: A value-selection heuristic to simulate local search behavior in complete solvers. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 99–108. Springer, Springer, 2018. doi:10.1007/978-3-319-98334-9_7.

12 Andreas A. Falkner, Alois Haselböck, Gerfried Krames, Gottfried Schenner, and Richard Taupe. Constraint solver requirements for interactive configuration. In Lothar Hotz, Michel Aldanondo, and Thorsten Krebs, editors, *Proceedings of the 21st Configuration Workshop, Hamburg, Germany, September 19-20, 2019*, volume 2467 of *CEUR Workshop Proceedings*, pages 65–72. CEUR-WS.org, 2019. URL: https://ceur-ws.org/Vol-2467/paper-12.pdf.

13 Thibaut Feydy, Adrian Goldwaser, Andreas Schutt, Peter J Stuckey, and Kenneth D Young. Priority search with minizinc. In *ModRef 2017: The Sixteenth International Workshop on Constraint Modelling and Reformulation*, 2017. URL: https://ozgurakgun.github.io/ModRef2017/files/ModRef2017_PrioritySearchWithMiniZinc.pdf.

14 M. R. Garey and David S. Johnson. "strong" np-completeness results: Motivation, examples, and implications. *J. ACM*, 25(3):499–508, July 1978. doi:10.1145/322077.322090.

15 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

16 Paul C Gilmore and Ralph E Gomory. A linear programming approach to the cutting-stock problem. *Operations research*, 9(6):849–859, 1961. doi:10.1287/opre.9.6.849.

17 Mikael Z. Lagerkvist, Martin Nordkvist, and Magnus Rattfeldt. Laser cutting path planning using CP. 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 790–804, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-40627-0_58.

18 Andrea Lodi, Silvano Martello, and Michele Monaci. Two-dimensional packing problems: A survey. *Eur. J. Oper. Res.*, 141(2):241–252, September 2002. doi:10.1016/S0377-2217(02)00123-6.

19 Andrea Lodi, Silvano Martello, and Daniele Vigo. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS J. Comput.*, 11(4):345–357, 1999. doi:10.1287/ijoc.11.4.345.

20 MiniZinc Documentation. *Redundant Constraints*. URL: https://docs.minizinc.dev/en/stable/efficient.html#redundant-constraints.

21 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, Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

22 Óscar Oliveira, Dorabela Gamboa, and Elsa Silva. An introduction to the two-dimensional rectangular cutting and packing problem. *Int. Trans. Oper. Res.*, 30(6):3238–3266, November 2023. doi:10.1111/itor.13236.

23 Laurent Perron and Vincent Furnon. Or-tools. URL: https://developers.google.com/optimization/.

24 Andreas Schutt, Peter J. Stuckey, and Andrew R. Verden. Optimal carpet cutting. 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 69–84, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-23786-7_8.

25 Helmut Simonis and Barry O'Sullivan. Search strategies for rectangle packing. In Peter J. Stuckey, editor, *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings*, volume 5202 of *Lecture Notes in Computer Science*, pages 52–66, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-85958-1_4.

26 The MiniZinc Team. *The MiniZinc Handbook*. URL: https://docs.minizinc.dev/en/latest/index.html.

27 Gerhard Wäscher, Heike Haußner, and Holger Schumann. An improved typology of cutting and packing problems. *Eur. J. Oper. Res.*, 183(3):1109–1130, December 2007. doi:10.1016/j.ejor.2005.12.047.

28 Linda L Zhang. Product configuration: a review of the state-of-the-art and future research. *International Journal of Production Research*, 52(21):6381–6398, November 2014. doi:10.1080/00207543.2014.942012.

A Additional visualizations

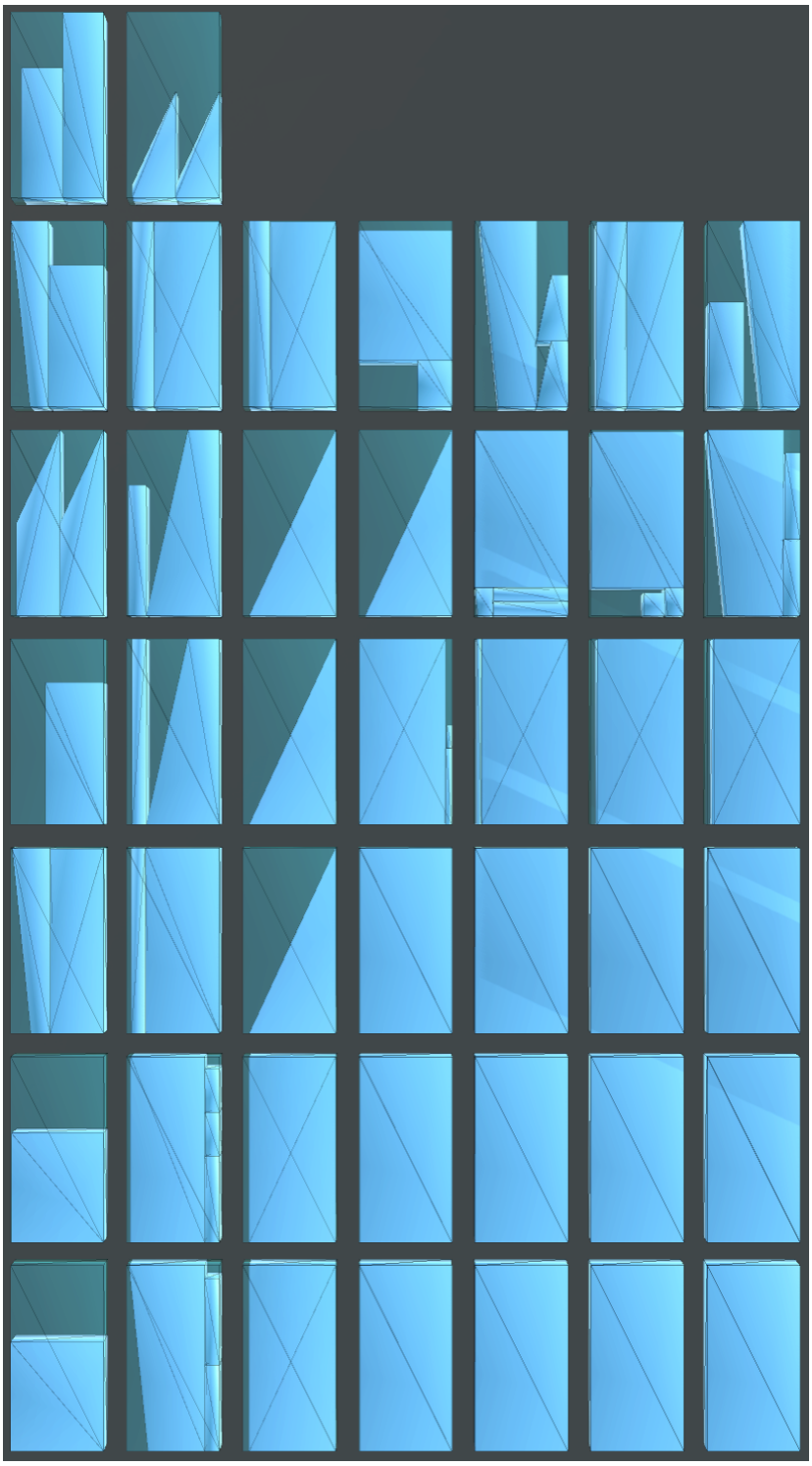


Figure 8 Solution obtained with Bounding box preprocessing on instance 2 - 44 new sheets.

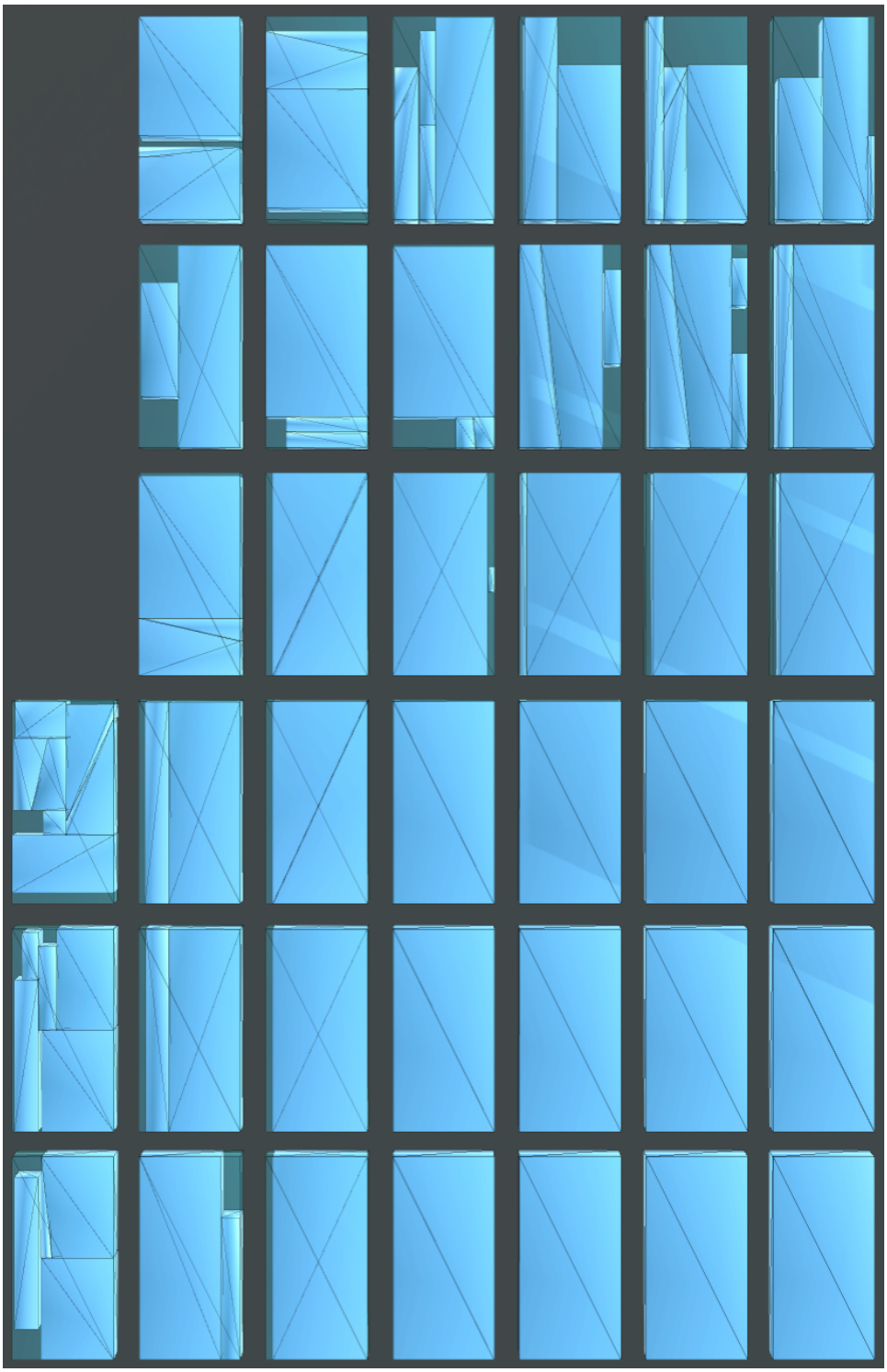


Figure 9 Solution obtained with Slope mapping + preprocessing on instance 2 - 39 new sheets.

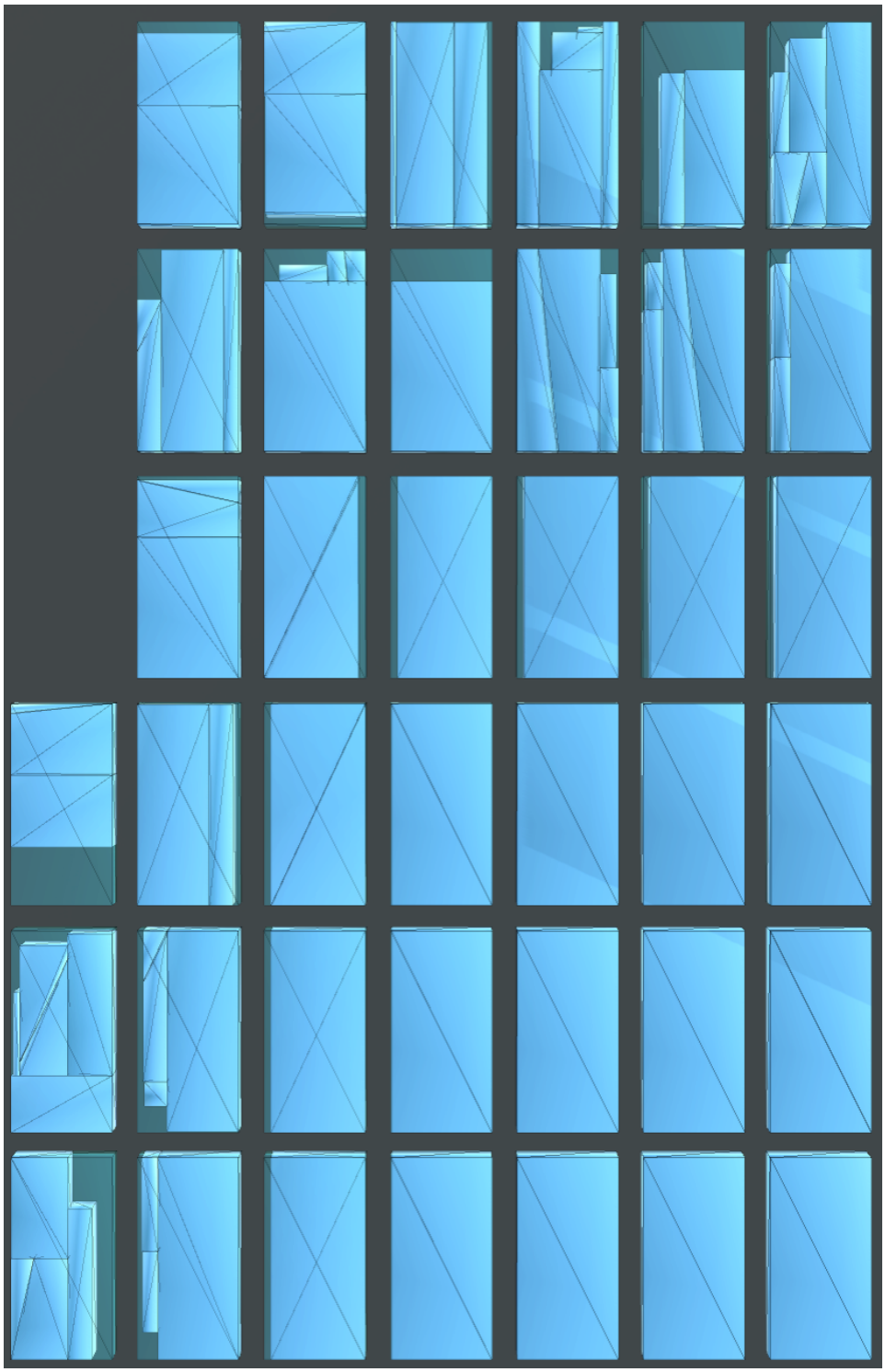


Figure 10 Results from Figure 9 after being optimized with the scrap reusability maximization model for one hour, with a final touching perimeter percent of 78.84 %.

Breaking Symmetries with Involutions

Michael Codish
Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel

Mikoláš Janota
CIIRC, Czech Technical University in Prague, Czech Republic

Abstract

Symmetry breaking for graphs and other combinatorial objects is notoriously hard. On the one hand, complete symmetry breaks are exponential in size. On the other hand, current, state-of-the-art, partial symmetry breaks are often considered too weak to be of practical use. Recently, the concept of graph patterns has been introduced and provides a concise representation for (large) sets of non-canonical graphs, i.e. graphs that are not lex-leaders and can be excluded from search. In particular, four (specific) graph patterns apply to identify about 3/4 of the set of all non-canonical graphs. Taking this approach further, we discover that graph patterns that derive from permutations that are involutions play an important role in the construction of symmetry breaks for graphs. We take advantage of this to guide the construction of partial and complete symmetry-breaking constraints based on graph patterns. The resulting constraints are small in size and strong in the number of symmetries they break.

2012 ACM Subject Classification Computing methodologies; Theory of computation → Constraint and logic programming

Keywords and phrases graph symmetry, patterns, permutation, Ramsey graphs, greedy, CEGAR

Digital Object Identifier 10.4230/LIPIcs.CP.2025.8

Funding The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project *POSTMAN* no. LL1902 and co-funded by the European Union under the project *ROBOPROX* (reg. no. CZ.02.01.01/00/22_008/0004590).

Acknowledgements The authors would like to thank the anonymous reviewers for their valuable comments and constructive feedback, which helped improve the quality and clarity of this paper. This work began at the *Dagstuhl-Seminar 23261 SAT Encodings and Beyond*.

1 Introduction

Graph search problems are about finding simple graphs with desired structural properties. Such problems arise in many real-world applications and are fundamental in graph theory. Solving graph search problems is typically hard due to the enormous search space and the large number of symmetries in graph representation: Any graph obtained by permuting the vertices of a solution (or a non-solution) is also a solution (or a non-solution), and is considered isomorphic, or "symmetric". The set of all such isomorphic graphs forms an isomorphism class. To optimize search, we aim to restrict it to focus on one "canonical" graph from each isomorphism class.

One common approach to eliminate symmetries is to add *symmetry breaking constraints* that are satisfied by at least one member of each isomorphism class [9, 24, 26]. A symmetry-breaking constraint is called *complete* if it is satisfied by exactly one member of each isomorphism class and *partial* otherwise. In many cases, symmetry-breaking constraints, complete or partial, are expressed in terms of "lex-constraints". Each lex-constraint corresponds to one symmetry, σ, which is a permutation on vertices, and restricts the search space to consider assignments that are lexicographically smaller than their permuted form obtained

© Michael Codish and Mikoláš Janota;
licensed under Creative Commons License CC-BY 4.0

according to σ. If one considers the set of lex-constraints corresponding to all n-factorial permutations, then the corresponding symmetry break is complete but too large to be of practical use.

Itzhakov and Codish [14] observe that a complete symmetry break can be defined in terms of a number of lex-constraints which is considerably smaller than n-factorial. They succeed in computing complete symmetry-breaking constraints of practical size for graphs with 10 or less vertices. Dančo et al. obtain similar results in the context of finite models [10]. But this approach does not scale. Itzhakov *et al.* [4] introduce the notion of "lex-implications", which are a refinement of lex-constraints. In the case of graphs, they compute complete symmetry breaks which are much smaller in size. But still, this approach does not scale beyond graphs with 11 vertices. It is known that breaking symmetry by adding constraints to eliminate symmetric solutions is intractable in general [1, 9]. So, we do not expect to find a complete symmetry break of polynomial size that identifies canonical graphs that are lex-leaders. Hence, the interest in partial symmetry breaks.

Codish *et al.* [7] introduce a partial symmetry break, which is equivalent to considering the quadratic number of permutations that swap a pair of vertices (transpositions). Rintanen *et al.* [23] enhance this approach for directed graphs. This approach is widely applied and turns out to work well in practice, despite eliminating only a small portion of the symmetries. However, when dealing with hard instances of graph search problems, this constraint does not suffice. Over the past decade, there has been little progress in the research of partial symmetry-breaking constraints for graph search problems. Some attempts are made in [15] and in [5]. However, there are no references in the literature to the applications that make use of the symmetry breaks defined in these papers.

In a recent paper, Codish and Janota [6] introduce a set-covering perspective on symmetry breaking. A permutation "covers" a graph if its application on the graph yields a smaller graph with respect to a given (lexicographic) order. A complete symmetry break is then a set of permutations that covers all non-canonical graphs. In that paper, the authors introduce the notion of a "pattern" which provides a concise representation for the set of graphs covered by a permutation.

In this paper, we pick up on the notion of "patterns" which we call here "graph patterns" and show how they can be used to provide complete and also partial symmetry breaks which improve on the current state-of-the-art. Motivated by the set-covering perspective presented in [6], we implement a greedy algorithm to cover the set of all non-canonical graphs using graph patterns. Although this approach does not scale, it provides a test bed to study the structure of the graph patterns selected in the greedy approach. We observe that the first four graph patterns selected in the greedy approach already cover 3/4 of the set of all graphs (and slightly more of the set of all non-canonical graphs). Our study also reveals the importance of a specific type of permutation when breaking symmetries on graphs. This is the class of permutations that are equal to their inverse, also called *involutions*. This class includes the classes of transpositions (which swap a pair of indices) and of consecutive transpositions (which swap a pair of consecutive vertices) that are at the basis of the partial symmetry breaks introduced in [8, 7].

We implement an algorithm that guides the search for a complete symmetry break based on the structure of graph patterns. Along the way, we obtain a chain of partial symmetry breaks of increasing precision. We present experimental results to evaluate these symmetry breaks. These results show that in the setting of a counterexample-guided abstraction-based (CEGAR) addition of symmetry breaks, focusing on "involutions first" reduces the number of iterations of the CEGAR-loop. Broadly speaking, this shows that it is beneficial to look for symmetry-breaking constraints systematically, rather than arbitrarily.

The rest of the paper is structured as follows. Section 2 provides definitions and notations that are used throughout. A series of examples are provided to demonstrate the concepts upon which we build. Section 3 builds on the set-covering approach to symmetry breaking and presents a greedy approach to cover all of the non-canonical graphs with graph patterns. While this approach does not scale, it provides a test bed with which to study the structure of the ("best") graph patterns selected. Section 4 applies the lessons learned from the greedy approach to guide a CEGAR-based algorithm to collect specific types of graph patterns that derive from various types of involutions. Section 5 describes a series of experiments in which we provide complete and partial symmetry breaks to a specific graph search problem and evaluate their performance. Finally, Section 6 concludes and presents future work.

2 Preliminaries and Notation

Representing Graphs. Throughout this paper, we consider simple graphs, i.e. undirected graphs with no self-loops. The adjacency matrix of a graph G is an $n \times n$ Boolean matrix. The element at row i and column j is *true* if and only if (i, j) is an edge. The list of edges of a graph G is denoted $edges(G)$. It consists of the $\binom{n}{2}$ elements obtained as the concatenation of the columns of the upper triangle of G (or any other predetermined order). In abuse of notation, we let G denote a graph in any of its representations. An *unknown graph* of order-n is represented as an $n \times n$ adjacency matrix of Boolean variables which is symmetric and has the values *false* (denoted by 0) on the diagonal, or as the corresponding list of edges.

Ordering Graphs. Let G_1, G_2 be known or unknown graphs with n vertices. Then, $G_1 \leq G_2$ if and only if $edges(G_1) \leq_{lex} edges(G_2)$ where $\leq_{lex}$ denotes the standard lexicographic ordering. When G_1 and G_2 are unknown graphs, then the lexicographic ordering, $G_1 \leq G_2$, can be viewed as specifying a *lexicographic order constraint* over the variables in G_1 and G_2. For Boolean strings $\bar{a} = \langle a_1, \ldots, a_m \rangle$ and $\bar{b} = \langle b_1, \ldots, b_m \rangle$ and $1 \leq i \leq m$, we denote

$$\bar{a} <^i_{lex} \bar{b} \Leftrightarrow \{\ (a_1 = b_1), \ldots, (a_{i-1} = b_{i-1}), (a_i = 0), (b_i = 1)\ \} \tag{1}$$

We denote $G_1 <^i G_2$ if $edges(G_1) <^i_{lex} edges(G_2)$.

Permutations. The group of permutations on $\{1 \ldots n\}$ is denoted S_n. We represent a permutation $\pi \in S_n$ as a sequence of length n where the ith element indicates the value of $\pi(i)$. For example: the permutation $[2, 3, 1] \in S_3$ maps as follows: $\{\ 1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 1\ \}$. We will refer to the following types of permutations:

Transpositions. A *transposition* is a permutation that swaps two elements and leaves all other in place. For example, $[4, 2, 3, 1]$ is the transposition that swaps elements 1 and 4. A transposition that swaps a pair of consecutive elements, is called a *consecutive transposition*. For example, $[1, 3, 2, 4]$ is a consecutive transposition.

Involutions. An *involution* is a permutation π such that $\pi \circ \pi$ is the identity. Involutions are permutations that swap any set of pairs of disjoint elements. For example, $[4, 3, 2, 1]$ is the involution which swaps the pairs $\{\ 1, 4\ \}$ and $\{\ 2, 3\ \}$. Transpositions are a special case of involutions. A *consecutive involution* is an involution that swaps any number of consecutive pairs. For example $[2, 1, 4, 3]$ is a consecutive involution. We say that an involution is *intersecting* if it swaps (among others) the pairs $\{\ i, j\ \}$ and $\{\ k, l\ \}$ with $i < j$ and $k < l$ such that the intervals $[i, j]$ and $[k, l]$ are intersecting. If an involution is not intersecting, then we say that it is *disjoint*. For example, $[3, 4, 1, 2]$ (in cyclic notation

$(1\ 3)(2\ 4)$) is a disjoint involution because the intervals $[1,3]$ and $[2,4]$ are intersecting. The number of involutions is given as the sequence `A000085` of the Online Encyclopedia of Integer Sequences. Note that an involution is always a composition of finitely many disjoint transpositions.

Permutations act on graphs and on unknown graphs in the natural way. For a graph and also for an unknown graph G, viewing G as an adjacency matrix, given a permutation $\pi \in S_n$, then $\pi(G)$ is the adjacency matrix obtained by mapping each element at position (i,j) to position $(\pi(i),\pi(j))$ (for $1 \leq i,j \leq n$). The composition of permutations is defined in the natural way. Two graphs G, H are *isomorphic* if there exists a permutation $\pi \in S_n$ such that $G = \pi(H)$. Note that applying an involution to the vertices of a graph (directed or undirected) induces an involution on its edges because $(\pi \circ \pi(i), \pi \circ \pi(j)) = (i,j)$ for any involution π.

Symmetry Breaks. A *symmetry break* for graph search problems is a predicate, $\psi(G)$, on a graph G, which is satisfied by at least one graph in each isomorphism class of graphs. If ψ is satisfied by exactly one graph in each isomorphism class, then we say that ψ is a *complete symmetry break*. Otherwise, it is *partial*.

Heule [13] defines the notion of *redundancy ratio*, which we denote $\rho(\psi)$, to measure the precision of ψ. This is the ratio between the number of graphs that satisfy $\psi(G)$ and the number of isomorphism classes. One can view $\rho(\psi)$ as the average number of graphs per isomorphism class that are not eliminated by ψ.

In our setting, the canonical graphs, are the minimal (lexleader) graphs of the isomorphism classes of graphs. The following example is adapted from [6] and demonstrates some of the concepts introduced so far. Notice that, in this paper, edge variables are ordered by columns.

▶ **Example 1.** The following depicts an unknown, order-4, graph G, its permutation $\pi(G)$, for $\pi = [1,2,4,3]$, and their representations as lists of edges. The lex-constraint $G \leq \pi(G)$ can be simplified as described by Frisch *et al.* [11] to: $\langle x_2, x_3 \rangle \leq_{lex} \langle x_4, x_5 \rangle$.

$$\mathbf{G} = \begin{bmatrix} 0 & x_1 & x_2 & x_4 \\ x_1 & 0 & x_3 & x_5 \\ x_2 & x_3 & 0 & x_6 \\ x_4 & x_5 & x_6 & 0 \end{bmatrix} \quad \pi(\mathbf{G}) = \begin{bmatrix} 0 & x_1 & x_4 & x_2 \\ x_1 & 0 & x_5 & x_3 \\ x_4 & x_5 & 0 & x_6 \\ x_2 & x_3 & x_6 & 0 \end{bmatrix} \quad \begin{array}{l} \text{edges}(G) = \langle x_1, x_2, x_3, x_4, x_5, x_6 \rangle \\ \text{edges}(\pi(G)) = \langle x_1, x_4, x_5, x_2, x_3, x_6 \rangle \end{array}$$

There are 64 graphs of order 4. Eleven of these are canonical and the other 53 are non-canonical. The canonical graphs, represented as lists of edges, are detailed below.

```
[0,0,0,0,0,0] [0,0,0,0,0,1] [0,0,0,0,1,1] [0,0,0,1,1,1] [0,0,1,0,1,1] [0,0,1,1,0,0]
[0,0,1,1,0,1] [0,0,1,1,1,1] [0,1,1,1,1,0] [0,1,1,1,1,1] [1,1,1,1,1,1]
```

Graph Patterns. A *graph pattern* is a partially instantiated graph G (some elements are variables) such that all instances of G are non-canonical. We typically represent a graph pattern G using the list notation edges(G).

▶ **Example 2.** The following six graph patterns describe all of the 53 non-canonical graphs of order 4.

```
[1,0,A,B,C,D] [A,1,0,B,C,D] [A,1,B,0,C,D]
[A,B,1,B,0,C] [A,A,B,C,1,0] [A,B,B,1,0,C]
```

So, an order 4 graph is canonical if and only if it is not an instance of any of these six graph patterns. One can check that this is the case for the 11 canonical graphs detailed in Example 1.

For demonstration, it is easy to see why the first graph pattern of Example 2 is a valid pattern. If the vector representation of the graph G starts with 1,0, which correspond to the edges $\{1,2\},\{1,3\}$, these can be swapped by the transposition $[1,3,2,4]$ (swap vertices 2 and 3) resulting in a smaller graph (this will possibly affect other edges, but these are no longer important for the lexicographic comparison). So, G is not canonical.

Formally, graph patterns derive from permutations as stated in the following definition which is adapted from [6] where edge variables are ordered by rows (in this paper we order the edges by columns).

▶ **Definition 3** ([6]). *Let π be a permutation, G be an unknown graph of order-n with $edges(G) = \langle x_1,\ldots,x_m\rangle$, $edges(\pi(G)) = \langle y_1,\ldots,y_m\rangle$, and let $1 \le i \le m$. The graph pattern, $\mathsf{pat}_i(\pi)$, is the result of applying the most general unifier to the equations from the right side in Equation (1) for the case of $edges(\pi(G)) <^i_{lex} edges(G)$ (i.e., the most general substitution that makes both sides of the equations identical). If the equations have no solution, then we denote $\mathsf{pats}_i(\pi) = \bot$. The set of all graph patterns of order n is denoted: $AllPats(n) = \{\ \mathsf{pat}_i(\pi) \neq \bot \mid\ \pi \in S_n, 1 \le i \le \binom{n}{2}\ \}$.*

The graph pattern $\mathsf{pat}_i(\pi)$ represents the set of graphs that get smaller at position i with π. If π is a particular type of permutation, e.g., a transposition, then we say that $\mathsf{pat}_i(\pi)$ is a pattern of that type. Note that $\mathsf{pat}_i(\pi)$ is a legal graph pattern because graphs that get smaller under some permutation are trivially non-canonical. Observe that $\mathsf{pat}_3([1,3,2,4]) = \bot$, as in this case $edges(G) = [x_1,x_2,x_3,x_4,x_5,x_6]$, $edges(\pi(G)) = [x_2,x_1,x_3,x_4,x_6,x_5]$ and $edges(\pi(G)) <^3_{lex} edges(G)$ has no solution (because $x_3 \not< x_3$). Observe also, that in some cases, different permutations lead to identical patterns. For example, $\mathsf{pat}_3([2,5,1,3,4]) = \mathsf{pat}_3([2,5,1,4,3]) = [x_1,x_1,1,x_2,x_3,x_4,0,x_1,x_5,x_6]$.

▶ **Example 4.** The following details how the graph patterns from Example 2 are derived from permutations.

$\mathsf{pat}_1([1,3,2,4]) = [1,0,A,B,C,D]$ $\quad$ $\mathsf{pat}_2([2,1,3,4]) = [A,1,0,B,C,D]$
$\mathsf{pat}_2([1,2,4,3]) = [A,1,B,0,C,D]$ $\quad$ $\mathsf{pat}_3([1,2,4,3]) = [A,B,1,B,0,C]$
$\mathsf{pat}_4([2,1,3,4]) = [A,B,B,1,0,C]$ $\quad$ $\mathsf{pat}_5([1,3,2,4]) = [A,A,B,C,1,0]$

▶ **Definition 5** (cover). *Let $\pi \in S_n$, $1 \le i \le \binom{n}{2}$, and let $\mathsf{pat}_i(\pi) \neq \bot$. Then, $\mathsf{pat}_i(\pi)$ covers a graph G if G is an instance of $\mathsf{pat}_i(\pi)$. Equivalently, $\mathsf{pat}_i(\pi)$ covers G if $\pi(G) <^i G$. We denote the set $cover(\mathsf{pat}_i(\pi)) = \{\ G \mid\ \pi(G) <^i G\ \}$. Sometimes we write also $cover(\pi,i)$. If $\mathsf{pat}_i(\pi) = \bot$ then it has no instances and so in this case, we define $cover(\mathsf{pat}_i(\pi)) = \emptyset$. The number of graphs covered by a graph pattern p is 2^k where k is the number of variables in p.*

Some graph patterns "dominate" others.

▶ **Definition 6** (dominate). *We say that graph pattern p_1 dominates graph pattern p_2 if $cover(p_2) \subseteq cover(p_1)$. Given a set S of graph patterns, $dominators(S)$ denotes the set of dominating atoms in S. We denote $DomPats(n) = dominators(AllPats(n))$.*

Viewing graph patterns as first-order logic terms, p_1 dominates p_2 if p_1 is more general than p_2. In Prolog, one can implement a test for "p_1 dominates p_2" using the built-in operator `subsumes_term(p1,p2)`.

▶ **Example 7.** There are 59 elements in $AllPats(4)$ (after removing redundancies). The set $DomPats(4)$ contains only 18 elements. For instance,

$\mathsf{pat}_3([1,2,4,3]) = [A,B,1,B,0,C]$ dominates $\mathsf{pat}_3([1,4,2,3]) = [A,A,1,A,0,B]$, and
$\mathsf{pat}_2([2,1,4,3]) = [A,1,B,C,0,D]$ dominates $\mathsf{pat}_3([3,4,2,1]) = [A,1,1,B,0,A]$.

Table 1 Numbers of graph patterns (and dominating graph patterns) for various types of permutations: consecutive transpositions (`ct`), transpositions (`t`), consecutive involutions with transpositions (`ci+t`), disjoint involutions (`di`), involutions (`i`), and all permutations (`all`).

order	ct	t	ci+t	di	i	all
4	6 (6)	12 (12)	14 (14)	14 (14)	17 (16)	59 (18)
5	12 (12)	30 (30)	40 (39)	47 (46)	80 (75)	550 (163)
6	20 (20)	60 (60)	92 (88)	136 (130)	348 (327)	4610 (1648)
7	30 (30)	105 (105)	187 (176)	361 (339)	1451 (1369)	43065 (17945)
8	42 (42)	168 (168)	354 (329)	906 (842)	6055 (5762)	421435 (199509)

Table 1 details the total number of graph patterns and the number of dominating graph patterns (in parentheses) for various types of permutations with small values of n. For consecutive transpositions, and for transpositions, all of the patterns are dominating. For other types of involutions, a large majority of the patterns are dominating.

Some graph patterns are orthogonal to others.

▶ **Definition 8** (orthogonal). *Let p_1 and p_2 be (non-$\bot$) graphs patterns. We say that p_1 is orthogonal to p_2 if $cover(p_1) \cap cover(p_2) = \emptyset$.*

In Prolog, one can implement a test for "p_1 is orthogonal to p_2" using the built-in operator for "not unifiable".

CEGAR. In [14] and in [4] the authors compute complete symmetry breaks for graphs based respectively on permutations and on a refinement of permutations which they term "implications". A similar approach is applied in [10] to break the symmetries for finite models. Common to all of these works is an algorithm based on *counter-example guided abstraction refinement (CEGAR)* [3].

In a nutshell, and in its simplest form, the CEGAR-based algorithm performs as follows: Let Ψ denote a set of permutations, which is initially empty. The algorithm repeatedly seeks a counter-example to the statement: "Ψ *is a complete symmetry break*". A counter-example takes the form: graph G and permutation π such that

$$\pi(G) < G \wedge \bigwedge_{\pi' \in \Psi} G \leq \pi'(G).$$

If such a counter-example is found then $\Psi = \Psi \cup \{\pi\}$. If no such counter-example is found, then Ψ is a complete symmetry break. The search for a counter-example is implemented using a SAT encoding and incremental SAT solving.[1]

An important detail is that Ψ may contain redundant elements. For example, if a permutation added at some point becomes redundant in view of permutations added later. A second phase of the algorithm iterates on the elements of Ψ to remove redundant permutations (similar to the iterative algorithm for a minimally unsatisfiable set or monotone predicates in general [21, 20]). It is important to note that the time to perform the second phase is costly. For example, in [14], the authors report that the time to compute the complete symmetry break Ψ for order 10 graphs is close to 10 hours and the time to reduce it is 84 hours.

In this paper, we focus on symmetry breaks defined in terms of graph patterns. We seek a set of graph patterns that covers all of the non-canonical graphs. The set $AllPats(n)$ of all graph patterns clearly covers all non-canonical graphs and hence it is a complete symmetry

[1] The SAT solver cadical-2.1.0 [2] was used in all our experiments.

break. But this set is too large to be of practical use. We adapt the CEGAR approach to graph patterns. In this setting, Ψ is a set of graph patterns, and we repeatedly seek a counter-example: a graph G and a graph pattern $\mathsf{pat}_i(\pi)$ such that G is covered by $\mathsf{pat}_i(\pi)$ but is not covered by any of the graph patterns in Ψ. If such a counter-example is found then $\Psi = \Psi \cup \{\mathsf{pat}_i(\pi)\}$. If no such counter-example is found, then Ψ is a complete symmetry break.

Let $p = [p_1, \ldots, p_m]$ be a graph pattern and let $G = [x_1, \ldots, x_m]$ denote an unknown graph. Let I_1 denote the set of induces in p which contain a zero, I_2 denote the set of indices in p which contain a one, and I_3 denote the set of pairs of indices that contain equal variables. The single clause:

$$c = (\bigvee_{i \in I_1} x_i) \vee (\bigvee_{i \in I_2} \neg x_i) \vee (\bigvee_{(i,j) \in I_3} x_i \oplus x_j)$$

encodes that G is not covered by p. Strictly speaking, c is not a clause due to the `xor` operations. It is straightforward to replace $x_i \oplus x_j$ by a fresh variable x_{ij} and to add clauses for $x_{ij} \leftrightarrow x_i \oplus x_j$ (note that these fresh variables are reused across the encoding of different graph patterns).

▶ **Example 9.** let $p = [A, B, 1, B, 0, C]$. The clause $(x_5 \vee \neg x_3 \vee x_2 \oplus x_4)$ encodes that $G = [x_1, x_2, x_3, x_4, x_5, x_6]$ is not covered by p.

In [4], the authors present a CEGAR based approach to derive complete symmetry breaks based on "lex-implications". Essentially, lex-implications are constraints of the form detailed in Equation (1). Similar to graph patterns, lex-implications can be represented as pairs of the form (π, i). The novelty in the presentation in this paper stems from the graph pattern perspective where it is natural to talk about concepts such as cover, dominance and orthogonality. In the graph pattern perspective, any set S of graph patterns is a symmetry break in the sense that a graph G is canonical, *only if* it is not an instance of an element in S. Moreover, S is a complete symmetry break if also the inverse *(if)* statement holds.

3 Breaking Symmetries with Graph Patterns: A Greedy Approach

The CEGAR approach outlined above breaks symmetries as they are suggested by the SAT solver as counter-examples. This might lead to unnecessarily large sets of graph patterns as the SAT solver makes arbitrary choices relatively to the ultimate objective. In this section, we apply a greedy approach to derive symmetry breaks consisting of graph patterns. At the core of the approach is the notion of ranking a graph pattern in view of the set Ψ of graph patterns selected so far.

▶ **Definition 10.** *Let Ψ be a set of graph patterns (a symmetry break) and let p be a graph pattern. We define ranking$_\Psi(p)$ to be the number of graphs covered by p but not covered by any of the elements of Ψ*

The basic idea is presented as Algorithm 1. At Line 2, The set S of candidate graph patterns is initialized to the set of dominating graph patterns. The symmetry break Ψ is initialized to the empty set. At each step of the algorithm, a graph pattern p with maximal ranking is selected and the sets S and Ψ are updated.

In the implementation, ranking the current set S of candidate graph patterns (Line 5 in Algorithm 1) is a bottleneck. Each graph pattern is ranked using a sat encoding with a model counter. As an optimization, in the loop at Lines 4–6, we remove from S all graph

Algorithm 1 Greedily compute symmetry break for order n graphs.

```
1:  procedure SYMBREAK(n)
2:      S ← DomPats(n); Ψ ← ∅;
3:      repeat
4:          select p ∈ S such that
5:                r = max_{p'∈S} ranking_Ψ(p');
6:                ranking_Ψ(p) = r;
7:          if r>0 then
8:              S ← S \ {p}; Ψ ← Ψ ∪ {p};
9:      until r = 0;
10:     return Ψ;                          ▷ Ψ is a complete symmetry break
```

patterns p such that $\text{ranking}_\Psi(p) = 0$. Moreover, whenever we select a pattern $p \in S$ at Line 4, we also select a set $S' \subseteq S$ (as large as possible) so that $\{p\} \cup S'$ are mutually orthogonal (as prescribed by Definition 8). The selection of S' is greedy: When selecting $p \in S$ at Line 4, initialize $S' = \{p\}$ and iterate over the patterns from S, adding a pattern to S' if it is orthogonal to those already in S'.

▶ **Example 11.** The following details the run of Algorithm 1 for $n = 4$. The algorithm selects 6 graph patterns in 3 rounds. The column labeled Δ indicates the number of graphs covered by this graph pattern and not covered by those above. In this example, the second graph pattern added in each round is orthogonal to that from the first added in the round. The selected graph patterns are the same as those detailed in Example 2. Note that the sum of the numbers in the Δ column is 53, corresponding to the number of non-canonical graphs of order 4.

round	pattern	Δ
1	[1,0,A,B,C,D]	16
1	[A,1,0,B,C,D]	16
2	[A,1,B,0,C,D]	8
2	[A,B,1,B,0,C]	6
3	[A,A,B,C,1,0]	5
3	[A,B,B,1,0,C]	2

The following proposition identifies four specific graph patterns that cover 3/4 of the total number of graphs. Observe that the permutations detailed in the proposition are consecutive transpositions.

▶ **Proposition 12.** *For $n \geq 5$, the four top ranking graph patterns always take the following particular form where $m = \binom{n}{2} - 2$. Below, we adopt the cycle notation for permutations. The permutation (j, k) is the permutation which swaps elements j and k and leaves all other elements fixed.*

1. $\mathsf{pat}_1((2,3)) = [1, 0, x_1, x_2, x_3, x_4, x_5, x_6, \ldots, x_m]$
2. $\mathsf{pat}_2((1,2)) = [x_1, 1, 0, x_2, x_3, x_4, x_5, x_6, \ldots, x_m]$
3. $\mathsf{pat}_2((3,4)) = [x_1, 1, x_2, 0, x_3, x_4, x_5, x_6, \ldots, x_m]$
4. $\mathsf{pat}_4((4,5)) = [x_1, x_2, x_3, 1, x_4, x_5, 0, x_6, \ldots, x_m]$

The first two graph patterns are orthogonal and each covers 2^{m-2} graphs. The second two are also orthogonal and each covers 2^{m-3} graphs from those not covered by the first two. Together they cover 3/4 of the total number of graphs.

Table 2 Profiling partial and complete symmetry breaks obtained from greedy algorithm.

n	symmetry break	total	ct	t	ci+t	di	i	ρ	% ncc
7	greedy complete	116	28	30	59	60	80	1.00	100.00
	greedy partial	58	28	28	48	49	51	1.27	99.99
	greedy ct partial	13	13	0	0	0	0	24.38	98.78
	cegar complete	136	15	16	44	48	62	1.00	100.00
	cegar partial	68	15	16	25	27	30	1035.25	51.56
	involutions partial	113	28	30	60	70	113	1.01	99.99
8	greedy complete	439	40	46	120	124	235	1.00	100.00
	greedy partial	219	40	44	115	116	153	1.07	99.99
	greedy ct partial	18	18	0	0	0	0	52.63	99.76
	cegar complete	492	19	24	104	114	194	1.00	100.00
	cegar partial	246	19	24	63	71	95	9710.21	44.66
	involutions partial	396	40	47	127	158	396	1.02	99.99

Proof (sketch). The proof follows from a theorem presented in [18], which states that if G is an order n lex-leader canonical graph where the order of edges is column-wise, then so is the subgraph $G(k)$ on the first k vertices for $1 \leq k < n$. It follows that for any position $i < \binom{n}{2}$, the graph pattern $\mathsf{pat}_i((j,k))$ on order-n graphs is also a graph pattern for order-$n+1$ graphs. ◀

Table 2 compares 6 symmetry breaks and profiles them with respect to 5 types of permutations. The first three symmetry breaks are obtained using Algorithm 1: `greedy complete` is the complete symmetry break, `greedy partial` is the partial symmetry break consisting of the first (highest ranking) 50% of the patterns obtained from Algorithm 1, and `greedy ct partial` is the partial symmetry break consisting of the (longest) prefix of patterns obtained which are all consecutive transpositions. The next two symmetry breaks are obtained using the CEGAR algorithm described in [4]: `cegar complete` is the complete symmetry break obtained, and `cegar partial` is the partial symmetry break consisting of the first 50% of the patterns obtained from the CEGAR algorithm. We will come back to the fifth symmetry break, `involutions partial` later.

The column labeled `total` details the total number of patterns in the symmetry break. The next 5 columns detail the number of patterns in the symmetry break derived from a permutation of a given type: `(ct)` consecutive transpositions, `(t)` all transpositions, `(ci+t)` consecutive involutions and transpositions, `(di)` disjoint involutions, and `(i)` all involutions. All of these types of permutations are specific forms of involutions. The first two: `(ct)` and `(t)` are those widely applied in the symmetry breaks defined in [8, 7].

The penultimate column, labeled ρ, shows the redundancy ratio. This is the ratio between the number of graphs that are not covered by the selected patterns (symmetries not broken) and the number of isomorphism classes (all symmetries broken). If the set of patterns is a complete symmetry break, then $\rho = 1$. The smaller this number is, the better the symmetry break. The last column, labeled `% ncc`, details the percentage of non-canonical graphs covered by the symmetry break. If the set of patterns is a complete symmetry break, then this number is 100.

What we learn from Table 2:

The first $4-10\%$ of the patterns selected are all derived from consecutive transpositions. These patterns alone already cover $> 98\%$ of the non-canonical graphs. Moreover,

as stated in Proposition 12, the first four patterns always cover a total of 3/4 of the total number of graphs which is also about 75% of the total number of non-canonical graphs.

About half of the patterns selected in `greedy complete` are involutions; These are highly ranked. More than 60% are in the top half (by ranking) of the patterns. In contrast, for $n = 7$ less than 5% of all (n-factorial) permutations are involutions, and for $n = 8$ less than 2%. The number of involutions is given as sequence `A000085` of the Online Encyclopedia of Integer Sequences.

Greedy selection pays off. Comparing the redundancy ratios for `cegar partial` and for `greedy partial` shows this. Basically, the CEGAR-based algorithm selects an arbitrary pattern which covers some graph that is not yet covered. In contrast to the greedy algorithm where the best such pattern is chosen.

There are a small number of graph patterns that cover a large portion of the search space. All of these patterns derive from consecutive transpositions. This is apparent from the last column in the table and the rows corresponding to `greedy ct partial`: for n=7, 13 graph patterns cover $> 98\%$ of the non-canonical graphs, and for $n = 8$, 18 graph patterns cover $> 99\%$. We propose to consider three numbers when evaluating the quality of a symmetry break: (1) the redundancy ratio, (2) the percentage of non-canonical graphs covered, and (3) the size of the symmetry break. It is easy to obtain perfect values for the first two numbers when the third is large.

Given the apparent importance of various types of involutions, we consider a fifth symmetry break in Table 2: `involutions partial`. Here, we take the set of all dominating patterns derived from involutions. These are reduced to remove redundant patterns (a pattern in a set is redundant if all of the graphs that it covers are covered by other patterns in the set). For example, when $n = 7$, there are 1369 dominating patterns for permutations which are involutions (see Table 1) and these can be reduced to 113 patterns after removing redundancies. As indicated in Table 2, adopting involutions gives a close-to-perfect symmetry break for small values of n.

The greedy approach does not scale. Algorithm 1 relies on the expensive application of a model counter to rank the candidate patterns. Also, the computation of all dominating patterns is too time-consuming. This motivates the approach taken in the next section.

4 Tweaking CEGAR for Better Partial and Complete Symmetry Breaks

In this section we take the lessons learned from Table 2 and apply them to guide a CEGAR-based algorithm to make better selections. We layer the selection of counter-examples with layers corresponding to the five types of patterns considered in the profiling of Table 2. This means that in each layer, each iteration of the CEGAR loop searches for a graph G and a permutation π s.t. $\pi(G) < G$ and π is the permutation used in that layer. This yields a graph pattern generated from π and covering graph G.[2]

In the first layer `(ct)`, we seek counter-examples that are consecutive transpositions. When no further counter-examples of this type remain, we proceed to layer `(t)` to collect counter-examples which are transpositions, and so on for the layers `(ci+t)` introducing consecutive

[2] Graph patterns generated by a single permutation are disjoint, hence only one can cover the graph G.

Table 3 Comparing partial symmetry breaks obtained by layered CEGAR.

ord	ct		t		ci+t		di		i		compl
	size	ratio	size	ratio	size	ratio	size	ratio	size	ratio	size
4	6	1.00	6	1.00	6	1.00	6	1.00	6	1.00	6
5	12	1.35	13	1.26	13	1.06	13	1.06	14	1.00	14
6	20	2.08	24	1.77	29	1.16	30	1.12	36	1.00	36
7	30	3.87	40	3.02	64	1.46	70	1.31	111	1.01	115
8	42	7.27	62	5.39	137	1.95	167	1.53	397	1.02	444
9	56	13.05	91	9.42	269	2.76	401	1.83	2,024	1.03	2,760
10	72	21.53	128	15.34	526	3.97	1,001	2.20	12,644	1.04	24,993
11	90	33.23	174	23.52	1,008	5.71	2,523	2.65	81,522	≈1.14	289,863
12	110	48.97	230	34.53	1,896	≈7.39	6,275	≈3.12	n/a	n/a	n/a

involutions, `(di)` for disjoint involutions, and `(i)` for all other involutions. In a final, sixth layer, we seek arbitrary counter-examples. To ensure that counter-examples are found of the required type, we encode in the SAT solver an additional condition in a straightforward fashion. For example, involutions are encoded as the implications $\pi(i) = j \Rightarrow \pi(j) = i$, for $i \neq j$.

The layered CEGAR-based algorithm can be applied in two different capacities: First, to provide partial symmetry breaks as obtained at the end of each layer of the run. Second, to guide the search for a complete symmetry break. Experimentation indicates that making a better selection of counter-examples early on reduces the number of iterations applied in the course of the algorithm and reduces the number of redundant patterns that need to be removed in the second phase of the algorithm.

Table 3 addresses the first capacity. Each pair of columns details the partial symmetry break obtained after the corresponding layer in the revised CEGAR algorithm. For each layer, we present the size (number of patterns) of the corresponding partial symmetry break and the corresponding redundancy ratio. For the last layer (in the last column), the symmetry break is complete and the redundancy ratio is always 1.00. The first two pairs of columns `ct` and `t` correspond to the symmetry breaks described in [8]. To compute the redundancy ratios, the knowledge-compilation-based tool `d4` [19] was applied for model counting.[3] In cases where the model counter was not able to compute the number of graphs we applied the approximate model counter `approx` [25]. These cases are marked by the symbol ≈.

Now we look at the effect of layering when CEGAR is used to calculate a complete symmetry break. Figure 1 summarizes the comparison of the layered approach to CEGAR and the standard one. Figure 1 (on the left), details numbers of iterations (and patterns). The two upper curves detail the number of iterations required to calculate a complete cover of all graphs of order $n \in \{8..11\}$. Each iteration of the CEGAR loop adds one pattern to the cover. However, some patterns might become redundant due to the addition of a stronger pattern later on. The lower two curves detail the number of patterns in the irredundant cover obtained by reducing the output of the CEGAR algorithm (irredundant cover is calculated by standard means [20]). Note that the size of the irredundant cover must always be smaller or equal to the number of iterations of the CEGAR iterations.

[3] Other model counters were considered but `d4` gave better performance.

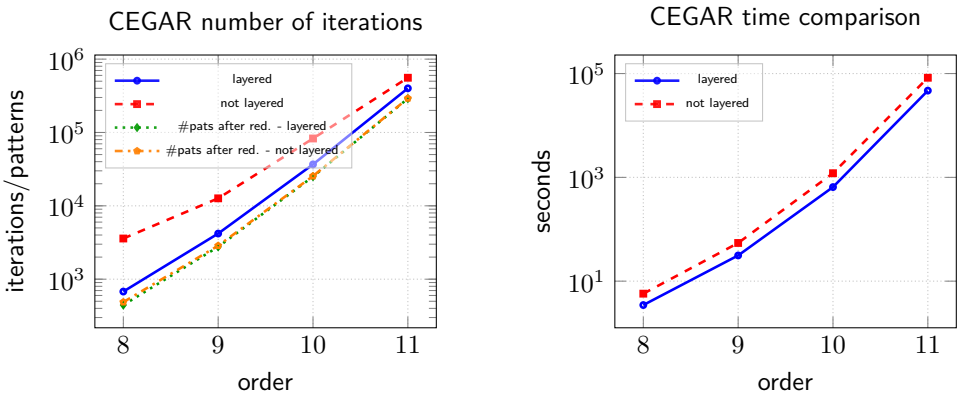


Figure 1 Comparison of CEGAR and Layered CEGAR (log scale).

The number of iterations is always lower for the layered approach than for the standard one. This typically also reduces the overall time. For $n = 11$, CEGAR needed 400,016 iterations in the layered approach, contrasting with 557,279 iterations in the non-layered (standard) approach. Interestingly, both were reduced to covers of similar sizes, 289,863 in the layered approach and 291,888 in the non-layered one. This indicates that the layered approach guides CEGAR *more precisely*.

Figure 1 (on the right) complements this information with the total CPU time needed to calculate the cover, and it can be observed that lowering the iterations of the loop also reduces the total time.

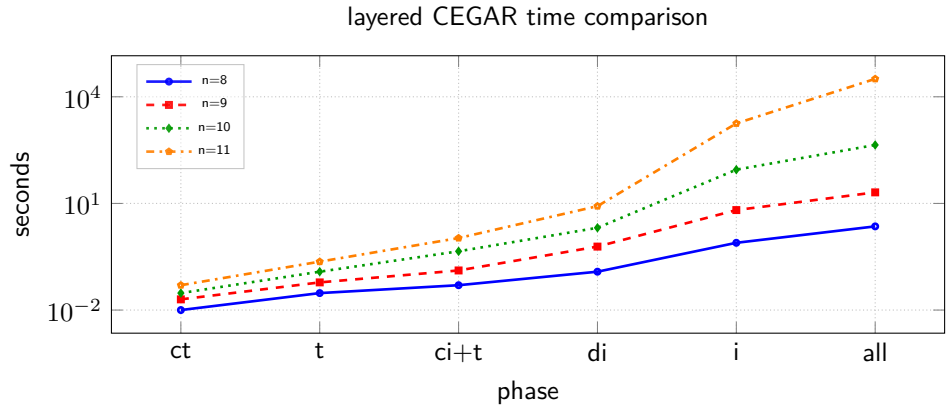


Figure 2 Computation time for partial symmetry breaks with layered CEGAR (log scale).

Figure 2 details the computation times for partial symmetry breaks obtained in the different phases of the layered CEGAR algorithm. For $n = 11$, computing the partial symmetry breaks defined in terms of disjoint involutions and in terms of all involutions is respectively 3 and 1 orders of magnitude faster than computing the complete symmetry break. Given the fact that these symmetry breaks are relatively small and have good redundancy ratios indicates that they provide a good choice when balancing time (to compute them) and size with precision.

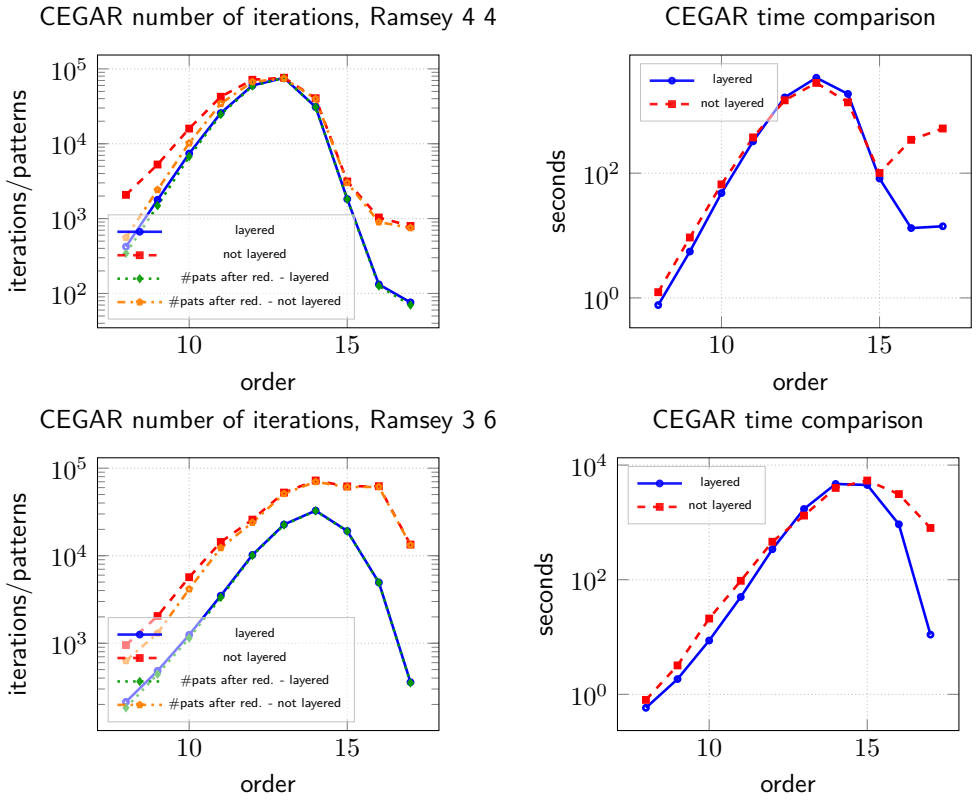


Figure 3 Comparison of CEGAR's Performance on Ramsey graphs (log scale).

5 Symmetry Breaks for a Specific Graph Search Problem

In this section, we describe the computation of partial and complete symmetry breaks tailored for a specific graph search problem. In this context, when seeking a counter-example in the CEGAR loop, we restrict the search to graphs that satisfy the constraints of the given search problem. We apply this approach for the layered approach to CEGAR and the standard one.

We consider a classic example: the search for *Ramsey graphs* [22]. The graph $R(s,t;n)$ is a simple graph with n vertices that contains no clique of size s, and no independent set of size t. The *Ramsey number* $R(s,t)$ is the smallest number n for which there is no $R(s,t;n)$ graph. We also describe the impact of using these symmetry breaks.

Figure 3 summarizes the comparison of the computation of symmetry breaks in this context. We focus on symmetry breaks for $R(4,4;n)$ (two upper plots) and $R(3,6;n)$ (two lower plots).[4] We study only the values of n until the respective Ramsey numbers, which are both 18. The number of Ramsey graphs peaks at some point, at which it is also difficult to calculate the cover. The two plots on the left detail the numbers of iterations in the layered/non-layered approaches as well as the numbers of patterns removed after the CEGAR loops (upper for $Ramsey(4,4;n)$ and lower for $Ramsey(3,6;n)$). The two plots on the right detail total CPU times (upper for $Ramsey(4,4;n)$ and lower for $Ramsey(3,6;n)$).

[4] These enable us to perform a precise analysis, however, unsatisfiability alone for larger Ramsey numbers is famously difficult [12].

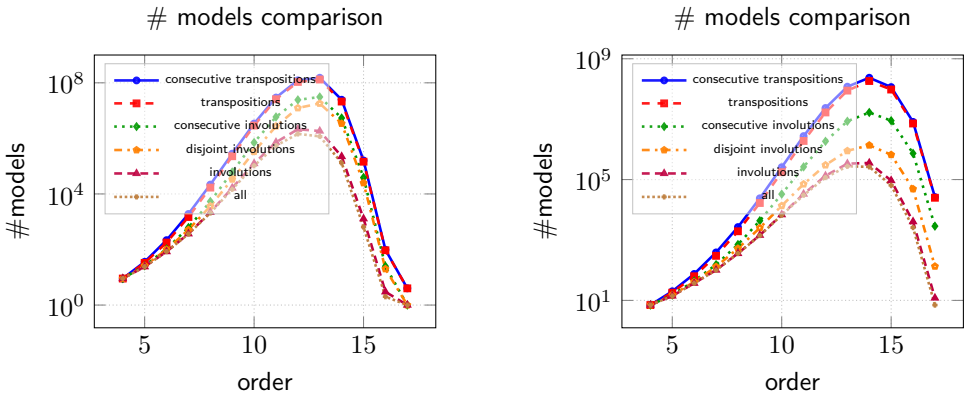


Figure 4 The impact of various partial symmetry breaks on Ramsey graphs (log scale).

The number of iterations is always lower for the layered approach than for the standard one. Also, the number of patterns removed in the reduce phase is lower. This typically reduces the overall time. This is, however, not always the case as individual SAT calls are more expensive.

Here, in comparison to Figure 1, the number of patterns removed after the CEGAR loop is much smaller. Interestingly, in the case of $R(3,6;n)$, not only that the number of iterations is significantly smaller in the layered approach, but it also remains so after reduction. This suggests that the two runs of CEGAR+reduction reach completely different local optima.

Figure 4 depicts the number of Ramsey graphs found when applying the various proposed symmetry breaks: $Ramsey(4,4;n)$ on the left and $Ramsey(3,6;n)$ on the right. The curves in both plots from highest to lowest, correspond precisely and in order to the 6 layers: `(ct)`, `(t)`, `(ci+t)`, `(di)`, `(i)`, and `(all)` (the complete symmetry break). The two upper curves describe the application of the widely applied partial symmetry breaks defined in [8]. The lowest curve describes the application of a complete symmetry break. General transpositions (`t`) do not give much advantage over just consecutive transpositions `(ct)`. However, both are by orders of magnitude worse than the other classes of permutations. Note the proximity of the curve for all involutions (`i`) to the lowest curve (the complete symmetry break). This suggests that focusing only on involutions gives a symmetry break very close to the complete break.

6 Conclusions and Future Work

The objective of this paper is to approach the search for graph symmetry breaks in a systematic way. We study the structure of graph patterns that are selected in a greedy algorithm to break all symmetries for graphs of small orders. We learn that graph patterns that derive from a specific class of permutations, called involutions, play an important role. Involutions generalize the class of transpositions, which play an important role in breaking symmetries for graphs. We then refine a CEGAR-based approach to compute symmetry breaks introducing a layered approach. In this way, we can guide the CEGAR-based search for a complete symmetry break and also provide a series of partial symmetry breaks of increasing precision.

It is important to reflect on why we construct partial symmetry breaks in a CEGAR-based algorithm, in contrast to simply collecting all permutations of the restricted types. For patterns based on transpositions, the straightforward construction is doable. However,

for patterns based on the other types of involutions, there are too many of them and the CEGAR-based approach enables us to select those that contribute to the corresponding symmetry breaks.

We do not expect to find a complete symmetry break of polynomial size that identifies canonical graphs that are lex-leaders, cf. [16]. However, we still aim to find small, perhaps even polynomial in size, *partial* symmetry breaks that break a majority of the symmetries on graphs. Coming back to the greedy algorithm presented in Section 3, for $n = 8$, 439 graph patterns are found to cover all of the 268,423,110 non-canonical graphs of order 8. The last 163 graph patterns selected in the greedy algorithm, cover a negligible total of 382 of these 268,423,110 graphs. In much the same way that Proposition 12 points to four specific graph patterns that cover 75% of the non-canonical graphs, the *holy grail* for symmetry breaking for graphs is to specify a small symmetry break that breaks a significant portion of the symmetries.

This paper opens a number of avenues for future work. Since focusing on specific permutation types positively impacts CEGAR, we plan to also apply the lessons learned in this paper in the context of a dynamic symmetry breaking for graph generation. In this approach, for example as performed in [17], symmetries are detected and broken during the generation (and enumeration) of the solutions of a given graph search problem. Another direction of research is to identify further sub-classes of permutations by either refining the framework proposed here or looking for completely different classes. Such division could also be problem-specific. A more theoretical direction of research is to find a justification for why involutions lead to better symmetry breaks.

References

1 László Babai and Eugene M. Luks. Canonical labeling of graphs. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 171–183. ACM, 1983. doi:10.1145/800061.808746.

2 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.

3 Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. doi:10.1007/10722167_15.

4 Michael Codish, Thorsten Ehlers, Graeme Gange, Avraham Itzhakov, and Peter J. Stuckey. Breaking symmetries with lex implications. In John P. Gallagher and Martin Sulzmann, editors, *Functional and Logic Programming - 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9-11, 2018, Proceedings*, volume 10818 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 2018. doi:10.1007/978-3-319-90686-7_12.

5 Michael Codish, Graeme Gange, Avraham Itzhakov, and Peter J. Stuckey. Breaking symmetries in graphs: The nauty way. In Michel Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2016. doi:10.1007/978-3-319-44953-1_11.

6 Michael Codish and Mikoláš Janota. Breaking symmetries from a set-covering perspective. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 22nd International Conference, CPAIOR, Melbourne, Australia November 10-13, 2025 Proceedings*, Lecture Notes in Computer Science. Springer, 2025.

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

8 Michael Codish, Alice Miller, Patrick Prosser, and Peter James Stuckey. Breaking symmetries in graph representation. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 510–516, 2013. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6480.

9 James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, pages 148–159. Morgan Kaufmann, 1996.

10 Marek Dančo, Mikoláš Janota, Michael Codish, and João Jorge Araújo. Complete symmetry breaking for finite models. In *AAAI 2025*, 2025. doi:10.48550/arXiv.2502.10155.

11 Alan M. Frisch and Warwick Harvey. Constraints for breaking all row and column symmetries in a three-by-two matrix. In *Proceedings of SymCon'03*, 2003.

12 Thibault Gauthier and Chad E. Brown. A formal proof of R(4, 5)=25. 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 16:1–16:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.ITP.2024.16.

13 Marijn J. H. Heule. Optimal symmetry breaking for graph problems. *Mathematics in Computer Science*, 2019.

14 Avraham Itzhakov and Michael Codish. Breaking symmetries in graph search with canonizing sets. *Constraints*, pages 1–18, 2016.

15 Avraham Itzhakov and Michael Codish. Breaking symmetries with high dimensional graph invariants and their combination. In André A. Ciré, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 20th International Conference, CPAIOR 2023, Nice, France, May 29 - June 1, 2023, Proceedings*, volume 13884 of *Lecture Notes in Computer Science*, pages 133–149. Springer, 2023. doi:10.1007/978-3-031-33271-5_10.

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

17 Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation and enumeration. *ACM Trans. Comput. Log.*, 25(3):1–30, 2024. doi:10.1145/3670405.

18 Vladmír Kvasnička and Jiří Pospíchal. Canonical indexing and constructive enumeration of molecular graphs. *J. Chem. Inf. Comput. Sci.*, 30(2):99–105, April 1990. doi:10.1021/CI00066A001.

19 Jean-Marie Lagniez and Pierre Marquis. An improved decision-DNNF compiler. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 667–673, 2017. doi:10.24963/ijcai.2017/93.

20 João Marques-Silva, Mikoláš Janota, and Anton Belov. Minimal sets over monotone predicates in boolean formulae. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 592–607. Springer, 2013. doi:10.1007/978-3-642-39799-8_39.

21 Alexander Nöhrer, Armin Biere, and Alexander Egyed. Managing SAT inconsistencies with HUMUS. In Ulrich W. Eisenecker, Sven Apel, and Stefania Gnesi, editors, *Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings*, pages 83–91. ACM, 2012. doi:10.1145/2110147.2110157.

22 Stanislaw P. Radziszowski. Small Ramsey numbers. *Electronic Journal of Combinatorics*, 1994. Revision #14: January, 2014. URL: http://www.combinatorics.org/.

23 Jussi Rintanen and Masood Feyzbakhsh Rankooh. Symmetry-breaking constraints for directed graphs. In Ulle Endriss, Francisco S. Melo, Kerstin Bach, Alberto José Bugarín Diz, Jose Maria Alonso-Moral, Senén Barro, and Fredrik Heintz, editors, *ECAI 2024 - 27th European Conference on Artificial Intelligence, 19-24 October 2024, Santiago de Compostela, Spain - Including 13th Conference on Prestigious Applications of Intelligent Systems (PAIS 2024)*, volume 392 of *Frontiers in Artificial Intelligence and Applications*, pages 4248–4253. IOS Press, 2024. doi:10.3233/FAIA240998.

24 Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics*, 155(12):1539–1548, 2007. doi:10.1016/J.DAM.2005.10.018.

25 Mate Soos and Kuldeep S. Meel. BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*, pages 1592–1599. AAAI Press, 2019. doi:10.1609/AAAI.V33I01.33011592.

26 Toby Walsh. General symmetry breaking constraints. In *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, pages 650–664, 2006. doi:10.1007/11889205_46.

Constraint Models for Klondike

Nguyen Dang
School of Computer Science, University of St Andrews, UK

Ian P. Gent
School of Computer Science, University of St Andrews, UK

Peter Nightingale
Department of Computer Science, University of York, UK

Felix Ulrich-Oltean
Department of Computer Science, University of York, UK

Jack Waller
Skyscanner, London, UK

Abstract

Klondike is the most famous single-player card game, and remains a challenging search problem even in the "thoughtful" variant where all card locations are known. We consider the full game of Klondike except for one restriction that the unusual move of "worrying back" is disallowed. This model is able to determine the winnability of all instances of the game and in practice does so in less than 2000 secs for 10,000 instances we tested, which no other known algorithm can achieve. On some instances, however, other techniques can produce answers more quickly. We use constraint modelling to produce schedules for running our constraint model in combination with other techniques. The combination outperforms any single solver across a range of time limits. Using this combination we are able to significantly improve the best estimate of winnability of Klondike without worrying back. Finally we show how we can use this work to also improve the estimate of winnability of the regular game of Klondike.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Planning and scheduling

Keywords and phrases AI Planning, Modelling, Constraint Programming, Solitaire and Patience Games

Digital Object Identifier 10.4230/LIPIcs.CP.2025.9

Supplementary Material *Software*: `https://github.com/turingfan/CP2025-Klondike` archived at `swh:1:dir:762160702e7aecea6867770880c1cad78fdca5c8`

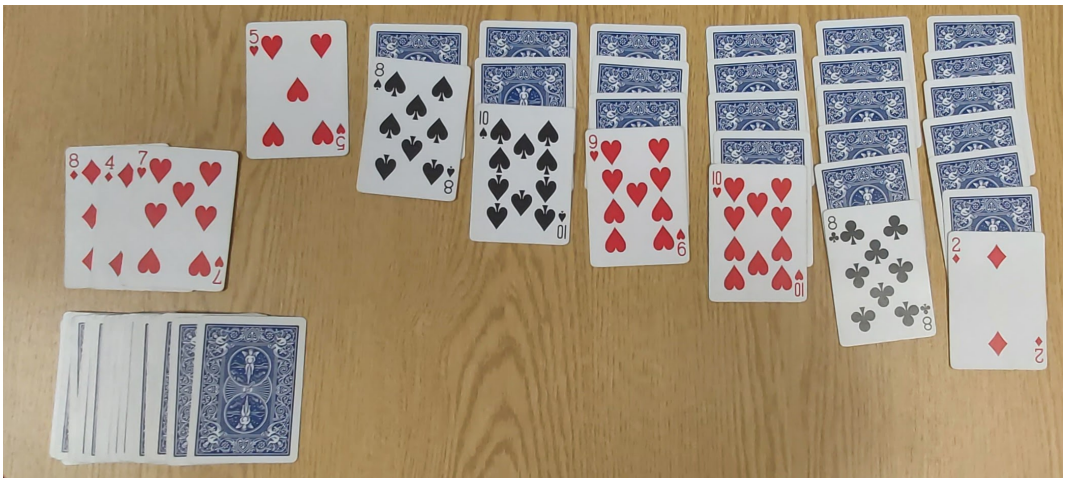
Funding *Peter Nightingale*: EPSRC grant EP/W001977/1
Felix Ulrich-Oltean: EPSRC grant EP/W001977/1

Acknowledgements Experiments were carried out on the Viking and Cirrus clusters. The Viking cluster is a high performance computing facility provided by the University of York, and we are grateful for computational support from the University of York IT Services and the Research IT team. Cirrus is a UK National Tier-2 HPC Service at EPCC (`http://www.cirrus.ac.uk`) funded by the University of Edinburgh and EPSRC (EP/P020267/1). Thanks to reviewers for their careful comments.

1 Introduction

"Klondike" is the formal name for the most popular solitaire or patience game, getting 100 million plays per day in 2020 just in Windows Solitaire [12]. We show that constraint models can perform extremely well on Klondike, despite the inherent complexity of the game due to its arbitrary rules. As with most previous research, we focus on the thoughtful variant, where the positions of all cards are known at the start of the game.

© Nguyen Dang, Ian P. Gent, Peter Nightingale, Felix Ulrich-Oltean, and Jack Waller;
licensed under Creative Commons License CC-BY 4.0



```
letting numRanks = 13
letting dealN = 3
letting numPiles = 7
letting firstFaceUpCards =[1, 2, 3, 4, 5, 6, 7]
letting tableau =[ [17], [8, 7], [35, 36, 9], [3, 48, 51, 21],
                   [50, 47, 29, 49, 22], [23, 11, 25, 32, 34, 33], [39, 14, 0, 26, 41, 20, 40]]
letting stock=[46, 42, 19, 43, 38, 12, 30, 2, 27, 44, 45, 28, 1, 10, 16, 4, 18, 31, 37, 6, 24, 13, 15, 5]
```

Figure 1 A sample layout of Klondike with a corresponding Essence Prime [18] parameter file. The file specifies 13 cards in each suit, cards dealt from stock in sets of 3, and with 7 piles of increasing size. The cards in the 7 tableau piles are given followed by the stock cards in order. Cards are numbers from 0 to `numRanks` $\times 4 - 1$. A card c's rank is given by c mod `numRanks` $+ 1$ and its suit by the integer c/`numRanks`. Suits 0, 1, 2, 3 are respectively Spades, Hearts, Clubs, Diamonds.

Figure 1 gives a sample layout of Klondike. In this example, the sole card in the first tableau pile is 5♡, while in the seventh pile the 2♢ is on top of 6 hidden cards. The first three cards in the stock of 8♢, 4♢, 7♡ have been turned face up ready for the 7♡ to be played. We provide a glossary of terms in Section A, since some may not be familiar. For example, in this paper we focus on "worrying back", which allows a card to be moved back from foundation to tableau. We quote the standard rules of Klondike from [4]: "A single standard deck is used and the goal is to build all cards on foundations in suit from A to K. The game begins with a tableau of 28 cards in a triangular form with piles from 1 to 7 cards, with all but the top card face-down. Face-up cards on the tableau may be built in alternating colour, and built groups may be moved. Face-down cards may not be moved. Spaces may be filled only by a K. Cards may be worried back from foundations to tableau. A stock of 24 cards may be drawn in groups of 3, and redeals are allowed without limit." Note that this is "deal-3" Klondike.

AI methods had not been generally effective at solving patience games until the introduction of Solvitaire [4], which performs a very fast depth-fast search with optimisations such as use of dominances and transposition tables. On some games, Solvitaire can perform poorly due to its lack of constraint propagation. Constraint solving has previously been used for different single player card games [10, 23], but not successfully for the full game of Klondike.

First, we give a constraint model for Klondike with a restriction that worrying-back is not allowed. This is a minor restriction in two senses. First because it only changes the result of about 0.4% of games [4]. Second, many players don't even realise that worrying back is possible, so in practice actually play this variant. Our model is able to solve all instances

– out of tens of thousands – that we have tested it on . Note that any game won without worrying back is definitely winnable with worrying back, so any solved instance is winnable in full Klondike.

Second, we describe models for the full game of Klondike with worrying back. Unlike the first model, they are not a complete model of the game. They are guaranteed correct when they report a layout unwinnable, but solve a relaxation of the game so can be wrong if they report a solution. They can identify 71% of all impossible layouts. Taken together, our constraint models slightly improve the best known estimate of winnability of full Klondike.

Finally, we show that we can use constraint modelling to produce schedules for running different solvers on Klondike which outperform any single solver across a range of time limits. We are able to use this schedule to give a significantly improved estimate of the winnability of Klondike without worrying-back.

2 Properties of Klondike Without Worrying-back

Most moves in Klondike involve moving only one card at a time. For example, when we move a card to foundation or a card from the stock to a tableau pile, this involves only that single card moving. There is an exception where multiple cards move at once. This is when a "built pile" is moved. This occurs if we have a tableau pile with consecutive descending cards of alternating colour. All these cards can be moved from one pile to another, if the highest ranked card being moved goes to a card of one higher rank and opposite colour. However, we note that we can still regard this as being the move of a single **critical card**, the highest ranked card in the pile. The other cards in the pile simply follow along with it. Given this, our models will concern only the move of the critical card: the critical card is either the single card being moved or the highest ranked card if a built pile is moved. This decision greatly simplifies the modelling process.

We make essential use of a dominance concerning the movement of built piles, proposed by Wolter [26]. This dominance restricts movements of a built pile where it is *not* the complete pile that moves. Moves of partial piles are allowed and are sometimes required to solve the problem. However, we can insist that if such a move is made, then the card above the critical card just moved must be immediately built to foundation. For example, if a built pile contains 9♡8♣7♢, we are allowed to move the partial pile 8♣7♢ to the 9♢, but only if we then immediately play the 9♡ to foundation. This restriction does not change the set of winnable instances [4], but does have an important consequence because we only consider games with exactly 4 suits in this paper. Since this version of Klondike does not allow worrying-back from foundation to tableau, it places a strict limit on the number of times any given card can be the critical card of a move. No card can be the critical card of a move more than three times in any solution. For example, in the case of 8♣, these three moves would be: building the 8♣ on either 9♡ (or 9♢); moving the 8♣ to the 9♢ (or 9♡); and moving the 8♣ to foundation on top of 7♣. Notice that there can be no third move of the 8♣ as critical card within the tableau: after the second move the the 9♡ (or 9♢) must be immediately moved to foundation and cannot be worried-back.

We can go slightly further and note that no pair of "twin" cards of same rank and colour (e.g. 8♣, 8♠) can make more than 5 moves as critical card between them. If they made 6 moves, they would both have to make a second tableau move. But whichever did so first would necessitate the building of the card it was built on to foundation, which would no longer be available for the second card to make its second tableau move to. This discussion establishes the following proposition, original to this paper.

▶ **Proposition 1.** *If a layout of Klondike without worrying back is winnable, it is also winnable with each card making at most three moves as the critical card. Furthermore, it is also winnable with each pair of cards of same colour and rank making at most five moves between them as the critical card.*

Note that a corollary of Proposition 1 is that the generalised game of thoughtful Klondike, with n cards of 4 suits, is in NP. For any winnable layout there would be a solution with at most $10n$ moves. However it does not follow that the game is NP Complete.

3 Constraint Model for Klondike Without Worrying-Back

We outline our models in this section but have also provided them in full in our supplementary material. The viewpoint of our model is based on the different possible critical moves that exist - at most $3 \times 52 = 156$ in the standard game. For each such move a variable determines when the move happens: we call this the *stage* the move is made at. We do allow two moves to happen at the same stage as each other. Doing this reduces the required domain size of the move variables. This reduces the search space and also the size of the models for the search phase, which is important since we are encoding to SAT for the main solving step.

Not all moves are immediately available at the start of a game. We encode this by stating that any move must happen no later than any move that depends on it happens. Expressing this is the core of our model. We summarise these rules in English as follows, referring back to these when giving a more detailed presentation of the model in Section 3.2. We start with rules which insist that one move must happen *strictly* later than a move it depends on.

1. Any move of a card to the tableau must be strictly after the card it is built on is first face-up in the tableau.
2. If two cards are built onto the same tableau card, the first one must move away strictly before the second one can be built.
3. Any move of a card originally face-down in the tableau must be strictly after the card covering it was first moved.
4. The stage a King is moved to a space must be strictly after the first move of the top card originally in the relevant pile.
5. If two Kings move to the same space, one must be built to foundation strictly before the other moves there.
6. The first stage that a card in the stock is moved must be strictly after the stage that the previous card was played from stock.

We also state rules restricting moves that must happen in monotonic stage order, but not necessarily strictly increasing. Not imposing a strict ordering in these cases allows us to further limit the required number of stages in Proposition 3.

7. Any move to foundation must be after (or the same stage) as the card of one less rank of the same suit.
8. Any move to foundation of a tableau card which has a card built on it must be after (or the same stage) that the covering card has moved away as either its second tableau move, or to foundation.

It is critical that the rules above disallow a cycle of dependencies at the same stage. If this was allowed then we might be able to assign stages to moves in the constraint model, but not use that solution to obtain a legal winning sequence of moves in the original game. We now prove two propositions related to these rules, both being original to this paper.

▶ **Proposition 2.** *There can be no cycle of dependencies at the same stage in the above rules, i.e. no two moves with the same stage number can each be required to happen no later than the other.*

Proof. There is nothing to prove for the rules which insist that the second move happens strictly after the first. The two rules which allow moves at the same stage share a key property: the card required to move later than the second card must be of higher rank than the second card. Thus no cycle is possible. ◀

▶ **Proposition 3.** *If a layout of Klondike without worrying back is winnable, the moves in it can be assigned stage numbers from 1 up to 1.5 times the number of cards in a deck, while respecting the rules above.*

Proof. Consider each pair of "twin" cards of same rank and colour, e.g. 8♣, 8♠. We have already established that these need make at most 5 moves as critical cards. But now note that any card which is moved from being built in the tableau to foundation does *not* require a unique stage number. It can happen at the same stage as whichever is later of: the last card built onto it in tableau is moved away; or the card of same suit and one rank less is built to foundation. Both of these cases fall into the non-strict monotonic requirements. Now we do a simple case analysis. Either both twin cards are built to tableau or at least one is not. In the first case we know there are at most 3 tableau moves between them and the moves to foundation do not require unique stages. In the second case, the card that does not move to tableau requires at most one unique stage when it moves to foundation: if the other card moves to tableau it requires at most two unique stages. In all cases therefore the two twin cards require at most 3 stages which are different to all other cards. Therefore in our model we set the number of stages to be at most 1.5 times the number of cards.[1] ◀

3.1 Decision Variables

The model contains a variety of decision variables, which we present in three thematic groups. The first group concerns movements of cards to foundation and tableau. As well as the natural domain values, e.g. cards or stages, some variables can take a dummy value ∅ which is used to indicate that a given variable does not take a meaningful value.

$M(c)$: The first stage at which card c is moved.
$F(c)$: The stage at which c is played to foundation.
$U(c)$: The first stage at which c is face up in the tableau and so can potentially be built on, or ∅ if it never is. A stock card c that is never played to tableau will have $U(c) = \varnothing$.
$T_1(c)$: The first stage at which c is built on another card in the tableau (or ∅ if never).
$P_1(c)$: If $T_1(c)$ is not ∅, the card that c was built onto in the tableau at that stage.
$T_2(c)$: The second stage at which c is built onto another card in the tableau (or ∅ if never).
$P_2(c)$: If $T_2(c)$ is not ∅, the card that c was built onto in the tableau at that stage.

The second group of decision variables deals with Kings moving to spaces in the tableau.

$PS(p)$: For each pile p, the stage at which p is first a space a King can be moved into.
$KS(k)$: For each king k, the stage at which the King is played to a space, or ∅ if never.
$KP(k)$: The tableau pile the King k is played to if it moves to a space, or ∅ if none.

[1] Minor optimisations beyond this could be made, since e.g. Aces need never make any tableau moves, but we did not add this to the model.

The final group of decision variables relates to movement of the stock and are vectors of variables with one variable for each index in the stock. The stock is provided in the parameter file by a vector `stock` which gives its cards in sequence at the start of the game.
***SI*(*i*)**: The i^{th} card played from stock is `stock`$(SI(i))$.
***SO*(*i*)**: The card `stock`(i) is the $SO(i)^{th}$ card played from the stock.

3.2 Constraints

Once the parameter file is read in, we set up some auxiliary variables to help in the expression of other constraints. Specifically, for any card c we set up the tableau parents (i.e. which two cards it can possibly be built onto) and the "twin" card `tw`(c) of the same rank and colour (but different suit). The tableau and twin matrices vary depending on the number of ranks, and they are both precomputed in `letting` statements using comprehensions.[2] For each card c, `tableau_set`(c) is true if c is in the tableau in the original layout, and `stock_set`(c) is true if c starts in the stock. For quantifications, we write $\mathbb{C}$ for the set of all cards, with $\mathbb{A}$ and $\mathbb{K}$ for the sets of Aces and Kings respectively, and $\mathbb{P}$ for the set of piles.

We start with some consistency constraints between the variables representing moves. A card's first move is no later than its build to foundation (1), and if a card is never face up in the tableau then its first move is to foundation (2). If a card is ever built on another card in the tableau, the first such move is its first move (3). Kings can never be built onto another card in the tableau - we deal with their moves to spaces below (4). A non-King cannot make a second move to tableau if it does not make a first, and if it makes neither then its first move is to foundation (5). If a non-King does not start on the tableau then it is first face-up in the tableau when first moved there (6). A card's second tableau move must be strictly after its first and no later than its foundation move, but can be at the same stage (7). Cards which are not built on another card in the tableau have no tableau parent (8,9). When not both dummy, both tableau parents are different since a card's second tableau move is to a different card than the first (10).

$$\forall c \in \mathbb{C} : M(c) \leq F(c) \tag{1}$$
$$\forall c \in \mathbb{C} : U(c) = \varnothing \rightarrow M(c) = F(c) \tag{2}$$
$$\forall c \in \mathbb{C} : T_1(c) \neq \varnothing \rightarrow M(c) = T_1(c) \tag{3}$$
$$\forall c \in \mathbb{K} : T_1(c) = \varnothing \wedge T_2(c) = \varnothing \tag{4}$$
$$\forall c \in \mathbb{C} \setminus \mathbb{K} : T_1(c) = \varnothing \rightarrow [\ T_2(c) = \varnothing\ \wedge\ M(c) = F(c)\ ] \tag{5}$$
$$\forall c \in \mathbb{C} \setminus \mathbb{K} : \neg\texttt{tableau_set}(c) \rightarrow T_1(c) = U(c) \tag{6}$$
$$\forall c \in \mathbb{C} : T_2(c) \neq \varnothing \rightarrow [\ T_1(c) < T_2(c)\ \wedge\ T_2(c) \leq F(c)\ ] \tag{7}$$
$$\forall c \in \mathbb{C} : T_1(c) = \varnothing \leftrightarrow P_1(c) = \varnothing \tag{8}$$
$$\forall c \in \mathbb{C} : T_2(c) = \varnothing \leftrightarrow P_2(c) = \varnothing \tag{9}$$
$$\forall c \in \mathbb{C} : P_1(c) \neq \varnothing \rightarrow P_1(c) \neq P_2(c) \tag{10}$$

For the rules of the game, we start with tableau and foundation moves. From Rule 7, a card must be played to the foundation no earlier than the card one lower in rank of the same suit, but we do allow both moves to happen at the same stage (11). From Rule 6, a card's tableau-parent must be face up on the tableau (strictly) before the card can be built onto it

[2] A `letting` statement in Essence Prime [18] allows values to be set which can be computed without search.

(12,13). From Rule 1, a card that is originally hidden in the tableau is not face up until the card covering it has moved, and it cannot move until strictly after that happens (14). This uses given values from the parameter file seen in Figure 1.

$$\forall c \in \mathbb{C} \setminus \mathbb{A} : F(c) \geq F(c-1) \tag{11}$$

$$\forall c \in \mathbb{C} \setminus \mathbb{K} : T_1(c) \neq \varnothing \rightarrow [\, U(P_1(c)) \neq \varnothing \ \wedge \ U(P_1(c)) < T_1(c) \,] \tag{12}$$

$$\forall c \in \mathbb{C} \setminus \mathbb{K} : T_2(c) \neq \varnothing \rightarrow [\, U(P_2(c)) \neq \varnothing \ \wedge \ U(P_2(c)) < T_2(c) \,] \tag{13}$$

$$\forall p \in \mathbb{P} : \forall i < \texttt{firstFaceUpCards}(p) : M(\texttt{tableau}(p,i)) > M(\texttt{tableau}(p,i+1)) \\ \wedge \ U(\texttt{tableau}(p,i)) = M(\texttt{tableau}(p,i+1)) \tag{14}$$

From Rule 8: if a card does move to tableau, then its tableau parent cannot move to foundation until the built card has moved away. The simpler case is where the card has made its second tableau move, since the card has no later move to make except to foundation (15). Where the card made its first tableau move, there are two cases for whether its next move is to foundation (16) or a second tableau move (17).

$$\forall c \in \mathbb{C} : T_2(c) \neq \varnothing \rightarrow F(P_2(c)) \geq F(c) \tag{15}$$

$$\forall c \in \mathbb{C} : (T_1(c) \neq \varnothing \wedge T_2(c) = \varnothing) \rightarrow F(P_1(c)) \geq F(c) \tag{16}$$

$$\forall c \in \mathbb{C} : T_2(c) \neq \varnothing \rightarrow F(P_1(c)) \geq T_2(c) \tag{17}$$

We use the dominance from Section 2 that after the move of a built card from one pile to another, the card it was underneath moves immediately to foundation. In fact we state that the move to foundation occurs at the same stage as the second tableau move (18). Also, we note that of any pair of twin cards, at most one can make a second tableau move (19).

$$\forall c \in \mathbb{C} : T_2(c) \neq \varnothing \rightarrow F(P_1(c)) = T_2(c) \tag{18}$$

$$\forall c \in \mathbb{C} : T_2(c) = \varnothing \vee T_2(\texttt{tw}(c)) = \varnothing \tag{19}$$

From Rule 2, we have to ensure that two cards cannot be built on the same tableau parent at the same time. These are the most complex constraints involving the tableau, and their correctness involves understanding the dominance above. The constraint arises when we have two different moves with the same tableau parent: they must be of two twin cards. Either move could be either the first or second tableau move of its card, but (19) shows that there are only three cases in total. The first case in (20) is that both cards' first moves share a tableau parent. The second of the twins to move to tableau can only do so after the first has moved away. Furthermore, the first card's second move must be to foundation since otherwise the card it was built on would immediately move to foundation and could not be built on again. The second and third cases (21,22) are similar except that it is one card's second move: the two cases are divided by which occurs first.

$$\forall c \in \mathbb{C} : [\, P_1(c) \neq \varnothing \wedge P_1(c) = P_1(\texttt{tw}(c)) \wedge T_1(c) \leq T_1(\texttt{tw}(c)) \,] \rightarrow T_1(\texttt{tw}(c)) > F(c) \tag{20}$$

$$\forall c \in \mathbb{C} : [\, P_1(c) \neq \varnothing \wedge P_1(c) = P_2(\texttt{tw}(c)) \wedge T_1(c) \leq T_2(\texttt{tw}(c)) \,] \rightarrow T_2(\texttt{tw}(c)) > F(c) \tag{21}$$

$$\forall c \in \mathbb{C} : [\, P_1(c) \neq \varnothing \wedge P_1(c) = P_2(\texttt{tw}(c)) \wedge T_1(c) > T_2(\texttt{tw}(c)) \,] \rightarrow T_1(c) > F(\texttt{tw}(c)) \tag{22}$$

For moving kings to spaces, we first establish consistency between dummy values in these variables (23). A non-tableau king is first face-up in tableau after it moves to a space (24). From Rule 4, A king can only be played to a pile strictly after that pile first became empty (25). A king's first move is either to a space or to foundation (26,27): the two implications use mutually exclusive preconditions so form a disjunction. From Rule 5, two Kings cannot

be played to the same space at the same time (28): this is considerably simpler than the analogous tableau constraints because there is no need ever to move a King from one space to another.

$$\forall k \in \mathbb{K} : KP(k) = \varnothing \leftrightarrow KS(k) = \varnothing \tag{23}$$

$$\forall k \in \mathbb{K} : \neg \texttt{tableau_set}(k) \rightarrow KS(k) = U(k) \tag{24}$$

$$\forall k \in \mathbb{K} : KP(k) \neq \varnothing \rightarrow KS(k) > PS(KP(k)) \tag{25}$$

$$\forall k \in \mathbb{K} : KP(k) \neq \varnothing \rightarrow M(k) = KS(k) \tag{26}$$

$$\forall k \in \mathbb{K} : KP(k) = \varnothing \rightarrow M(k) = F(k) \tag{27}$$

$$\begin{aligned}\forall k_1, k_2 \in K : [\ k_1 \neq k_2 \wedge KP(k_1) \neq \varnothing \wedge KP(k_1) = KP(k_2)\ ] \\ \rightarrow [\ KS(k_1) > F(k_2) \vee KS(k_2) > F(k_1)\ ]\end{aligned} \tag{28}$$

Except when played with a deal size of 1, the most complicated part of the encoding is to ensure that the stock is played in the right order. When a deal size of 1 is used, all the following constraints can be omitted as stock cards can be played in any order. The complication comes first from the four different ways that a stock move can be legal, and second from the fact that the legality of a stock move at any stage depends on the previously played stock moves. Some initial constraints are straightforward. The vector SO forms a permutation (29). The SO and SI are inverses of each other (30): note that we do not use the global constraint "inverse" as it is not available in Savile Row [19, 18], which we used here. From Rule 6, the i^{th} card played from stock is first moved strictly after the $i - 1^{st}$ card (31).

We now turn to the main constraint on stock ordering (32). We do not have to constrain the last index since when only one card is left it is automatically playable as the last card in stock. The four disjuncts of (32) each correspond to the four ways that a stock card can be played legally at a given position in the stock order. We take these in the order of the disjuncts. First, a card can be played from the stock if it is now the last unplayed card in stock: this is expressed by stating all later cards in the stock were played earlier. Second, a card can be played if the previously played card was directly above it in the stock. The previously played card is given by $SI(SO(i)-1)$. We state that any cards that were originally in the stock between these two were played earlier. Third, a card can be played when it is at a position divisible by `dealN` in the stock. This is done by a reified sum: we regard a constraint as having value 1 when true and 0 when false. This is common in constraint systems including Savile Row. Our reified sum is on the cards earlier in the stock which have not been played yet. Fourth, a final reified sum states that a card can be played when it is a multiple of `dealN` later in the stock than the last card played.

$$\texttt{allDifferent}(SO) \tag{29}$$

$$\forall i \in \{1 \ldots |\texttt{stock}|\} : SI(SO(i)) = i \tag{30}$$

$$\forall i \in \{2 \ldots |\texttt{stock}|\} : M(\texttt{stock}(SI(i))) > M(\texttt{stock}(SI(i-1))) \tag{31}$$

$$\begin{aligned}&\forall i \in \{1 \ldots |\texttt{stock}| - 1\} : [\ \forall j \in \{i+1 \ldots |\texttt{stock}|\} : SO(j) < SO(i)\ ] \\ &\quad \vee [\ i < SI(SO(i)-1) \wedge (\forall k \in \{i+1 \ldots SI(SO(i)-1) - 1\} : SO(k) < SO(i))\ ] \\ &\quad \vee [\ (\sum_{j<i} SO(j) > SO(i)) \bmod \texttt{dealN} = \texttt{dealN} - 1\ ] \\ &\quad \vee [\ i > SI(SO(i)-1) \wedge (\sum_{SI(SO(i)-1)<j<i-1} SO(j) > SO(k)) \bmod \texttt{dealN} = \texttt{dealN} - 1]\end{aligned} \tag{32}$$

This rather complex constraint completes our presentation of constraints in the model. While very detailed, the presentation here still slightly simplifies some aspects of our actual model, which is written in Essence Prime [18]. For example, we have a separate integer representation of $\varnothing$ for each type (e.g. cards, stages) to make the relevant domains contiguous: this is a matter of convenience rather than for performance reasons. Some constraints are expressed slightly differently, though equivalently to the above. Finally, we have omitted some implied and dominance constraints which are not essential to correctness: these are summarised in Section B. That appendix also gives the variable names used in the Essence Prime model for each set of decision variables, to help the reader correlate the two.

Two checks reassure us that our model is correct. First, we ran our models on the first 1000 random seeds that Solvitaire reports as winnable and the first 1000 it reports as unwinnable, and we always got the same result. Second, the solution files produced by this model are difficult to understand and verify. To check the validity of the solutions, we created a small Python program to parse these solutions. This program then uses the moves proposed in the solution files to simulate a Klondike solitaire game, checking the legality of each move in the process. Should the game prove valid, the program outputs each state of the valid game, alongside all of the cards moved between states. We ran this verifier program on the solutions provided by our Essence Prime model for the first 10,000 randomly generated deals solvitaire generated: all of the solutions provided were valid.

4 Partial Models for Full Klondike

While we have achieved a complete model when disallowing worrying-back, we still consider the full game of Klondike where moves from foundation to tableau are allowed. Unfortunately, this increases the number of stages that can be needed, and we do not know of any simple way to limit this number as we were able to do above. As well as greatly enlarging the number of stages, model complexity would increase even further. This means that we cannot present a *full* model of the full game of Klondike. However, we can build *partial* models for the full game. Indeed the model for no worrying-back is one: any solution found by this model also solves the full game, but unwinnability without worrying-back might not apply to full Klondike. We now briefly discuss further partial models with the opposite property: if these report no solution then it is guaranteed that Klondike cannot be won on that instance either with or without worrying-back.[3]

Our partial models are based on careful relaxation of the rules of Klondike, making the game less constrained to win but still leaving many cases where both it and the full game are unwinnable. Our relaxations are as follows. Once a card is in the foundation, the next card in the same suit can be built to it for the rest of the game. This is a relaxation because in the real game we are allowed to "worry-back" the foundation card to tableau, preventing it being built on in foundation. Similarly, once a card is available to be tableau-built upon, it remains available for the rest of the game with one key exception. The relaxation still allows cards to be tableau-built upon even after they have been moved to foundation, without the need to be worried-back first. The exception is that we disallow tableau-building two cards onto the same card at the same time. This completes the first model, which we call "Partial with Strict Stock". A second partial model uses a considerable relaxation of the stock rule. We call this "Partial with Relaxed Stock". This only constrains early stock moves. Initially all cards at a multiple of n from the start of the stock, and the last card, are available to

[3] Full details of these models are discussed in [7].

Table 1 Results on Klondike using our constraint models and Solvitaire. The first line for each model gives the total for all instances, with breakdowns on result in the following two lines. The nodes is the mean nodes reported by Kissat. Mean times in seconds are given for Savile Row, Kissat solving, and their sum. The final column gives the maximum of total time for any layout.

Model/Result	Count	Nodes	Time (seconds)			
			Mean			Max
			SR	Solver	Total	Total
No Worry-Back	**10000**	**748134**	**11.77**	**27.52**	**39.29**	**470.10**
Unwinnable	1906	327873	10.75	14.06	24.81	470.10
Winnable	8094	847098	12.01	30.69	42.7	206.64
Unknown	0	–	–	–	–	–
Partial (Strict)	**10000**	**885253**	**4.67**	**30.26**	**34.93**	**1373.54**
Unwinnable	1328	881784	3.92	37.89	41.81	1373.54
Unknown	8672	885784	4.78	29.10	33.88	424.64
Partial (Relaxed)	**10000**	**24303**	**1.18**	**0.20**	**1.38**	**21.76**
Unwinnable	1042	54884	0.97	1.30	2.00	21.76
Unknown	8958	20746	1.20	0.11	1.31	2.84
Solvitaire	**10000**	$\mathbf{1.25 \times 10^8}$			**154.00**	**3600**
Unwinnable	1639	6.00×10^7			76.49	3529.24
Winnable	7991	8.45×10^6			10.34	3538.33
Unknown	370	2.93×10^9			3600	36000

play. Other cards become available when the card directly after them in the stock has been moved, or any card in an earlier multiple of n in the stock has been moved. This proves fewer instances unwinnable but runs much faster.

5 Experimental Results

The model and parameter files are written in Essence Prime [18] and solved with Savile Row [20] version 1.10.1 with some minor changes - the version with changes is included in our supplementary material. Savile Row uses the constraint solver Minion [11] for preprocessing to optimise the model, and then has available a number of different backend solvers. For the backend solver we used Kissat 4.0.2 [1][4] following exploratory experiments in which it outperformed Minion and Chuffed [6]. The experiments were carried out on a high-performance cluster with 48-core AMD EPYC 7643 2.3GHz CPUs, with the job memory limit set to 8GB. Solvers ran single-threaded, except that Savile Row as a Java program may have had automated garbage collection run in a separate thread.

Solvitaire's code and experimental results on a line-by-line basis are openly available [3, 9]. This allows us to compare our results in detail with Solvitaire on identical instances. Most importantly, none of the instances any model reported as impossible were shown to be winnable by Solvitaire. While not a guarantee of correctness, it is a great reassurance especially considering how many layouts were claimed to be impossible.

[4] `https://github.com/arminbiere/kissat`

Table 1 gives our experimental results on the first 10,000 seeds in the Solvitaire experimental data set [9]. As well as each constraint model discussed in this paper we report results using Solvitaire for Klondike without worrying-back on the same machines with a one hour CPU timeout for a direct comparison. Number of nodes is as reported by Kissat and Solvitaire respectively so are not directly comparable. We can see that our No Worry-Back constraint model is able to solve all instances in a max of 470.1s while Solvitaire fails to solve 370 after one hour. Solvitaire is faster on average on winnable instances but is slower on unwinnable instances. While not seen in the table, Solvitaire is able to solve many winnable and unwinnable instances very fast compared to No Worry-Back, which we will discuss further in Section 6. To give an indication of model size, winnable instances given to Kissat in the No Worry-Back model had a mean of 81,552 SAT variables and 648,422 clauses, s.d. 1,592 and 12,826. Some unwinnable instances were greatly reduced in size during prerpocessing but the largest ones were similar with a maximum number of variables of 83,722 variables and 668,313 clauses.

Our result of 13% of layouts unwinnable for the Partial (Strict) model in Table 1 greatly improves on any previous approach for proving games unwinnable without exhaustive state-based search. Previous results have been 2.24% of layouts proved unwinnable [25], 3.33% [8] and 8.56% [2]. Of the 10,000 seeds, [9] reports that 8,131 are winnable, 1,868 unwinnable, and one unknown.[5] This means that when using the full stock rules, our model was able to prove that 71% of the impossible layouts were unsolvable using the relaxed version of Klondike.

Using the relaxed stock rule, more than 55% of unwinnable layouts were proved using an average of just over 1s CPU time. In Section 6 we show that we can combine our models with Solvitaire to achieve better performance than either alone.

We also experimented on instances which Solvitaire was unable to resolve either way among the 1,000,000 it experimented on [9] for three different games. For Klondike without worrying-back, Solvitaire left 249 layouts unresolved. Our model was able to solve *all* of these, proving 66 have solutions and 183 do not. For full Klondike, Solvitaire left 157 layouts unresolved for deal-3 and 1145 for deal-1 Klondike. Our models were able to prove 63 were unwinnable for regular Klondike and as many as 522 for deal-1. The longest CPU time for any of these instances was less than 4s for deal-1 and less than 213s for deal-3. This compares extremely favorably with the fact that Solvitaire failed on all these instances after hours of CPU time each. Additionally, 11 of the newly winnable instances for no worrying-back were also newly winnable for full Klondike deal-3. Solvitaire gave the 95% confidence interval of the winnability of deal-3 Klondike as $81.945 \pm 0.084\%$ and of deal-1 as 90.480 ± 0.116 [4]. Our resolution of unknown layouts allow us to improve that estimate slightly to 81.942 ± 0.080 for deal-3 and 90.454 ± 0.090 for deal-1, using the same calculation method [4]. We report a much larger improvement for Klondike without worrying-back in Section 6.3.

6 Scheduling Solvers

6.1 Solving Complementarity

At this point, in addition to Solvitaire, we have three constraint models to support the solving of Klondike, including:

[5] The number unknown is much lower than in Table 1 as [9] used significantly more resources.

Table 2 Number of instances (over a subset of 10,000 instances) where a model wins or solve the instance within x seconds of the best approach's solving time on the same instance ($x \in \{1, 2, 4\}$).

Model	Klondike Without Worrying-Back				Full Klondike			
	#best	#best +1s	#best +2s	#best +4s	#best	#best +1s	#best +2s	#best +4s
Solvitaire	9221	9254	9275	9306	9445	9481	9506	9535
`no worry-back`	441	441	443	444	173	176	181	187
`relaxed`	329	608	784	932	351	612	787	933
`strict`	9	99	348	580	15	108	359	589

- Two models for the relaxed variants of Klondike. These include the relaxed stock and full stock models from Section 4, denoted as `relaxed` and `strict`, respectively. As empirically validated in Section 5, those models can prove unwinnability very quickly for a large number of cases, but they cannot solve winnable instances.
- Our proposed model in Section 3, denoted as `no worry-back`, for solving Klondike *without* worrying-back. This model can also be used for solving winnable instances of Full Klondike. However, the model cannot prove unwinnability for this Klondike version.

The two relaxed constraint models are *incomplete* for both Klondike versions: statements of unwinnability are always correct but false positives are possible. The `no worry-back` model is *complete* for the Klondike without worrying-back but is incomplete for full Klondike: a false negative is possible if full Klondike can *only* be won with at least one card worried back. Solvitaire can be used as a complete solver for both Klondike versions.

The three constraint models and Solvitaire can potentially complement each other. To verify this hypothesis, we run them on both versions of Klondike on $10,000$ random instances, using the same set for each technique. We then count the number of instances where each model performs best, as listed in Table 2. Additionally, for each model, we also count the number of instances where the solving time is within x seconds ($x \in \{1, 2, 4\}$) of the best-performing model on the corresponding instance. Note that the counting only applies to cases where a valid answer is returned, e.g., a winnable answer from the `relaxed` model will not be counted even if the run is finished very quickly.

Although Solvitaire appears to win on a majority of instances for both Klondike versions, the three constraint models do show complementary strengths to Solvitaire. For example, on Klondike without worrying-back, the `no worry-back` model wins on 441 instances, while the two relaxed models (`relaxed` and `strict`) perform competitively to the winning approach on several cases. For example, although the `strict` model only wins on 9 instances, it becomes competitive to the best solving approach on 99 instances when we allow just one extra second to the best approach's solving time. The number of competitive cases also increases significantly as the amount of extra time allowed increases.

6.2 Constraint-based Solver Scheduling

Our complementarity analysis suggests that it can be beneficial to combine those models and Solvitaire together to speed up the solving of both Klondike versions. Selecting the best algorithm(s) from a portfolio of algorithms with complementary strengths is a well-studied topic and several successful automated algorithm selection techniques have been

developed [15, 14]. However, they often assume the existence of high-quality *instance features*, which are crucial for building effective machine learning models to predict the best algorithms or to build an instance-specific schedule of solvers [22, 24]. Constructing informative instance features is a non-trivial task, and although it may be possible to do so for Klondike games, we leave this option for future work.

In this work, we adopt a simple approach where we build a single solving schedule for *all* problem instances. We model the scheduling task as an optimisation problem [21, 13] and find the optimal solving schedule using constraint programming. Concretely, given a *training* instance set $\mathcal{I}$, a portfolio of n algorithms $\mathcal{A} = \{a_1, a_2, ..., a_n\}$ with *complementary* strengths, and a cutoff time T that specifies the time limit we can spend on solving each instance. Assuming that we can collect information about the performance of all algorithms in the portfolio $\mathcal{A}$ on the instance set $\mathcal{I}$, an algorithm schedule is defined as a sequence of algorithms chosen from $\mathcal{A}$ and the maximum amount of solving time assigned to each algorithm in the sequence (the total amount of time must not exceed the cutoff time T). The scheduling problem aims at finding a schedule that can solve as many instances in $\mathcal{I}$ as possible within the shortest amount of time. We model this optimisation problem as a constraint model and solve it using the chuffed constraint solver [6].[6]

Note that the application of a schedule in the *test* phase (i.e., on instances unseen during the schedule building process) is slightly different. Given that an incomplete algorithm may finish before its maximum assigned runtime without producing a conclusive answer, we allocate the total *leftover time* (the positive difference between the maximum execution time of all incomplete algorithms in the schedule and the actual execution time of those algorithms when inconclusive) to the last complete algorithm in the schedule (not surprisingly, in our experiments, the last algorithm in the optimal schedules returned by our scheduling constraint model is always a complete algorithm). This approach ensures that we can utilise the whole cutoff time during the solving process on unseen instances.

Current algorithm scheduling techniques only make use of each solver once in the schedule [5, 17]. We realised that it can be beneficial to allow a solver to be paused and then later resumed in the schedule. For example, on a 70-second timeout for full Klondike, the optimal schedule would be running Solvitaire first, suspending after 3.9 seconds, then `relaxed` with a cutoff time of 12 seconds, then `no worry-back` with a cutoff time of 50 seconds, and finally resuming Solvitaire for the remaining time. This results in approximately 2 percentage points more determined instances compared to just running Solvitaire. This might seem counterintuitive but is explained by the fact that Solvitaire is able to prove many instances winnable and others unwinnable in the first 3.9 seconds.

Performance metric (PAR10). Note that the objective of our scheduling task consists of two components: (i) the number of instances solved by the schedule, and (ii) the average solving time on the solved instances. A typical approach to aggregate them into a single objective is to use the *Penalised Average Runtime (PAR10)* [16], where the runtime of each unsolved instance is counted as ten times the cutoff time. We will adopt this metric as the objective function when building our schedule and in our evaluation.

The constraint model for this scheduling task together with a detailed explanation on the model are available at `https://github.com/turingfan/CP2025-Klondike`. Although our scheduling constraint model allows an arbitrary number of repetitions for each individual

[6] `https://github.com/chuffed/chuffed`

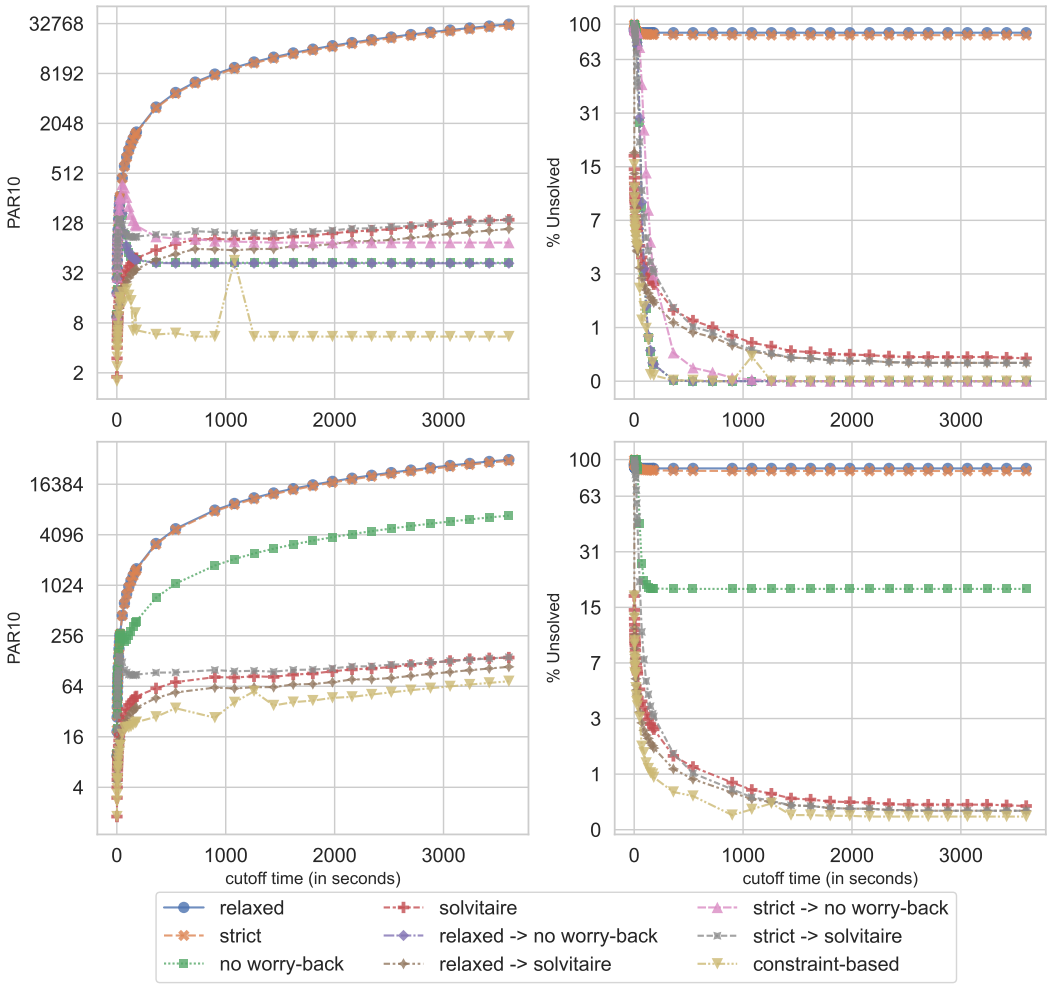


Figure 2 PAR10 score (lower is better) and the percentage of unsolved instances of the baselines and our constraint-based schedule on different cutoff times on Klondike without worrying-back (top row) and full Klondike (bottom row). y-axis is in log scale of 2.

approach, for practical reasons (due to limited computational resources), in our experiments, we limit the ability of being repeated to Solvitaire only as it is the one that probably provides the most potential gain in performance.

6.3 Results

Baseline schedules. We use each individual solving approach (`relaxed`, `strict`, `no worry-back` and Solvitaire) as baselines in our evaluation. Moreover, since the incomplete approaches often finish in a short amount of time, to utilise the complementary strengths of both incomplete and complete approaches, an intuitive approach is to build a schedule that runs a chosen incomplete solver first, followed by a complete solver if the former one does not return a guaranteed answer. Therefore, we also consider four extra baseline schedules of this type in our evaluation: `relaxed -> solvitaire`, `relaxed -> no worry-back`, `strict -> solvitaire`, and `strict -> no worry-back`.

We build the constraint-based schedule using 1000 training instances and evaluate the performance of all schedules, including the baselines, on a separate test set of 9000 instances. Figure 2 shows performance on the test set of all approaches for both Klondike versions. Performance metrics include PAR10 score and the percentage of unsolved instances.

Our results clearly indicate the advantage offered by our constraint-based scheduling approach. For both Klondike versions, the constraint-based schedules achieve the best PAR10 scores on all cases except a very few, where its scores are very close to the best ones. On Klondike without worrying back, although the three strongest baselines (`no worry-back`, `relaxed -> no worry-back` and `strict -> no worry-back`) and the constraint-based schedules can solve 100% of the test set given a sufficient cutoff time, the constraint-based schedule is almost always the fastest one as indicated by the low PAR10 values. On full Klondike where the games are harder to solve, the constraint-based schedules not only consistently achieve the highest amount of solved instances but also the lowest PAR10 values across all cutoff times.

Unsurprisingly, among the baseline schedules, the incomplete models (i.e., `relaxed` and `strict` for Klondike without worrying-back, and those two plus `no worry-back` for full Klondike), when being used alone, are the weakest. However, when combined with a complete approach, they can offer significant improvement in performance. For example, on Full Klondike, the combination of `relaxed` and Solvitaire is consistently better than Solvitaire alone across all cutoff times. Those observations further strengthen our hypothesis about the complementary strengths of those various solving approaches.

Examining the optimal schedules produced by our constraint-based approach, we observe a consistent pattern across all cutoff times for each Klondike version. Specifically, for Klondike without worrying-back, the schedules typically begin with a brief run of Solvitaire (less than 2 seconds per instance on average), followed by a short run of the `relaxed` model (less than 1 second per instance on average), and conclude with `no worry-back` for the remainder of the allotted time. For Full Klondike, the schedules also start with a short run of Solvitaire (less than 2s on average), then `relaxed` (less than 1s on average). Unsurprisingly, Solvitaire is always the last solver in the schedule, but for all cases where the cutoff time is sufficiently large (more than 50s), `no worry-back` is also called for a short amount of time (2s on average) just before Solvitaire (i.e., right after `relaxed`). The consistent appearance of the two relaxed models and `no worry-back` in the optimal schedules of both Klondike versions once again strongly confirms the complementary of those models to Solvitaire.

The optimal schedule for one hour for Klondike without worrying-back is 6.6s Solvitaire, 5.9s `relaxed`, then remainder using `no worry-back`. Using this, we solved one million new seeds. We solved all instances with 815548 winnable and 184452 unwinnable. Results are shown in Table 3. Mean time across all instances was 3.483s and the longest instance took 1,874s. Combined with existing Solvitaire results and the 249 results described earlier, this gives us 2 million random seeds with 1630728 winnable and 369272 not. For Klondike without worrying-back, we now have a 95% confidence interval for winnability of $81.536 \pm 0.055\%$. This is a considerable improvement over the previous result of $81.524 \pm 0.089\%$ [4].

All winnable instances above are also winnable for Full Klondike, but 184452 seeds remain which may or may not be. To resolve these, we constructed a new schedule for those we know are unwinnable without worrying-back. This was 18s of Solvitaire, then the Partial model with Strict Stock to completion, then resuming Solvitaire for the remainder. This method was able to prove 4253 winnable and 177949 unwinnable but left 2250 unknown after one hour. While this number of unknowns is too many to allow us to improve the winnability estimate, the total run time combined for these two experiments is only about 110 days CPU time, compared to the 960 days reported by [4] for their equivalent experiments.

Table 3 Results on seeds 1,000,001 to 2,000,000. Times are mean times in ms, and we also give the number of instances in that category. Results when the partial model reports winnable might be incorrect and are discarded. 815548 seeds are winnable and 184452 not.

Stage	Result	Time (s)				# Solved
		Solvitaire	Relaxed	No Worry-Back	Total	
0:solvitaire	winnable	190.0	n/a	n/a	190.0	782362
	unwinnable	681.5	n/a	n/a	681.5	133934
1:relaxed	unwinnable	6601.2	4746.7	n/a	11347.9	20514
2:no worry-back	winnable	6602.7	5950.6	44455.6	57008.9	33186
	unwinnable	6601.8	5895.9	24788.3	37286.0	30004

7 Conclusions

We have described two classes of constraint models for the patience/solitaire game Klondike. One class is complete for the game where worrying-back is disallowed, and is able to solve all instances we have tested it on, with a maximum time across all instances of just over half an hour. The second class is two partial models for the game where worrying-back is allowed, which can only confirm unwinnability of layouts. All models have passed extensive testing.

Our results show that our models have complementary strengths to the state-of-the-art solver Solvitaire. This suggested that a combination of the two approaches could be better than either. We therefore investigated the potential of building a schedule of solvers by modelling the scheduling task itself as a constraint model. Our results on a range of cutoff time limits up to 1 hour reveal that it is indeed beneficial to combine the proposed models with the state-of-the-art solver Solvitaire: the obtained schedules perform better than both Solvitaire and the individual models on the whole range of cutoff times considered.

Our constraint models are able to resolve many layouts the state-of-the-art solver Solvitaire is unable to resolve. As a result, we are able to slightly improve the best-known estimate of winnability of thoughtful Klondike where worrying-back is allowed. We got far more impressive results where worrying-back is disallowed. Using an optimal schedule we were able to solve a million new instances at less than 3.5s per instances, and greatly improve the best estimate of winnability of that game.

Among future work, we would like to refine our methods so that they can also significantly improve the winnability estimate of full Klondike. Among other techniques we might be able to develop instance features for Klondike to enable machine learning to select the best solver per instance. We would also like to extend our work to other patience and solitaire games.

References

1 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.

2 Ronald Bjarnason, Prasad Tadepalli, and Alan Fern. Searching solitaire in real time. *ICGA Journal*, 30(3):131–142, 2007. doi:10.3233/ICG-2007-30302.

3 Charlie Blake and Ian Gent. thecharlesblake/Solvitaire: Release for Zenodo DOI-issuing (v0.10.2), November 2019. Zenodo. doi:10.5281/zenodo.3529524.

4 Charlie Blake and Ian P. Gent. The winnability of klondike solitaire and many other patience games, 2024. arXiv:1906.12314.

5 Derek Bridge, Eoin O'Mahony, and Barry O'Sullivan. Case-based reasoning for autonomous constraint solving. In *Autonomous search*, pages 73–95. Springer, 2012. doi:10.1007/978-3-642-21434-9_4.

6 Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, 2018. Available from https://github.com/chuffed/chuffed/.

7 Nguyen Dang, Ian P. Gent, Peter Nightingale, Felix Ulrich-Oltean, and Jack Waller. Constraint models for relaxed Klondike variants. In *ModRef 2024: The 23rd workshop on Constraint Modelling and Reformulation*, 2024. URL: https://modref.github.io/ModRef2024.html.

8 Johan de Ruiter. Counting classes of klondike solitaire configurations. Technical Report Internal Report 2012-9, Leiden Institute of Advanced Computer Science, 2012. URL: https://theses.liacs.nl/pdf/2012-09JohandeRuiter.pdf.

9 Ian P. Gent and Charles Blake. Patience Experimental Results (Version 5), January 2023. doi:10.6084/m9.figshare.8311070.v5.

10 Ian P Gent, Chris Jefferson, Tom Kelsey, Inês Lynce, Ian Miguel, Peter Nightingale, Barbara M Smith, and S Armagan Tarim. Search in the patience game 'black hole'. *AI Communications*, 20(3):211–226, 2007. URL: http://content.iospress.com/articles/ai-communications/aic405.

11 IP Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. *ECAI*, pages 98–102, 2006.

12 Paul Jensen. Celebrating 30 years of Microsoft Solitaire with those oh-so-familiar bouncing cards, May 2020. Xbox.com. URL: https://web.archive.org/web/20200522210053/https://news.xbox.com/en-us/2020/05/22/celebrating-30-years-microsoft-solitaire/.

13 Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In *Principles and Practice of Constraint Programming–CP 2011: 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings 17*, pages 454–469. Springer, 2011. doi:10.1007/978-3-642-23786-7_35.

14 Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary computation*, 27(1):3–45, 2019. doi:10.1162/EVCO_A_00242.

15 Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data mining and constraint programming: Foundations of a cross-disciplinary approach*, pages 149–190. Springer, 2016. doi:10.1007/978-3-319-50137-6_7.

16 Marius Lindauer, Jan N van Rijn, and Lars Kotthoff. The algorithm selection competitions 2015 and 2017. *Artificial Intelligence*, 272:86–100, 2019. doi:10.1016/J.ARTINT.2018.10.004.

17 T. Liu, R. Amadini, M. Gabbrielli, and J. Mauro. sunny-as2: Enhancing sunny for algorithm selection. *Journal of Artificial Intelligence Research*, 72:329–376, 2021. doi:10.1613/JAIR.1.13116.

18 Peter Nightingale. Savile Row manual, 2021. doi:10.48550/arXiv.2201.03472.

19 Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, 2017. doi:10.1016/J.ARTINT.2017.07.001.

20 Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, 2017. doi:10.1016/J.ARTINT.2017.07.001.

21 Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish conference on artificial intelligence and cognitive science*, pages 210–216, 2008.

22 Hadar Shavit and Holger H Hoos. Revisiting satzilla features in 2024. In *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, pages 27–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SAT.2024.27.

23 Barbara M. Smith. Caching search states in permutation problems. In Peter van Beek, editor, *Principles and Practice of Constraint Programming – CP 2005*, pages 637–651, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11564751_47.

24 Felix Ulrich-Oltean, Peter Nightingale, and James Alfred Walker. Learning to select sat encodings for pseudo-boolean and linear integer constraints. *Constraints*, 28(3):397–426, 2023. doi:10.1007/S10601-023-09364-1.

25 M Voima. Klondike solitaire solvability. Technical Report Internal Report 2012-9, Tampere University of Applied Sciences, 2021. URL: https://urn.fi/URN:NBN:fi:amk-2021060213520.

26 Jan Wolter. solsolve: a solving workbench for various solitaire games, December 2014. Google Code Archive. URL: https://code.google.com/archive/p/solsolve/.

A Appendix: Glossary of Patience/Solitaire Terms

While extremely widely known, both terminology and precise rules of Klondike can be unclear and/or vary. We have given the rules we use in the main text, but for precision and reviewers' convenience we define the key terms used in those rules, following the "Terminology of Patience Games" section by Blake and Gent [4]. We describe these in terms of the standard western deck of 52 cards.

Colour: Hearts and Diamonds are red while Spades and Clubs are black.

Deal-n: The number of cards that are moved from the stock to waste in a single movement. Standard Klondike is deal-3. A major variant is deal-1: with infinite redeals allowed, this in fact allows us to play the stock in any order.

Deck: A set of 52 cards, comprising one copy of each rank/suit combination.

Face-down: A tableau card placed with its reverse facing up, not allowed to be moved or played to.

Foundation: Initially empty piles in which to build up from A to K in suit. Only Aces can be placed in an empty pile. Cards can be moved to the foundation pile, provided that the card of the same suit and one lower in value is already in the same pile.

Group: A sequence of cards in a tableau pile which are all face-up and in valid build sequence. All the cards may be moved together in the tableau if the highest card in the sequence is legal to be moved.

Hidden: A card is hidden when it is face-down in the tableau.

Layout: A particular placing down of the deck onto the tableau and stock, typically random.

Pile: One of 7 locations in the tableau that we can build cards down in alternating colours. If all face-up cards are moved from a pile, the topmost face-down card on that pile is turned face-up

Rank: One of (in order) Ace (valued as 1), 2 to 10, Jack (11), Queen (12), King (13).

Redeal: Once the stock is exhausted we are allowed to start from the beginning again, keeping the stock cards in the same order.

Space: A tableau pile that currently has no cards in it.

Stock: The 24 cards of the deck not in the tableau are initially placed in an ordered pile called the "stock". In deal-n Klondike, we can draw n cards from the stock and place them in a separate pile named "waste", maintaining the order from the stock.

Suit: One of "Spades" (written ♠), "Hearts" (♡), "Clubs" (♣), and "Diamonds" (♢)

Tableau: The 7 piles which initially range from 1 to 7 cards so contain 28 in total. Only the top card of each pile is originally face-up. Face-up cards can be moved from one pile on the tableau to another tableau pile, provided that the card they are being placed on is of a different colour and that the value of the card being placed is one less than the card being placed on.

Thoughtful: The variant of Klondike we study in this paper, where we know the location of all cards at the start of the game but the rules about cards being unplayable when face-down still apply.

Waste: The pile we move cards from the stock to. The topmost card from the waste can be placed on a tableau pile or a foundation pile, if it is legal to do so. When no more stock cards exist we can take the waste as the stock again and redeal.

Win: A layout is considered won when each Foundation contains the relevant K.

Worrying-back: Building a card from the foundation to a legal place on the tableau.

B Appendix: Essence Prime Model of Klondike Without Worrying-back

In this Appendix we give details which should help the reader who wishes to correlate the presentation of our model in Section 3 with the actual Essence Prime model available at `https://github.com/turingfan/CP2025-Klondike`. We also summarise some implied and dominance constraints which we did not present in Section 3.

$M(c)$: The first stage at which c is moved. Model name: `first_moved_stage`

$F(c)$: The stage at which c is played to foundation. Model name: `foundation_stage`

$U(c)$: The first stage at which c is face up in the tableau and so can potentially be built on, or $\varnothing$ if it never is. Model name: `tableau_face_up_stage`.

$T_1(c)$: The first stage at which c is built on another card in the tableau (or $\varnothing$ if never). Model name: `first_tableau_stage`

$P_1(c)$: If $T_1(c)$ is not $\varnothing$, the card that c was built onto in the tableau at that stage. The domain of $P_1(c)$ is set to be either dummy or a card of opposite colour and one rank higher. Setting $p = (c + 1 + \texttt{numRanks}) \mod (2 * \texttt{numRanks})$, this means that for non-Kings the domain of $P_1(c)$ $\{\varnothing, p, p + (2 * \texttt{numRanks})\}$. Model name: `first_tableau_parent`

$T_2(c)$: The second stage at which c is built onto another card in the tableau (or $\varnothing$ if never). Model name: `second_tableau_stage`

$P_2(c)$: If $T_2(c)$ is not $\varnothing$, the card that c was built onto in the tableau at that stage. Model name: `second_tableau_parent`. The domain of $P_2(c)$ is the same as $P_1(c)$

$PS(p)$: For each pile p, the stage at which p is first empty and is therefore a space a King can be moved into. Model name: `stage_first_space` indexed by pile.

$KS(k)$: For each king k, the stage at which the King is played to a space, or $\varnothing$ if never. Model name: `king_played_space_stage` indexed by suits as there is one king per suit.

$KP(k)$: The tableau pile the King k is played to if it moves to a space, or $\varnothing$ if none. Model name: `king_pile` indexed by suits.

$SI(i)$: The i^{th} card played from stock is $\texttt{stock}(SI(i))$. Model name: `stock_index`

$SO(i)$: Card $\texttt{stock}(i)$ is the $SO(i)^{th}$ card played from the stock. Model name: `stock_order`

We have a number of additional constraints which may help reduce search but are not important to correctness. They are either implied constraints or dominance constraints.

- For any card c originally face-up in the tableau, we set its value of $U(c) = 1$. Although simple, this is technically a dominance.
- A card tableau-built until after at least one of the cards of opposite colour and one higher rank is face-up in the tableau. This is an implied constraint since the value of P_1 will force it, but it can propagate before P_1 is known.

- An implied constraint is that if one possible tableau parent is not face up until both a card and its twin are first played to tableau, then one twin must move from tableau to the foundation before the other twin moves to tableau.
- We add the constraint from the relaxed stock model summarised in Section 4, which in this context is now an implied constraint.
- We add another set of variables $SIM(i)$ for indexes in the stock. The value of $SIM(i)$ is the value of $M(c)$ for the card $c = \texttt{stock}(SI(i))$, i.e. the first move stage of the i^{th} card played from stock. We then add the implied constraint that the values of $SIM(i)$ strictly increase with i.
- We have a dominance constraint that a king which starts on the bottom of a tableau pile need never move to a space since it is in effect already in one.

Unit Types for MiniZinc

Jip J. Dekker
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Jason Nguyen
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Peter J. Stuckey
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Guido Tack
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Abstract

Discrete optimization models often reason about discrete sets of objects, but discrete optimization solvers only deal with integers. One of the key challenges when building models for discrete optimization problems is avoiding bugs. Because the model only defines constraints, decisions, and an objective that are then run on a solver, bugs in the model can be very difficult to track down. Hence, modelling languages should have strong type systems to detect as many bugs as possible at the modelling level. In this paper, we propose unit types for MiniZinc. Unit types allow us to differentiate between different integers appearing in the model. Almost all integer decisions in models are either about a set of objects or some measurable resource type. Using unit types, we can add more type safety to our models by avoiding confusion of decisions on different resource types. Compared to other programming languages, unit types in our proposal are unusual. MiniZinc models often deal with multiple levels of granularity of the same resource, e.g., scheduling to the minute, but doing resource allocation on the half day, or use an unspecified granularity, e.g., the same job-shop scheduling model could use task durations given in minutes or days. Our proposed unit types also differentiate between *coordinate* unit types, e.g., the time when an event occurred, and the usual *delta* unit types, e.g., the time difference between two events. Errors arising from mixing coordinate and delta types can be very challenging to debug, so we extend the type system to track this for us.

2012 ACM Subject Classification Software and its engineering → Constraint and logic languages

Keywords and phrases Modelling, Type Safety, Unit Types

Digital Object Identifier 10.4230/LIPIcs.CP.2025.10

Funding This research was partially funded by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Project ID IC200100009.

1 Introduction

Consider the following MiniZinc model of the knapsack problem where we need to choose exactly `k` different products from a given set `PRODUCT`, so that the total weight is under the limit, and we maximize profit.

© Jip J. Dekker, Jason Nguyen, Peter J. Stuckey, and Guido Tack;
licensed under Creative Commons License CC-BY 4.0

```
1 int: k;        % number of products to choose
2 int: limit;    % available weight limit
3 enum PRODUCT; % set of products available
4 array[PRODUCT] of int: wght;
5 array[PRODUCT] of int: profit;
6 array[1..k] of var PRODUCT: chosen;
7 constraint all_different(chosen);
8 constraint sum(i in 1..k)(profit[chosen[i]]) <= limit;
9 solve maximize sum(i in 1..k)(profit[chosen[i]]);
```

The model runs and will give us answers, but unfortunately it does not define the correct problem. The modeller has used `profit` instead of `wght` on line 8. In line 8 we are summing up profits to see that they are under the weight limit. This is a *unit type error* of the model, comparing integers that represent two different kinds of things.

While there are unit type extensions for almost any programming language, in practice they are not commonly used. The problem is that the *overhead* of adding unit types to the program has to pay off in comparison to the extra development time to be worth it. For safety critical code this is clearly worthwhile (as the sad demise of the $125M Mars Climate Orbiter indicates[1]). Given that debugging constraint models can be quite difficult, particularly if the solver simply fails after a large amount of computation, an important role of types in modelling languages is to provide type safety. Many subtle errors can be avoided if we use strong type checking based on the types. Here we introduce unit types for MiniZinc. Interestingly, the question of unit types for *modelling languages* for combinatorial problems is different from that of procedural languages, and we find the necessity for introducing new kinds of unit types to help prevent type errors in models.

The contributions of this paper are:

- A proposed definition of unit types for MiniZinc, including dimensions and units.
- The introduction of *abstract units*, and coercions between them, which results from the fact that we are often dealing with integer decisions, so arbitrary multiplication or division of units is not obviously meaningful.
- The introduction of *counting types*, used in models which count the occurrence of objects, a new form of unit type.
- The introduction of *coordinate* instead of the usual *difference* unit types to differentiate variables representing coordinates from those representing distances. These have been defined for other programming languages but less frequently.
- The definition of a unit type checker based on type erasure at the model level, resulting in zero overhead at solving time compared to models that do not use unit types.
- An exploration of existing MiniZinc models to determine how many of them could improve type safety utilizing unit types.

2 Unit Types

Unit types are refinements of numeric types that attach units of measurement to the numeric objects. They aim to find more type errors. Note that we introduce unit types as an addition on top of the current MiniZinc type system. Although the enhanced type system might reject models that use unit types, it does not influence the correctness of existing MiniZinc models.

[1] An investigation indicated that the failure resulted from a navigational error due to commands from Earth being sent in English units (in this case, pound-seconds) without being converted into the metric standard (Newton-seconds).

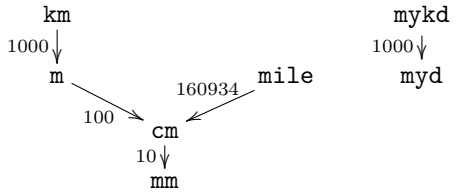


Figure 1 Graph for dimension `distance`.

A *dimension* is a kind of measurement, for example distance, time, mass, and worth are basic dimensions. Dimensions can be declared as "`unit type` *dimension*". We assume m distinct dimensions are available. A *basic unit* is an identifier such as `metre` (metre), `sec` (second), `kg` (kilogram), or `dollar` which has a dimension. A basic unit type can then be *declared* as "`unit` *dimension*`:` u", where $dim(u) = dimension$. Or it can also be *derived* from an earlier declared or defined unit type u' as "`unit` *dimension*`:` u `=` c`@`u'" where $dim(u) = dim(u') = dimension$. The basic unit types for a dimension *dimension* define a graph with a node for each basic unit type and edges (u, u') with weight c for each declared unit type definition.[2]

One basic unit type u can be *downcast* to another v of the same dimension by multiplying by the product of constants appearing in the path from u to v, denoted $\downarrow(u, v)$, which equals $\bot$ if no path exists. We use $\bot$ to represent a type error/ill-defined type. We define the *meet*, largest common subunit, of two basic unit types of the same dimension $\sqcap_b(u, v)$ as the nearest common ancestor of u and v in the graph, or $\bot$ if none exists.

Figure 1 shows the graph resulting for dimension `distance` from the sample declarations on the left below. For example, $\sqcap_b$(`km`,`mile`) = `cm`, and $\sqcap_b$(`km`,`mykd`) = $\bot$; similarly $\downarrow$(`mile`,`mm`) = 1609394×10, while $\downarrow$(`mm`,`km`) = $\bot$. Note how we can have multiple declared basic units for the same dimension, but we cannot coerce from one to the other.

```
1 unit type distance;                unit type time;
2 unit distance: mm;                 unit time: sec;
3 unit distance: cm = 10@mm;         unit time: minute = 60@sec;
4 unit distance: metre = 100@cm;     unit time: hour = 60@minute;
5 unit distance: km = 1000@m;        unit type: mass;
6 unit distance: mile = 160934@cm;   unit mass: gram;
7 unit distance: myd;                unit mass: kg = 1000@gram;
8 unit distance: mykd = 1000@myd;    unit type worth;
9                                    unit worth: dollar;
```

A (complex) unit type u for a numeric object o takes the form $u \equiv b_1^{n_1} b_2^{n_2} \cdots b_m^{n_m}$, where each $b_i, 1 \leq i \leq m$ is a basic unit type for dimension i, and each $n_i, 1 \leq i \leq m$ is an integer. The dimension of the type u is $dim(b_1)^{n_1} \cdots dim(b_2)^{n_m}$. For example, a speed may have unit type `metre/sec`, which is technically `metre`^{1}`sec`$^{-1}$`kg`^{0}`dollar`^{0} and has dimension `distance`^{1}`time`$^{-1}$`mass`^{0}`worth`^{0}. Note that we do not allow complex unit types with two basic unit types of the same dimension. We usually omit writing terms where $n_i = 0$. We denote by $\mathbf{1}$ the dimensionless unit type (where $n_i = 0, 1 \leq i \leq m$).

[2] Usually c is an integer. Floating-point constants c are allowed, but they result in automatic casts of integer expressions to float.

We can add shorthands for dimension expressions and (complex) unit type expressions. A declaration "`unit type` *short* `=` *de*" defines *short* as a shorthand for the dimension expression *de*. *de* takes the form of *de*`*`*de*, *de*`/`*de*, de^n, or d where d is a previously defined dimension or shorthand and n is an integer. Similarly, (complex) unit type shorthands can be defined as "`unit` *dimension*`:` *u* `=` *ue*" where *ue* is defined as above except that d must be a basic or shorthand unit type, and $dim(ue) = dimension$. For example, the declarations below define velocity as a shorthand for distance over time, and a unit of velocity which is always treated as its two components.

```
1 unit type velocity = distance/time;
2 unit velocity: kph = km/hour;
```

We extend the meet operation to a single dimension with power as follows: $\sqcap_d(b_i^{n_i}, {b'_i}^{n'_i})$ equals $\bot$ if $dim(b_i) \neq dim(b'_i)$ or $n_i \neq n'_i$; otherwise it equals $\sqcap_b(b_i, b'_i)^{n_i}, n_i \geq 0$ or $b_i^{n_i}$ if $\downarrow(b_i, b'_i) \neq \bot$, or ${b'_i}^{n_i}$ if $\downarrow(b'_i, b_i) \neq \bot$, or $\bot$ otherwise. So for positive powers we use the meet operation, for negative powers we require that one unit is downcastable to the other, and the resulting type is the downcastable type, inverse powers swap the direction to smaller units.

We can then extend meet to two complex unit types: $\sqcap(b_1^{n_1} \cdots b_m^{n_m}, c_1^{k_1} \cdots c_m^{k_m})$ is defined as $\Pi_i \sqcap_d (b_i^{n_i}, c_i^{k_i})$ if none of the dimensional meets results in $\bot$, and $\bot$ otherwise. We can similarly extend downcasting from one complex type to another as $\downarrow(b_1^{n_1} \cdots b_m^{n_m}, c_1^{k_1} \cdots c_m^{k_m})$ as $\Pi_{i,n_i>0} \downarrow(b_i, c_i)^{n_i} \times \Pi_{i,n_i<0} \downarrow(c_i, b_i)^{-n_i}$ if no downcasts results in $\bot$ and $n_i = k_i, 1 \leq i \leq m$, or $\bot$ otherwise. This allows us to, for example, determine that $\sqcap(\mathtt{km}^2\mathtt{sec}^{-1}, \mathtt{m}^2\mathtt{hour}^{-1})$ is $\mathtt{m}^2\mathtt{hour}^{-1}$ and $\downarrow(\mathtt{km}^2\mathtt{sec}^{-1}, \mathtt{m}^2\mathtt{hour}^{-1})$ is $1000^2 \times (60 \times 60)^{--1} = 3600000000$.

We can multiply two complex unit types $u \otimes v$ together: $(b_1^{n_1} \cdots b_m^{n_m}) \otimes (c_1^{k_1} \cdots c_m^{k_m})$. This is defined as $x_1^{n_1+k_1} \cdots x_m^{n_m+k_m}$, where $x_i = b_i$ if $b_i = c_i \vee k_i = 0$, and $x_i = c_i$ if $n_i = 0$, and the result is $\bot$ if neither of these conditions hold for any dimension. Note that this *requires* that we do not multiply unit types with different basic types for the same dimension together.[3] We can invert a unit type $u \equiv b_1^{n_1} \cdots b_m^{n_m}$ to obtain $\mathbf{1}/u$ as $b_1^{-n_1} \cdots b_m^{-n_m}$. Division of unit types is just inversion followed by multiplication: $u/v = u \otimes (\mathbf{1}/v)$. For example, $\mathtt{km}^1\mathtt{sec}^{-1} \otimes \mathtt{km}^1$ gives unit type $\mathtt{km}^2\mathtt{sec}^{-1}$ and $\mathbf{1}/(\mathtt{km}^1\mathtt{sec}^{-1}) = \mathtt{km}^{-1}\mathtt{sec}^1$.

Note that while meet and down casting are a bit complex to define, actual models mainly use simple cases, as there are usually one or two basic unit types for each dimension.

3 Unit Type Checking

The main role of unit types is to prevent making unit type errors. They restrict the possible correct arithmetic expressions we can write. Below, we give our candidate syntax to work with unit types in existing MiniZinc models, and the type computation rules for important classes of arithmetic expression. Importantly, unit types are checked at the model level (without the data) and then erased, incurring no runtime overhead when solving the problem.

A constant c can be created of arbitrary unit type u using the `@` symbol, *c*`@`*u*. The operator should have a high precedence to ensure it binds tightly to prevent confusion about which expression it influences. A variable can be declared with a unit type by appending the unit type onto the type declaration using the `@` symbol as follows: "[`var`][`int`|`float`]`@`*u*`:`*id*"

[3] It is possible to introduce automatic coercion during multiplication, but there are quite a few non-obvious cases, so at present we restrict multiplication. The coercions could always be explicitly added to the model to allow a multiplication.

declares identifier id to have unit type u. We can similarly define set or array arithmetic parameters or decisions with unit types. During type inference/checking, the type of identifier id is its declared unit type u.

Here are the most important typing rules for this new syntax. The complete set of typing rules is given in Appendix B.

- $type(k@u) = u$. Constants simply have the annotated unit type.
- $type(e_1 + e_2) = \sqcap(type(e_1), type(e_2))$. Addition (and subtraction) require unit types which have a meet (i.e. they have the same dimensions and there is a coercion to a common unit).
- $type(e_1 \times e_2) = type(e_1) \otimes type(e_2)$. Multiplication of unit types is free in the dimension of the indices but restrictive in the basic unit types.
- $type(e_1 \texttt{ div } e_2) = type(e_1) \otimes type(\mathbf{1}/e_2)$. Division of unit types is simply inversion followed by multiplication.

Most other integer operations act similarly to $+$. For example, max, min and `if-then-else-endif` have type given by the meet of their (last 2) arguments.

Returning to the example from the introduction, we might add units types as follows

```
1 include "units.mzn";  % make declared units available
2 int@kg: limit;        % available weight limit
3 array[PRODUCT] of int@kg: wght;
4 array[PRODUCT] of int@dollar: profit;
```

The first line makes the standard library unit declarations available to the model. The parameters are now specified in terms of units. With these declarations, we immediately get the following unit type error on the second last line of the original model: `unit mismatch: expected "dollar", but got "kg"`.

To produce meaningful error messages for unit mismatches, the original units as written by the modeller (or propagated through multiplication or division) are always used in any user-facing messaging, in addition to the computed normalized units. This ensures that the modeller is not faced with error messages only mentioning constructs they never wrote, and the normalized units allow for easy comparison of the two units to see the exact mismatch. For example, for the following model an error will be produced for the last line stating: `unit mismatch: expected "vel = "m*s^-1" but got "vel/s = m*s^-2"`.

```
1 unit velocity: vel = m / s;
2 float@m: a = 5@m;
3 float@s: b = 2@s;
4 float@vel: c = a / b;
5 float@vel: d = c / b;
```

3.1 Coercion

In usual unit systems, coercion from one unit to a unit of the same dimension is generally allowed. The physical quantities being measured by dimensions are typically real, and are represented by a form of floating point number. This means that creating large or small values is not problematic. Even in the case that the dimension is arguably not a real number (e.g., worth might only be meaningful down to 1¢), it is still generally acceptable to use its smallest unit type (using large integers to represent it).

Unfortunately, this is not a suitable approach for (discrete) modelling languages such as MiniZinc. Most MiniZinc models contain integer decision variables, and using the smallest unit type to represent them can immensely expand the search space. This, in turn, would lead to large increases in solving time. Hence, we need to be conservative with automatic casting between units.

We propose a system that only allows automatic *downcasts* of units over the same dimension from a larger unit to a smaller unit, e.g. from `kg` to `gram`.

```
var int@kg: x;
var int@gram: y = x + 55@gram;
var int@kg: z = x + y;
```

which will in effect constrain $y = 1000 \times x + 55$, while the last line with create a unit type error, since $x + y$ has unit type `gram` and we cannot automatically upcast to fit into `kg`.

The user can instead *upcast* from a smaller unit to a larger unit when required using provided built-in upcast operations: `round`, `ceil` and `float`. We can correct the error above by rewriting it as

```
var int@kg: z = ceil(kg, x + y);
```

which will constrain $z = \lceil \frac{1000 \times x + y}{1000} \rceil$. The compiler can automatically create the underlying upcast function `ceil_kg_g` and replace the call in the original code by looking up the appropriate conversion factor:

```
var int@kg: ceil_kg_g(var int@gram: a) = ceil(a / 1000@g) *1@kg;
```

Automatic downcasts should be added every time a meet operation does not result in the same type as its arguments. So an expression $x + y$ where $type(x) = u$ and $type(y) = v$ and $\sqcap(u, v) = w$ will be replaced by $\downarrow(u, w) * x + \downarrow(v, w) * y$. For example, assuming $type(p) =$ `mile`, the expression `2@km + p@mile`, will be rewritten to $1000 \times 100 \times 2 + 160934 \times p$ with unit type `cm`. Note that if all constants in derived unit type declarations are integers, then automatic downcasts of an integer expression always remain integer.

Consider a model which schedules tasks to the minute, but tracks machine usage over each half day in the planning period (where tasks cannot span across multiple half days). A model that computes in which half days a machine is used from the start times of the tasks is given below.

```
unit time: halfday = 12@hour;
int@minute: makespan;
set of int@minute: TIME = 1@minute..makespan;
int@halfday: halfdays = ceil(halfday,makespan);
set of int@halfday: HALF = 1@halfday..halfdays;
array[TASK] of var TIME: start;
array[TASK] of var MACHINE: machine;
array[MACHINE,HALF] of var bool: used;
constraint forall(t in TASK)
                  (used[machine[t], ceil(halfday, start[t])]);
```

The translation from start times in minutes to the correct halfday is easily defined by the user using `ceil` upcasting. Note that we cannot know which rounding semantics is correct for this upcast automatically, so we must rely on the modeller.

Once unit type checking is complete, we add the multiplications required by an automatic downcasts that arose in the unit type checking, and add definitions for any upcast predicates used by the modeller. This is the only stage where code needs to be generated. After this, all unit types can be *erased* from the model allowing all following compiler phases to continue unaffected. For example, the model

```
var int@kg: x;
var int@gram: y = x + 55@gram;
var int@kg: z = ceil(kg,x + y);
```

will produce the following after unit coercion and erasure.

```
var int: x;
var int: y = x * 1000 + 55;
var int: z = ceil_kg_g(100*x + y);
function var int: ceil_kg_g(var int: a) = ceil(a / 1000);
```

3.2 Abstract units

Most MiniZinc models do not reason about a dimension using two different units, but we often reason about a dimension without specifying exactly what it means. For example, the knapsack problem of the introduction is equally valid whatever unit we describe mass in, as long as we use the same unit for both `limit` and `wght`. Similarly, the unit of worth could be cents or millions of dollars without changing how the model works. Hence, models can define *abstract* units. This does not require any extension to the language. Instead, we simply define new basic unit types in the model itself. For our knapsack example, we could introduce a new unit for mass `mmass` and worth `mworth` as follows.

```
include "units.mzn";  % make declared dimensions available
unit mass: mmass;
unit worth: mworth;
int@mmass: limit;     % available weight limit
array[PRODUCT] of int@mmass: wght;
array[PRODUCT] of int@mworth: profit;
```

Every dimension automatically creates a *default* abstract unit of the same name. Since in most models we only reason about one unit for a dimension, we can just use this unit. For example in the above example, we could remove lines 2 and 3, and replace `mmass` by `mass` and `mworth` by `worth` to use the default abstract units.

3.3 Units and Data in MiniZinc

Since data for discrete optimization models is usually created outside the model in existing unitless formats like CSV, JSON, or a database, it is important that we support unitless data. We assume that datafiles are unitless, and the unit information is attached to the definition of parameters. For our first example model, a sample (`.dzn`) data file might be:

```
k = 3;
PRODUCT = {cooktop, stove, oven, fridge, washer, dryer};
limit = 170;
wght = [16, 70, 32, 90, 70, 30];
profit = [400, 750, 500, 1200, 600, 500];
```

In the future, interfaces that use MiniZinc from other languages with unit types, such as MiniZinc Python [6], could provide information about the unit types of the data.

4 Counting types

One of the differences of discrete optimization models from normal programs is that they often reason about counts of different kinds of objects. We want to ensure that we do not make unit errors in our model when dealing with counting. Consider a variation of our running model where we use 0/1 variables to define the chosen products.

```
1 int: k;        % number of products to choose
2 int: limit;    % available weight limit
3 enum PRODUCT; % set of products available
4 array[PRODUCT] of int: wght;
5 array[PRODUCT] of int: profit;
6 array[PRODUCT] of var 0..1: chosen;
7 constraint sum(chosen) = k;
8 constraint sum(p in PRODUCT)(chosen[p]*wght[p]) <= limit;
9 solve maximize sum(p in PRODUCT)(chosen[p]*profit[p]);
```

What unit type do `k` and the `chosen` variables have? They are *counts* of `PRODUCT`. We extend any enumerated type in MiniZinc to also be used as the *unit type* for counting that enumerated type. In this model, the suggested unit type declarations are

```
1 int@PRODUCT: k;      % number of products to choose
2 int@mmass: limit;    % available weight limit
3 array[PRODUCT] of int@(mmass/PRODUCT): wght;
4 array[PRODUCT] of int@(mworth/PRODUCT): profit;
5 array[PRODUCT] of var (0..1)@PRODUCT: chosen;
```

The `wght` array records *mass per PRODUCT*, while the `profit` array records *worth per PRODUCT*. Since we are multiplying the counts of `PRODUCT`s `chosen[p]` by the product's weight `wght[p]`, it is clear that the constraint on line 8 is unit correct.

Note that the two declarations `PRODUCT: k` and `int@PRODUCT: k` are quite different, the first `k` holds a (non-numeric) product, while the second `k` holds an integer *count* of products.

A counting type is simply a new dimension together with a basic unit. So each enumerated type `E` adds a new basic unit also named `E` and a new dimension also named `E` to the unit type system, where $dim(\texttt{E}) = \texttt{E}$.

4.1 Fine counting types

We can actually still generate erroneous models even with the counting types we consider above. Consider the following model, which is checking that for any two chosen products their usage of any resource is below the pairwise usage limit for that resource:

```
1 enum RESOURCE;
2 array[PRODUCT, RESOURCE] of int@(RESOURCE/PRODUCT): usage;
3 array[RESOURCE] of int@RESOURCE: limit2;
4 array[PRODUCT] of var (0..infinity)@PRODUCT: chosen;
5 constraint forall(p1, p2 in PRODUCT where p1 < p2,
6                   r in RESOURCE)
7                  (usage[p1,r]*chosen[p1] +
8                   usage[p2,r]*chosen[p1] <= limit2[r]);
```

This is unit type correct, but unfortunately, there is a copy and paste bug. The problem here is that the counting types are too broad. The erroneous expression `usage[p2,r] * chosen[p1]` talks about different products. These types of errors are easy to make when we reason about two or more resource types in the same arithmetic expression.

To prevent these errors, we allow fine counting types for each member of an enum, as long as we have the enum as an index in the expression. We can rewrite the model declarations

```
enum RESOURCE;
array[p of PRODUCT, rr of RESOURCE] of int@(rr/p): usage;
array[rr of RESOURCE] of int@rr: limit2;
array[p of PRODUCT] of var (0..1)@p: chosen;
```

The notation v `of` E gives us access to the value v of the enumerated type E in the rest of the declaration.[4] In this declaration, we are explicit that each value in the `usage` array is specific to the PRODUCT and RESOURCE combination. It defines the resource usage per product. We similarly define the resource limits and the chosen resources. With these unit type definitions we can detect that the unit type of `usage[p1,r]*chosen[p1]` is `r`, but the unit type of `usage[p2,r]*chosen[p1]` is `r/p2*p1`, and therefore, the last constraint is erroneous. The right-hand side of the `+` operator is giving a unit mismatch error: `expected "r" but got "r*p2^-1*p1"`.

In the formalization, each fine-grained unit type used in the model, e.g., `p of PRODUCT`, defines a new basic unit type $PRODUCT(e)$ of dimension $PRODUCT$ for each expression e appearing in an array lookup at that position. For the example above, we have new unit types $PRODUCT(p1)$, $PRODUCT(p2)$ and $RESOURCE(r)$. The type of `usage[p2,r]` is obtained by replacing the p by $p2$ and r by rr giving $PRODUCT(p2)/RESOURCE(r)$, similarly the type of `chosen[p1]` is $PRODUCT(p1)$, and the result of the multiplication is invalid (two units for the same dimension). The printed error message omits the dimension annotation around the units.

There is no coercion from fine-grained unit type `p` to the coarse grained unit type `PRODUCT`, since allowing it would remove the unit type error in the example above, by coercing both unit type `p1` and unit type `p2` to `PRODUCT`.

5 Coordinate types

Consider the following MiniZinc model for a simple scheduling problem with precedences between pairs of tasks:

```
enum TASK;                              % set of tasks
array[TASK] of int@minute: d;           % duration of task (mins)
enum PREC;                              % set of precedences
array[PREC, 1..2] of TASK: pt;          % set of task pairs
array[TASK] of var int@minute: s;       % start time decisions
constraint forall(p in PREC)
                  (s[pt[p,1]] + d[pt[p,1]] <= s[pt[p,2]]);
constraint disjunctive(d,s);
```

[4] Similar notation has been used for other purposes in the modelling language OPL.

Each task has a duration `d`, and we wish to decide its start time `s` (all in minutes). The constraint in lines 6-7 ensures that the first task in each row of `pt` must be completed before the second task. Finally, no two tasks should overlap. The model is unit type correct, and runs and can give answers, but it is wrong!

In MiniZinc and almost all languages, we consider the numeric types to be essentially *differences*. That is, they are a value that can be added to each other, or subtracted or multiplied. But there is a different use of numeric types as *absolute coordinates* in some numerical system. It does not make sense to add these coordinates usually: consider 11°C and 25°C, while it certainly makes sense to determine their difference $25°\text{C} - 11°\text{C} = 8°\text{C}$, how are we to interpret $11°\text{C} + 25°\text{C}$? Adding two temperatures does not make any sense.

This arises in models where we are reasoning both about absolute coordinates in some unit and the differences. Coordinate unit types and difference unit types are supported in some other unit type systems [9].

Suppose we are computing the end position we reach, given a fixed duration of running at a fixed velocity, given an unknown start position. Then the start and end position are both coordinates (in some coordinate space), while the length, velocity, and duration are non-coordinate (difference) values. We use the notation `coord(`u`)` to define a coordinate unit type of basic unit type u. The model that separates these different types is written:

```
1 var int@coord(m): start;
2 var int@coord(m): end;
3 var int@m: len = end - start;
4 var int@s: duration;
5 int@(m/s): velocity;
6 constraint len = duration * velocity;
```

Coordinate types are much more restricted than ordinary unit types, they support the following typed arithmetic operations only:

- $\texttt{coord}(x) + x = \texttt{coord}(x)$
- $\texttt{coord}(x) - x = \texttt{coord}(x)$
- $\texttt{coord}(x) - \texttt{coord}(x) = x$

So we can only add or subtract a (difference) unit value to a coordinate unit value to get another coordinate, or subtract one coordinate from another to get a (difference) unit value. Importantly, we cannot add two coordinate type values together, even if they have the same unit type. Similarly, we cannot multiply or divide a coordinate type, since this is equivalent to a repeated addition or its inverse.

Coordinate types appear frequently in scheduling and packing problems within the combinatorial problem class. We can restrict our definitions of scheduling and packing global constraints to enforce the correct use of coordinate types. For example, for `disjunctive` the start times are coordinates while the durations are usual (difference) unit types. With the unit typed definition (shown in the next section) we discover the error in the model at the beginning of the section. We have reversed the arguments to disjunctive. They both have the same units, so there is no unit error, but one is a coordinate type, the other not. The corrected model rewrites these two lines:

```
1 array[TASK] of var int@coord(minute): s;
2 constraint disjunctive(s, d);
```

If we only rewrite just one of the two lines, we still get a type error.

Since coordinate types only seem to make sense for a single dimension, we restrict $coord(u)$ to only be defined for declared or derived basic unit types u. Again, this restriction is made to get stricter unit types, and hence discover more errors.

6 Global constraints and Units

One of the strengths of constraint programming modelling languages including MiniZinc is the wide range of global constraints that they support. In order for units to be truly useful, we need to extend these global constraint definitions to use units accurately. To do so, we need to be able to define functions which are parametric in their unit types. We introduce (unit) type variables `$v` to appear as unit expressions. In MiniZinc the notation `$$V` indicates an arbitrary enumerated type.

6.1 Function parameters with units

There are many arithmetic constraints to which we can attach units to the arguments of interest. For example, `sliding_sum` constrains the sum of each consecutive sequence of `seq` elements in the array `vs` to be in the range `low..up`. The type of this global constraint including units is simply:

```
predicate sliding_sum(int@$u: low,
                      int@$u: up,
                      int@$$E: seq,
                      array [$$E] of var int@$u: vs)
```

Notice how `seq` is a counting type for the index set of `vs`.

Similarly, the knapsack constraint deals with at least two units: measuring worth (for the objective) and weight for the constraint. The unit typed declaration is:

```
predicate knapsack(array [$$I] of int@($W/$$I): w,
                   array [$$I] of int@($P/$$I): p,
                   array [$$I] of var int@$$I: x,
                   var int@$W: W,
                   var int@$P: P)
```

where we are explicit about the two units used for the constraint and objective. We can also define a version using fine counting types:

```
predicate knapsack(array [i of $$I] of int@($W/i): w,
                   array [i of $$I] of int@($P/i): p,
                   array [i of $$I] of var int@i: x,
                   var int@$W: W,
                   var int@$P: P)
```

Note we can use fine unit types when the index type is an enumerated type.

The counting global constraints have explicit unit types using the counting units. The `among` constraint counts how many occurrences of the elements in a subset appear in an array. Obviously if the array contains elements of an enumerated type, then it returns a count unit for that type. The parametric type of `among` is defined as

```
function var int@$$E: among(array [$X] of var $$E: x,
                            set of $$E: v)
```

We can similarly treat `global_cardinality` constraints. But the ability to use fine counting types also allows us to define a new version of `global_cardinality` when we count the elements of an enumerated type:

```
function array [t of $$T] of var int@t:
         global_cardinality(array [$X] of var $$T: x)
```

Finally, packing and scheduling constraints make use of coordinate unit types, and polymorphic unit type variables. The `disjunctive` constraint is defined as:

```
predicate disjunctive(
    array[$$T] of var int@coord($t): start,
    array[$$T] of var int@$t: duration);
```

For example, for `cumulative`, the start time is a coordinate time unit type, while duration is the usual (difference) version of the same time unit. The resource usage for each task and capacity uses the same unit for resource. This allows us to give, for example, the capacity in metric tons and the resource usage in kilograms, with coercion done at the call interface.

```
predicate cumulative(
    array[$$T] of var int@coord($t): start,
    array[$$T] of int@$t: duration,
    array[$$T] of int@$u: usage, int@$u: capacity);
```

Similarly, the `span` constraint, which enforces that the optional task defined by start `s0` and duration `d0` *spans*, i.e., goes from the earliest start time to the latest end time, of a set of optional tasks defined by start times `s` and durations `d`, combines coordinate and difference types with optionality:

```
predicate span(var opt int@coord($t): s0,
               var int@$t: d0,
               array [$$E] of var opt int@coord($t): s,
               array [$$E] of var int@$t: d)
```

7 Evaluation

We have developed a prototype implementation of the unit type extension to MiniZinc.[5] The implementation performs the type checking for the unit types, adds in the required coercions, and then erases the unit types from the model. Afterwards, type checking and compilation of MiniZinc continues as normal.

7.1 Analysis: MiniZinc Challenge

To evaluate the extent of the usefulness of unit types, we examine the MiniZinc Challenge problems from the last four years, 2021–2024, to see how they would be written with unit types. In our analysis, we exclude procedurally generated models and the `unison.mzn` model because their immense size and lack of documentation prevent us from rewriting these models. Models used in multiple years have only been included in the counts for their earliest occurrence. The increase in model character count was calculated by counting the number of non-whitespace, non-comment characters that were added to each model where unit types were applicable (reported as a percentage increase). Table 1 gives a summary of the results of the evaluation, and the models are available at `https://www.minizinc.org/unit-types/`.

In our analysis, we found that the majority of models can benefit from unit types (*Units applicable* column), e.g. 13 out of 18 models in 2021 could be rewritten to use units, and 56 out of 71 overall. Notably, the use of unitless numeric types is nearly eliminated when models

[5] The prototype is available at `https://www.minizinc.org/unit-types/`.

Table 1 A table showing the mean increase in model size for using unit types and usage of the different unit type features: **Chars** increase in non-whitespace, non-comment characters; **Bytes** increase in raw file size; **Count** counting types; **Fine** Fine counting types; **Coord** coordinate types; **Global** unit-specialized globals.

		Mean size increase		Benchmarks using the unit type feature			
Year	Units applicable	Chars	Bytes	Count	Fine	Coord	Global
2021	13/18	9.2%	3.2%	6	3	5	4
2022	19/20	8.0%	3.6%	8	3	3	9
2023	13/18	4.7%	1.4%	4	4	10	3
2024	11/15	6.9%	5.5%	1	0	5	3
Overall	56/71	7.1%	3.3%	19	10	23	19

are refactored to employ enumerated and unit types. Remaining integer expressions without unit types generally involve: counting of Boolean conditions, factors to convert between different units, or reasoning about the numbers themselves (e.g. in pure combinatorial problems). For example, the `mznc2017_aes_opt.mzn` model reasons about a cryptographic algorithm, where integers are used from the perspective of numerics. Using unit types in a model only results in a small increase in code size in most cases. In fact, they can even lead to smaller models since having the units alleviates the need to include these in variable names or comments as is often done.

We observed that use of the already existing `any` keyword for inferring variable type and unit significantly reduces repetitive specification of unit types when storing the results of expressions. Although the removal of the specific types might remove some "self-documentation" of variable declarations, modern code editors can often display the type that the compiler determined, and this could be extended to also include the inferred unit.

Furthermore, many models where counting types can be used can utilize fine-grained counting types. However, the use of fine-grained counting types over general counting types does not always translate into improved type safety. Most expressions only access a single counting element at a time. Due to the nature of the unit types implementation, it is easy for modellers to choose what counting type suits their implementation best.

The most extensive changes in models arise from the introduction of coordinate unit types. In these models, normal units and their corresponding difference types are stored together. This suggests that the introduction of coordinate types might offer an opportunity to detect many subtle problems in optimization models which reason about both coordinates and differences.

The benchmarks used in the MiniZinc Challenge are mostly designed for benchmarking purposes and also do not include any floating-point variables, so we did not see extensive use of the concrete units defined in the units type library, which are most useful when dealing with real-world physical quantities. In most cases, models were able to use the predefined dimensions from the units type library, and use their own or default abstract units.

7.2 Case Study: Equipment Placement

To illustrate the use of complex unit types, we show a MiniZinc model for placing equipment in a two-dimensional array, with restrictions on how close they can be placed in terms of Manhattan distance. This example is designed to show one of the most **intrusive** uses of unit types because we *choose* to use separate units for width and height as well as Manhattan distances. We can make a much simpler model if we only use one distance unit, but we can then mix up the usages of the three types of distance.

The three units are defined below using coordinate types for the x, y coordinates:

```
unit distance: dist;
unit distance: width;
unit distance: height;
int@width: W; % width of area
set of int@coord(width): COL = 1@width..W;
int@height: H; % height of area
set of int@coord(height): ROW = 1@height..H;
```

There are different kinds of equipment that can be placed, with data about it, the cost, availability, and effectiveness at various distances, defined using fine counting types:

```
enum EQUIP;                        % different equipment available
int@dist: maxradius;               % max protection radius
set of int@dist: DIST = 0@dist..maxradius;
unit worth: mworth;                % unit of cost
unit type measure;                 % new dimension "measure"
unit measure: effect;              % unit of measure "effectiveness"
array[e of EQUIP] of int@(mworth/e): cost;      % cost
array[e of EQUIP] of int@e: avail;              % availability
array[e of EQUIP, DIST] of int@(effect/e): eff; % effect at dist
```

We are given a limit on the total number of equipment placed. The key decisions are where to place equipment up to some given limit on number, defined as an x and y position and the type of equipment placed t.

```
int@mworth: budget;                % budget on equipment
int@EQUIP: limit;                  % max number of equipment;
set of int@EQUIP: LEN = 1@EQUIP..limit;
array[LEN] of var opt COL: x;      % x position of unit (or absent)
array[LEN] of var opt ROW: y;      % y position of unit (or absent)
array[LEN] of var opt EQUIP: t;    % which equipment at this pos
var 0@EQUIP..limit: used;          % number of equipment used
```

Option types allow us not to use all of `LEN` to hold actual equipment, only the positions up to `used` are actually used. Entries in these arrays at positions greater than `used` are forced to be absent:

```
constraint forall(i in LEN)(i > used <->
           absent(x[i]) /\ absent(y[i]) /\ absent(t[i]));
```

The most complex part of the unit types is computing Manhattan distances defined as:

```
function var int@dist: man(var opt int@coord(width): x1,
                           var opt int@coord(height): y1,
                           var opt int@coord(width): x2,
                           var opt int@coord(height): y2) =
    abs(x1 - x2) * 1@(dist/width) +
    abs(y1 - y2) * 1@(dist/height);
```

where we strip the width and height units and add back `dist`.[6] Note that the subtraction of one coordinate type from another to compute a non-coordinate typed result. The most complicated constraint adds up the effectiveness of each unit at each location, defined as

```
array[ROW,COL] of var (0..100)@effect: cover;
constraint forall(r in ROW, c in COL)
    (cover[r,c] = min(100@effect, sum(i in 1@EQUIP..used)
        (let { var int@dist: dst = man(x[i],y[i],r,c); } in
        if dst > maxradius then 0@effect
        else eff[t[i],dst] endif)));
```

Here for every row and column r,c we sum up the effectiveness of each placed equipment that covers that location by computing the distance to each placed equipment and, if within the max radius, adding its effectiveness, capped at 100.

Availability constraints are enforced by a unit typed global cardinality (lower and upper bounds are the 3rd and 4th arguments):

```
constraint global_cardinality(t, EQUIP, [0@e|e in EQUIP], avail);
```

Most constraints are only minutely affected by units, if at all. For example, the budget constraint, and the constraint forcing equipment to be no closer than 3 units of distance are written as follows:

```
constraint sum(i in 1@EQUIP..used)(cost[t[i]]) <= budget;
constraint forall(i, j in 1@EQUIP..used where i < j)
                  (man(x[i], y[i], x[j], y[j]) >= 3@dist);
```

The three separate units used for distance make this one of the more complicated models to add strong unit types too. The payoff is that we can no longer mix them up. The only real complexity arises from computing Manhattan distances, which encapsulated the type coercion of the distance types. Note that, finally, every integer in the model has a unit type (or is an enumerated type).

8 Related Work

The concept of unit types in programming languages has a long history. For many years, it has been clear that programming languages can help avoid crucial problems by incorporating unit types [10]. The first implementation of these features was in the Pascal and Ada languages [13, 7, 3]. Since then, extensions have been proposed for many types of programming languages, including functional languages [11, 12] and object-oriented languages [1].

Research in database systems has also explored the use of unit types. Works like [8] propose extending database schemas to include unit types. In particular, when dealing with international data, where the same data might be recorded with different units, the use of unit types enables robust data storage, retrieval, and manipulation across datasets.

These days, the incorporation of units directly within programming languages and database systems seems to have dwindled. However, this can be directly attributed to the "meta" capability of modern systems, allowing the functionality to be implemented as external libraries. The widespread adoption of libraries like Boost.Units [16], mp-units [15],

[6] The function takes *optional* integers since the x and y positions are optional, although it will never be called with absent values.

astropy.units [2], Pint [9], postgresql-unit [4], and many others [14] in various programming environments underscores the continued importance of unit type safety. These libraries provide extensive unit catalogues and enforce unit consistency during calculations, reducing the risk of errors in modern day scientific and engineering applications.

In the field of constraint modelling, the Pyomo constraint modelling library [5] is the only modelling tool that we are aware of that enables the use of unit types. Based on the Pint Python package, Pyomo allows users to specify units of decision variables and provides helper functions to check what the units of expressions are and to check whether they are consistent. Because Python and Pyomo do not use a strong typing system, the user is charged with ensuring the consistency of the model is checked.

In comparison to Pyomo, once unit types are added to a MiniZinc model, the checking of these types is always enforced by the compiler. In addition, the static type checking and type erasure in MiniZinc mean that using unit types incurs no significant time overhead during the translation. The additional unit type variants, counting types and coordinate types, are not available in Pyomo. Similar to Pyomo, MiniZinc's interfaces to other languages, such as MiniZinc Python, might in the future be extended to connect to existing unit type libraries, such as Pint. By extending these interfaces to handle unit information from other library, developers can verify that unit-aware data is consistent with their unit-aware MiniZinc model.

9 Conclusion

Unit types provide stronger type safety than is usual for even strongly typed languages. While unit type extensions have been defined for almost every programming language, they are still rarely used, even though they can detect rather subtle bugs, which can have catastrophic effect (as exemplified by the Mars Climate Orbiter debacle). The problem with unit types for developers is that the overhead of using them may not appear to pay off. The trade-off for modelling languages is much more attractive, since debugging subtle errors in a model is much more challenging than in procedural code. Interestingly, there are kinds of unit types that only arise in discrete optimization, so-called counting types. In this paper, we define a lightweight but fairly complete system of unit types suitable for discrete optimization modelling languages. We have fully implemented this as an extension of the MiniZinc modelling language. In practice, we see a fairly small overhead for using them, with potentially large gains in avoiding subtle modelling errors. Unit types will be available in MiniZinc in a near future release.

References

1 Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele. Object-oriented units of measurement. *SIGPLAN Not.*, 39(10):384–403, October 2004. doi:10.1145/1035292.1029008.

2 Astropy Collaboration. The Astropy Project: Sustaining and Growing a Community-oriented Open-source Project and the Latest Major Release (v5.0) of the Core Package. *The Astrophysical Journal*, 935(2), 2022.

3 Geoff Baldwin. Implementation of physical units. *SIGPLAN Not.*, 22(8):45–50, August 1987. doi:10.1145/35596.35601.

4 Christoph Berg. postgresql-unit, January 2023. URL: https://github.com/mpusz/units.

5 Michael L Bynum, Gabriel A Hackebeil, William E Hart, Carl D Laird, Bethany L Nicholson, John D Siirola, Jean-Paul Watson, and David L Woodruff. *Pyomo - optimization modeling in Python, 3rd Edition*, volume 67. Springer, 2021. doi:10.1007/978-3-030-68928-5.

6 Jip J. Dekker. MiniZinc Python, April 2023. URL: `https://github.com/MiniZinc/minizinc-python`.

7 A Dreiheller, B Mohr, and M Moerschbacher. Programming Pascal with physical units. *SIGPLAN Not.*, 21(12):114–123, December 1986. `doi:10.1145/15042.15048`.

8 N.H. Gehani. Databases and units of measure. *IEEE Transactions on Software Engineering*, SE-8(6):605–611, 1982. `doi:10.1109/TSE.1982.236021`.

9 Hernan E. Grecco. Pint, December 2023. URL: `https://github.com/mpusz/units`.

10 Michael Karr and David B. Loveman. Incorporation of units into programming languages. *Commun. ACM*, 21(5):385–391, May 1978. `doi:10.1145/359488.359501`.

11 Andrew Kennedy. Dimension types. In Donald Sannella, editor, *Programming Languages and Systems - ESOP'94, 5th European Symposium on Programming, Edinburgh, UK, April 11-13, 1994, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 1994. `doi:10.1007/3-540-57880-3_23`.

12 Andrew John Kennedy. Programming languages and dimensions. Technical Report UCAM-CL-TR-391, University of Cambridge, Computer Laboratory, April 1996. `doi:10.48456/tr-391`.

13 R. Männer. Strong typing and physical units. *SIGPLAN Not.*, 21(3):11–20, March 1986. `doi:10.1145/382280.382281`.

14 Steve McKeever, Oscar Bennich-Björkman, and Omar-Alfred Salah. Unit of measurement libraries, their popularity and suitability. *Softw. Pract. Exp.*, 51(4):711–734, 2021. `doi:10.1002/SPE.2926`.

15 Mateusz Pusz, Johel Ernesto Guerrero Peña, Chip Hogg, and The mp-units project team. mp-units, May 2021. URL: `https://github.com/mpusz/units`.

16 Boris Schäling. *The Boost C++ libraries*. Boris Schäling, 2011.

A MiniZinc Unit Types Syntax

Unit types add syntax to MiniZinc to declare dimensions, dimension expression, basic unit types, and unit type expressions. The extensions are defined below.

Basic Dimension Declaration	d	::=	`unit type` id `;`	
Dimension Expression	de	::=	id	basic
		|	`coord(` id `)`	coordinate dimension
		|	de `*` de	multiplication
		|	de `/` de	division
		|	`1 /` de	inversion
		|	de `^` k	power (k is integer)
Dimension Declaration	cd	::=	`unit type` id `=` de`;`	
Basic Unit Type	b	::=	`unit` id `:` id `;`	declared
		|	`unit` id `:` id `=` k `@` id `;`	derived (k is arithmetic constant)
Unit Type Expression	ue	::=	id	basic
		|	`coord(` id `)`	coordinate type
		|	ue `*` ue	multiplication
		|	ue `/` ue	division
		|	`1 /` ue	inversion
		|	ue `^` k	power (k is integer)

The extension to the existing MiniZinc syntax is adding unit type annotations using `@`, together with the syntax for fine-grained unit iterators. Note that the `@` binds tighter than all arithmetic operators.

Expression	e	::=	k `@` ue	adding unit type (k is arithmetic)
Par/Var Declaration	dec	::=	[`var`] t [`@` ue] `:` id `;`	unit typed var/par
Index Set	is	::=	[id `of`] $enum$	fine-grained unit id

B MiniZinc Unit Typing Formalization

We now elaborate on the formal definitions in the paper and provide a full formalization of the unit typing in MiniZinc. We begin by formalizing unit types, and bottom up the type inference/checking procedure. We then show how we can add automatic coercions to programs that use multiple units for the same dimension.

B.1 Typing Rules

We use $\bot$ to represent a type error. A model contains m different dimensions $di_1, \ldots, di_m$ defined by the dimensions declared (or imported) into the model, and a dimension for each enumerated type defined in the model. We assume a set B of defined and declared basic unit types, together with each enumerated type defined in the model, and each fine grained unit expression. For each $b \in B$, $dim(b) = di_i$ for some i. We pick a *default* basic unit for each dimension di_i denoted $defb_i$ simply for notational convenience.

A *compound unit type* u is of the form $b_1^{n_1} b_2^{n_2} \cdots b_m^{n_m}$ where each $n_i, 1 \leq i \leq m$ is an integer, and $dim(b_i) = di_i, 1 \leq i \leq m$. The dimension of u is $di_1^{n_1} di_2^{n_2} \cdots di_m^{n_m}$.

A *coordinate unit type* $coord(b)$ is defined only for declared or derived unit types $b \in B$. We let $dim(coord(b)) = coord(dim(b))$. We extend the meet operation to coordinate types in the obvious way $\sqcap(coord(b_1), u) = coord(\sqcap(b_1, b_2))$ iff $u = coord(b_2)$ and $\sqcap(b_1, b_2) \neq \bot$ and $\bot$ otherwise. We extend downcasting to coordinate types also, so $\downarrow(coord(b_1), u) = coord(\downarrow(b_1, b_2))$ if $u = coord(b_2)$ and $\bot$ otherwise. We extend multiplication of unit types to coordinate types as follows: $u \otimes coord(b)$ is $coord(b)$ iff $u = \mathbf{1}$ and $\bot$ otherwise. Note these extension definitions implicitly rely on the fact that each operations is commutative.

A *unit type* is either a proper unit type or a coordinate unit type.

Type expression evaluation is defined as follows.

$$
\begin{array}{rcl}
\tau(\mathbf{1}) & = & defb_1^0 defb_2^0 \cdots defb_m^0 \\
\tau(b) & = & b_1^{n_1} b_2^{n_2} \cdots b_m^{n_m} \text{ where if } dim(b) = di_i,\ b_i = b, n_i = 1. \\
 & & \text{and } b_j = defb_j, n_j = 0, 1 \leq j \neq i \leq m \\
\tau(\texttt{coord}(u)) & = & \begin{cases} coord(\tau(u)), & \tau(u) = b_i, b_i \in B \\ \bot, & \text{otherwise} \end{cases} \\
\tau(u \times u') & = & \tau(u) \otimes \tau(u') \\
\tau(\mathbf{1}/u) & = & \begin{cases} b_1^{-n_1} b_2^{-n_2} \cdots b_m^{-n_m} & \tau(u) = b_1^{n_1} b_2^{n_2} \cdots b_m^{n_m} \\ \bot & \text{otherwise} \end{cases} \\
\tau(u/u') & = & \tau(u) \otimes \tau(\mathbf{1}/u') \\
\tau(u\hat{}c) & = & \begin{cases} b_1^{n_1 \times c} b_2^{n_2 \times c} \cdots b_m^{n_m \times c}, & \tau(u) = b_1^{n_1} b_2^{n_2} \cdots b_m^{n_m}, c \text{ integer constant} \\ \bot, & \text{otherwise} \end{cases} \\
\tau(u) & = & \tau(e),\ u \text{ is a compound unit type defined by } u = e
\end{array}
$$

The unit typing rules for arithmetic expressions are defined below. Note that if in typing expression e some subexpression results in $\bot$ the whole expression is ill-typed, and we emit an error message.

$$
\begin{aligned}
type(k) &= \tau(\mathbf{1}),\ k \text{ is a numeric constant}\\
type(e@u) &= \begin{cases} \tau(u), & type(e) = \tau(\mathbf{1}) \\ \bot, & \text{otherwise} \end{cases}\\
type(x) &= \tau(u),\ \text{identifier } x \text{ has declared unit type } u\\
type(\texttt{bool2int}(e)) &= \tau(\mathbf{1})\\
type(e_1 + e_2) &= \begin{cases} type(e_1), & type(e_1) = coord(type(e_2)) \\ \sqcap(type(e_1), type(e_2)), & \text{otherwise} \end{cases}\\
type(e_1 - e_2) &= \begin{cases} u, & type(e_1) = type(e_2) = coord(u) \\ type(e_1), & type(e_1) = coord(type(e_2)) \\ \sqcap(type(e_1), type(e_2)), & \text{otherwise} \end{cases}\\
type(e_1 \times e_2) &= \tau(type(e_1)) \otimes type(e_2))\\
type(e_1 / e_2) &= \tau(type(e_1)/type(e_2))\\
type(e_1 \ \texttt{div}\ e_2) &= \tau(type(e_1)/type(e_2))\\
type(e_1 \ \texttt{mod}\ e_2) &= \begin{cases} type(e_1) & type(e_2) = \tau(\mathbf{1}) \\ \bot, & \text{otherwise} \end{cases}\\
type(\texttt{abs}(e)) &= type(e)\\
type(\texttt{max}(e_1, e_2)) &= \sqcap(type(e_1), type(e_2))\\
type(\texttt{min}(e_1, e_2)) &= \sqcap(type(e_1), type(e_2))\\
type(e_1 \ \texttt{\^{}}\ e_2) &= \begin{cases} \tau(type(e_1)\texttt{\^{}}e_2), & e_2 \text{ is a integer constant} \\ \tau(\mathbf{1}), & type(e_1) = type(e_2) = \tau(\mathbf{1}) \\ \bot, & \text{otherwise} \end{cases}\\
type(trig(e)) &= \begin{cases} \tau(1), & type(e) = \tau(1) \text{ or } type(e) = \tau(b) \text{ where } dim(b) = \texttt{angle} \\ \bot, & \text{otherwise} \end{cases}\\
&\quad \text{where } trig \text{ is a trigonometric function } \{\texttt{sin}, \texttt{cos}, \dots\}.\\
type(upcast(u, e)) &= \begin{cases} type(u), & \sqcap(type(u), type(e)) = type(e) \\ \bot & \text{otherwise} \end{cases}\\
&\quad upcast \text{ is a upcast function } \texttt{ceil}, \texttt{floor}, \texttt{round}.\\
type\begin{pmatrix} \texttt{if } e_1 \\ \texttt{then } e_2 \\ \texttt{else } e_3 \texttt{ endif} \end{pmatrix} &= \begin{cases} \sqcap(type(e_2), type(e_3)), & type(e_1) = \texttt{bool} \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

Container types in MiniZinc can contain objects with unit types. We extended the unit types to container types in the obvious way. We can define $\texttt{array}[is]\ \texttt{of}\ u$ an array with index set is of objects with unit type u;[7] $\texttt{set of}\ u$ a set of objects with unit type u; similarly for records and tuples. We extend the meet operation to apply to a sequence of types in the obvious manner $\sqcap([u_1, \dots, u_n]) = \sqcap(u_1, \sqcap(u_2, \dots, \sqcap(u_{n-1}, u_n) \cdots))$.

Unit typing rules for (some) containers are given below:

$$
\begin{aligned}
type([e_1, \dots, e_n]) &= \begin{cases} \texttt{array[1..n] of } t & t = \sqcap([type(e_1), \cdots, type(e_n)], t \neq \bot \\ \bot, & \text{otherwise} \end{cases}\\
type(\{e_1, \dots, e_n\}) &= \begin{cases} \texttt{set of } t & t = \sqcap([type(e_1), \cdots, type(e_n)], t \neq \bot \\ \bot, & \text{otherwise} \end{cases}\\
type(e_1 \in e_2) &= \begin{cases} \texttt{bool} & type(e_1) = u, type(e_2) = \texttt{set of } u', \sqcap(u, u') \neq \bot \\ \bot, & \text{otherwise} \end{cases}\\
type(e_1[e_2]) &= \begin{cases} type(u) & type(e_1) = \texttt{array}[int]\ \texttt{of}\ u \\ \bot, & \text{otherwise} \end{cases}
\end{aligned}
$$

Unit typing rules for arithmetic constraints are defined below.

$$
\begin{aligned}
type(e_1\ op\ e_2) &= \begin{cases} \texttt{bool}, & \sqcap(type(e_1), type(e_2)) \neq \bot \\ \bot, & \text{otherwise} \end{cases}\\
&\quad op \text{ is a mathematical relation } \texttt{=}, \texttt{>}, \texttt{<}, \texttt{>=}, \texttt{<=}, \texttt{!=}.
\end{aligned}
$$

Unit typing rules for assignment are defined as:

$$
type(x = e) = \begin{cases} \texttt{bool} & \sqcap(type(x), type(e)) = type(x) \\ \bot & \text{otherwise} \end{cases}
$$

which requires the type of x to be a smaller unit type than that of e.

[7] Just showing the case for one dimensional arrays for simplicity.

Unit typing rules for predicates and user-defined functions are given below. Here, because the declared types can be parametric in unit (and other) types, we have to find a consistent mapping θ from type variables to fixed types that agrees with the types (including units) of the arguments.

$$type(p(e_1,\ldots,e_n)) = \begin{cases} \texttt{bool}, & type(e_i) = \tau(\theta(u_i)), 1 \le i \le n \text{ for type substitution } \theta \\ \bot, & \text{otherwise} \end{cases}$$
where f is a predicate with unit type $u_1 \times \cdots \times u_n \to \texttt{bool}$

$$type(f(e_1,\ldots,e_n)) = \begin{cases} type(\theta(u)), & type(e_i) = \tau(\theta(u_i)), 1 \le i \le n \text{ for type substitution } \theta \\ \bot, & \text{otherwise} \end{cases}$$
where f is a user defined function with unit type $u_1 \times \cdots \times u_n \to u$

A *fine-grained unit type* is written $\texttt{array[}i\ \texttt{of}\ enum\texttt{]}\ \texttt{of}\ u(i)$ where $enum$ is an enumerated type, and $u(i)$ is a unit type expression making use of the identifier i.[8] To allow fine-grained unit types, we have to extend the type system to include unit type identifiers i which are replaced by the appropriate expression during type checking. There are treated as if they were a new basic type (so B includes all the unit type identifiers used in the model). Fine-grained types are introduced via variable declarations:

$$\texttt{array[}i\ \texttt{of}\ enum\texttt{]}\ \texttt{of}\ u(i) : x$$

declares a variable x to have (fine-grained) unit type $\texttt{array[}i\ \texttt{of}\ enum\texttt{]}\ \texttt{of}\ u(i)$. We extend the typing rules for array lookups to:

$$\tau(u) = u,\ u \text{ is a unit type variable.}$$
$$type(e_1[e_2]) = \begin{cases} type(u) & type(e_1) = \texttt{array}[int]\ \texttt{of}\ u \\ type(u(e_2)) & type(e_1) = \texttt{array}[i\ \texttt{of}\ enum]\ \texttt{of}\ u(i), e_2 \text{ is an identifier} \\ \bot, & \text{otherwise} \end{cases}$$

Fine-grained unit type checking does not otherwise change the typing rules. Type equivalence relies on the unit type identifiers appearing identically in the two types, which means that all the identifiers must appear identically in each term.

[8] We restrict ourselves to one dimensional fine-grained types for simplicity, the extension to more dimensions is reasonably straightforward.

Cargo Routing Optimization in Liner Shipping Networks

Yousra El Ghazi
Aix Marseille Univ, CNRS, LIS, Marseille, France

Djamal Habet
Aix Marseille Univ, CNRS, LIS, Marseille, France

Cyril Terrioux
Aix Marseille Univ, CNRS, LIS, Marseille, France

Abstract

Given a maritime liner network, the cargo routing problem consists of determining which maritime routes containers will take to be transported from their loading port to their destination port. This constrained optimization problem is an essential task in the context of maritime commodity transportation, both in terms of network design and operational implementation. Several variants of this problem have been considered depending on their usage (e.g. for designing the network or for carrying containers in practice), the constraints taken into account or the criterion to be optimized.

In this article, we address several variants of this problem, relying on the flexibility of Constraint Programming. First, we propose two general COP models. Then, we describe a local search method that can be easily adapted to the desired context. Finally, we experimentally compare our two models and the proposed method.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence

Keywords and phrases Constraint optimization problem, modeling, solving, industrial application

Digital Object Identifier 10.4230/LIPIcs.CP.2025.11

Supplementary Material
Dataset: `https://pageperso.lis-lab.fr/cyril.terrioux/Cargo_Routing/instances.zip`

Funding This work has been funded by Bpifrance under the PIA project Transformation Numérique du Transport Maritime (TNTM).

1 Introduction

Containerized shipping plays a crucial role in global trade, as evidenced by the report of the United Nations Conference on Trade and Development (UNCTAD). More than 80% of the volume of international trade is currently transported by sea, with 1.95 billion tons transported in containers. There are two main container sizes: 20-foot and 40-foot containers. So, the vessels' storage space is divided into 40-foot units, allowing for the stacking of either 40-foot or 20-foot containers. The Twenty-foot Equivalent Unit (TEU) is the unit generally used to count containers, where a 40-foot container is equivalent to 2 TEUs. Moreover, containers can be divided in three main categories. *Dry* containers are the most commonly used and are suitable for transporting dry commodities (e.g. textiles or electronics). *Reefers* are designed for transporting temperature-sensitive commodities (e.g. perishable foodstuffs or pharmaceuticals). These containers are equipped with temperature control systems to maintain specific conditions throughout the trip and so require a connection to the vessel's electrical network. *Empty* containers correspond to dry or reefer containers that have been emptied of their commodities after delivery. They must be transported to ports where they will be refilled.

© Yousra El Ghazi, Djamal Habet, and Cyril Terrioux;
licensed under Creative Commons License CC-BY 4.0

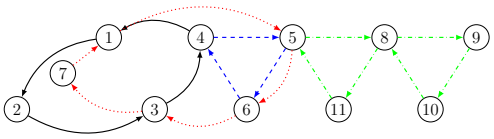


Figure 1 A liner network with four rotations (including a complex one) relying on eleven ports. A rotation is defined here by the set of arcs of the same nature.

Containers are transported in a regular liner shipping network that can be defined as a set $\mathcal{R}$ of *rotations* (or *services*). Each rotation r of $\mathcal{R}$ is defined by a sequence s_r of ports ($s_r = (p^r_1, p^r_2, \ldots, p^r_{|s_r|})$). Note that a port can appear multiple times in a given rotation. In such a case, the rotation is said *complex*. For instance, Figure 1 proposes a network with four rotations among which rotation (5, 8, 9, 10, 8, 11) is complex. Each rotation must visit the ports on a regular schedule (usually weekly [9]). Assuming a weekly schedule, the maritime company assigns to the rotation as many vessels as the number of weeks that are needed to make a complete trip and these vessels must belong to the same vessel type. Each vessel type gathers the vessels having the same features (like the size, the draft, the number of plugs for reefers, or the fuel consumption). For example, all vessels of type v can transport up to $\kappa(v)$ containers (expressed in TEU) and have $\rho(v)$ electrical plugs for reefers. We denote $\mathcal{V}_r$ the set of vessel types compatible with the visited ports of rotation r (e.g. based on size, draft restrictions, port infrastructure, or other operational constraints).

For sake of simplicity, containers having the same features (e.g. origin, destination and nature) are gathered into commodities. Each commodity k is then viewed as a number $q(k)$ of containers (expressed in TEU) to be transported from the origin port $pol(k)$ to the destination port $pod(k)$, with a revenue $rev(k)$ per TEU and a nature $nat(k)$ (reefer, dry or empty). To optimize cargo transportation, it is possible to divide a commodity into s parts if necessary, each part taking a different route. The route taken by a commodity (or one of its parts) is defined by the sequence of ports and rotations through which it transits. The route can be direct when the origin and destination ports of the commodity are on the same rotation. Otherwise, the commodity will need to transit through multiple rotations. The transition from one rotation to another is an operation called *transshipment*, which takes place at a port common to both rotations. We denote $ts_{max}(k)$ the maximum number of transshipments authorized for the commodity k. This number generally depends on the nature of the commodity. For example, reefers are rarely transshipped in practice to ensure that temperatures are respected. We denote $ts(p)$ the cost generated by transshipping one TEU at port p.

The *cargo routing problem* in a liner shipping network can be defined as follows: given a predefined network $\mathcal{R}$ regularly serving a set $\mathcal{P}$ of *ports* in a determined order, the set $\mathcal{K}_0$ of commodities, and a set $\mathcal{V}$ of *vessel* types compatible with the visited ports, the problem consists in determining which maritime routes the commodities will take to be transported from their port of origin to their port of destination while optimizing a given criterion. Depending on the needs, different criteria can be exploited, ranging from maximizing the revenues generated by the transportation of commodities, to optimizing the total quantity of commodities transported, or even minimizing the distances traveled by the containers.

This problem has various crucial applications in logistics management and planning operations. It allows shipping companies to optimize their networks by identifying the best routes to transport containers more efficiently and profitably. Furthermore, by understanding long-term transport demands and container flow patterns, investments in new vessels can

be more precisely planned, thus minimizing financial risks and ensuring adequate capacity to meet future demand. Another possible use is related to event simulation, such as the inability to pass through the Suez Canal, in order to find the most appropriate response, possibly by routing the commodities differently.

The cargo routing problem is addressed in the literature at different levels, with a primary focus on the tactical and operational levels [12, 23, 15]. The tactical level is related to the design of the liner shipping network [6]. Designing such a network requires defining the rotations and deciding which type of vessels will operate them, while taking into account the flow of commodities to be transported. The amount of carried commodities, the revenues or benefits they generate or the costs associated with the routes they take are important criteria to assess the quality of the designed network. The problem of cargo routing is therefore a crucial step in network design. Studying this problem can lead to create more efficient and profitable networks that are well-suited to the underlying cargo flows. Reinhardt and Pisinger [16] propose an exact branch-and-cut method to address the combined problem. Their mixed-integer program integrates the cargo routing problem with transshipments, considering factors such as the cost of transshipment and complex routes, which are routes in which the same port is visited twice. Wang and Meng [22] impose transit time restrictions on commodities, excluding any transshipment between rotations. To address this, they introduced a mixed-integer non-linear non-convex programming model. This method simultaneously designs the network and determines the cargo flow through it. Ameln et al. [3] introduce a novel flow-based formulation for network design, incorporating complex rotation structures and transshipment costs. Lee et al. [11] propose a framework that simultaneously optimizes service network design and commodity routing, incorporating total transit time into the minimization criteria.

The second level is operational in nature. Given an existing liner shipping network and the types of vessels associated with each rotation, its purpose is to decide how a given set of commodities is to be transported. It is important to note that here, the commodities are perfectly known (number of containers, weight, customers, etc.), whereas, at the tactical level, we are only talking about estimates based on target market shares. Moreover, the focus is on optimizing the cargo routing within the given network structure. Agarwal and Ergun [1] present one of the first scalable approaches to this problem. They propose a "route-first-flow-next" approach, where optimal routes for vessels and the number of vessels deployed on each route are determined in the first step, and then containers are routed through the network in the second step. The model incorporated cargo transshipment operations, but transshipment costs were not considered in the network design stage. The transshipment costs are taken into account in the work of Alvarez [2], with also a two-step approach used to solve the problem. This approach applies a matheuristic method to perturb the rotations using a taboo search scheme, solves the container flow problem using an interior point method, and generates new rotations from the dual variables obtained from the container flow solutions. This two-step approach has been successfully applied and expanded in multiple studies (e.g., [5, 4, 14, 10, 20, 13, 7]).

Most works in the literature focus on a single type of vessel per rotation, handle only dry containers and consider either the tactical level or the operational one. In this paper, we consider a more general approach by handling three types of containers, making it possible to divide commodities and taking as input a set of vessel types compatible with the visited ports of the rotation. This latter point allows for using our approach both at the network design level and the operational level. For network design, considering a set of compatible vessel types offers the option of choosing the type of vessels to be used for each rotation later.

The operational level is taken into account by simply considering a singleton for the set of possible vessels of each rotation. Dividing up commodities can be useful in ensuring a better vessel filling factor, which is often used as a quality indicator of rotations by shipowners. We implement our approach thanks to two COP models and a local search method. These latter are quite flexible and, notably, can exploit various optimization criteria.

This paper is structured as follows. Section 2 introduces the necessary concepts of constraint programming. Then, in Sections 3 and 4, we present two modeling approaches in the form of COP instances. In Section 5, we describe a local search method. Finally, we conduct an experimental comparison of our two models and the local method in Section 6, before drawing our conclusions in Section 7.

2 Constraint Programming

An instance P of the Constraint Optimization Problem (COP) [17] can be defined as a 4-tuple (X, D, C, f) where $X = \{x_1, \ldots, x_n\}$ is the set of *variables*, $D = \{D_{x_1}, \ldots, D_{x_n}\}$ is the set of domains, the domain D_{x_i} being related to the variable x_i, $C = \{c_1, ..., c_e\}$ represents the set of the *constraints* which define the interactions between the variables and describe the allowed combinations of values and f specifies the criterion to be optimized. Solving a COP instance $P = (X, D, C, f)$ amounts to finding an assignment of all variables of X satisfying all constraints of C and optimizing the criterion given by f. This is an NP-hard problem.

One of the advantages of constraint programming lies in the existence of specialized constraints (the *global constraints*) which will make easier the modeling of problems, but also, their solving thanks to their dedicated filtering algorithms. In the following, we will exploit the following global constraints (where $\odot$ denotes a relational operator among $\leq$, $<$, $=$, $\neq$, $>$ or $\geq$ and Y an ordered subset $\langle y_1, y_2, \ldots, y_{|Y|}\rangle$ of X):

- $\texttt{Element}(Y, i) = k$ which ensures that the ith value of Y (using a 0-based indexing) is equal to k (Y can be here an ordered set of variables or values) [21],
- $\texttt{Ordered}(Y, \odot)$ that imposes $y_1 \odot y_2 \odot \cdots \odot y_{|Y|}$.
- $\texttt{Sum}(Y, \Lambda) \odot k$ which imposes that the sum of the values of Y weighted by the coefficients of Λ satisfies the condition imposed by the relation $\odot$ with respect to k. In the following, this constraint will be represented in the more explicit form $\sum_i \lambda_i.y_i \odot k$.

3 A First CP Model

The cargo routing problem being one of the sub-problems constituting the liner shipping network design problem (LSNDP [6]), this first model is inspired by a part of the model presented in [8]. However, it allows going further by taking into account complex rotations. The idea of this first model (denoted $\mathcal{M}_1$) is to deduce the routes taken by the commodities from the rotations.

3.1 Commodities

In order to transport as many commodities as possible, we consider the possibility that a commodity may be split into s parts. Each of these parts can then be transported independently of the others (i.e., using different routes). Let $\mathcal{K}$ be the set of parts obtained by splitting each commodity in $\mathcal{K}_0$ into s parts. We denote $\mathcal{K}_k^s$ the set of parts related to commodity k. Each part has the same POL, POD and nature as the initial commodity. Note that the sum of the number of TEUs of the parts of commodity k cannot exceed $q(k)$

$$\sum_{j \in \mathcal{K}_k^s} q_j \leq q(k) \qquad k \in \mathcal{K}_0 \text{ (C.1)}$$

$$\texttt{Ordered}(\langle q_j | j \in \mathcal{K}_k^s \rangle, \geq) \qquad k \in \mathcal{K}_0 \text{ (C.2)}$$

$$\alpha_j = 1 \iff q_j > 0 \qquad j \in \mathcal{K} \text{ (C.3)}$$

Figure 2 Constraints related to the commodities (used in models $\mathcal{M}_1$ and $\mathcal{M}_2$).

and a part can be empty. Furthermore, if, for the sake of simplicity, we consider here the same value s for all commodities, the model could easily be adapted to exploit a value s per commodity.

For each part j of a commodity k, we introduce an integer variable q_j with value in $[0, q(k)]$ to represent the number of TEUs associated with part j and a Boolean variable α_j indicating whether the part is used or not. Constraint (C.1) (see Figure 2) ensures that the number of TEUs transported for commodity k does not exceed $q(k)$. Constraint (C.2) allows to break some symmetries, while constraint (C.3) ensures the consistency between the values of the variables α_j and q_j.

3.2 Cargo Flow

For each part j of a commodity k, we consider a variable nbr_j with domain $[0, ts_{max}(k) + 1]$ to express the total number of rotations used to transport part j. To determine the ports where each part is loaded or unloaded, we introduce a Boolean variable $from^r_{j,i}$ (resp. $to^r_{j,i}$) which is equal to 1 if part j is loaded (resp. unloaded) at port p^r_i of rotation r, 0 otherwise.

Constraint (F.1) (resp. (F.2)) in Figure 3 ensures that a part j is transported if and only if there exists a rotation in which it is loaded at its port $pol(j)$ (resp. unloaded at its port $pod(j)$). The number of rotations used to transport a part j corresponds to the number of ports in which the part j is (un)loaded (constraints (F.3) and (F.4)). For a port p^r_i, a part j cannot be both loaded and unloaded at the same time (constraint (F.5)). A part j is loaded (resp. unloaded) in a given port p at most once if it is accepted, 0 otherwise (constraints (F.6) and (F.7)). If a part j is loaded (resp. unloaded) at a port p^r_i distinct from $pol(j)$ (resp. $pod(k)$), then it must be unloaded (resp. loaded) at the same port, but for another rotation (constraints (F.8) and (F.9)). The constraint (F.10) ensures that, for each rotation, a part j is loaded and unloaded an equal number of times. Let d_r be the maximum number of occurrences of the same port in rotation r. d_r is greater than 1 in the case of complex rotations. For each rotation r, a part j cannot be (un)loaded more than d_r times (constraints (F.11) and (F.12)).

Finally, it is necessary to know which parts are transported at the exit of each port of each rotation. To achieve this, we introduce a Boolean variable $leave^r_{j,i}$ which equals 1 if part j is transported by rotation r at the exit of port p^r_i, 0 otherwise. Constraints (F.13)–(F.17) ensure the consistency of the values of variables $from^r_{j,i}$, $to^r_{j,i}$ and $leave^r_{j,i}$. Constraint (F.18) guarantees that a part j cannot pass through the same port multiple times. Similarly, a part j cannot leave its destination port (constraint (F.19)).

3.3 Vessels Capacities

We must ensure that when leaving each port, vessels do not exceed their capacities in terms of the number of containers carried and the number of plugs used for reefer containers. Since the maximum values of these two quantities are not correlated (in particular, the vessel

$$\sum_{r\in\mathcal{R}}\sum_{p_i^r\in s_r|p_i^r=pol(j)} from_{j,i}^r = \alpha_j \qquad j\in\mathcal{K} \text{ (F.1)}$$

$$\sum_{r\in\mathcal{R}}\sum_{p_i^r\in s_r|p_i^r=pod(j)} to_{j,i}^r = \alpha_j \qquad j\in\mathcal{K} \text{ (F.2)}$$

$$\sum_{r\in\mathcal{R}}\sum_{p_i^r\in s_r} from_{j,i}^r = nbr_j \qquad j\in\mathcal{K} \text{ (F.3)}$$

$$\sum_{r\in\mathcal{R}}\sum_{p_i^r\in s_r} to_{j,i}^r = nbr_j \qquad j\in\mathcal{K} \text{ (F.4)}$$

$$from_{j,i}^r + to_{j,i}^r \leq 1 \qquad j\in\mathcal{K}, r\in\mathcal{R}, p_i^r\in s_r \text{ (F.5)}$$

$$\sum_{r\in\mathcal{R}}\sum_{p_i^r\in s_r|p_i^r=p} from_{j,i}^r \leq \alpha_j \qquad j\in\mathcal{K}, p\in\mathcal{P} \text{ (F.6)}$$

$$\sum_{r\in\mathcal{R}}\sum_{p_i^r\in s_r|p_i^r=p} to_{j,i}^r \leq \alpha_j \qquad j\in\mathcal{K}, p\in\mathcal{P} \text{ (F.7)}$$

$$from_{j,i}^r = 1 \Rightarrow \sum_{r'\in\mathcal{R}\setminus\{r\}}\sum_{p_i^{r'}\in s_{r'}|p_{i'}^{r'}=p_i^r} to_{j,i'}^{r'} = 1 \qquad j\in\mathcal{K}, r\in\mathcal{R}, p_i^r\in s_r, p_i^r\neq pol(j) \text{ (F.8)}$$

$$to_{j,i}^r = 1 \Rightarrow \sum_{r'\in\mathcal{R}\setminus\{r\}}\sum_{p_i^{r'}\in s_{r'}|p_{i'}^{r'}=p_i^r} from_{j,i'}^{r'} = 1 \qquad j\in\mathcal{K}, r\in\mathcal{R}, p_i^r\in s_r, p_i^r\neq pod(j) \text{ (F.9)}$$

$$\sum_{p_i^r\in s_r} from_{j,i}^r = \sum_{p_i^r\in s_r} to_{j,i}^r \qquad j\in\mathcal{K}, r\in\mathcal{R} \text{ (F.10)}$$

$$\sum_{p_i^r\in s_r} from_{j,i}^r \leq \alpha_j.d_r \qquad j\in\mathcal{K}, r\in\mathcal{R} \text{ (F.11)}$$

$$\sum_{p_i^r\in s_r} to_{j,i}^r \leq \alpha_j.d_r \qquad j\in\mathcal{K}, r\in\mathcal{R} \text{ (F.12)}$$

$$from_{j,i}^r = 1 \Rightarrow leave_{j,i}^r = 1 \qquad j\in\mathcal{K}, r\in\mathcal{R}, p_i^r\in s_r \text{ (F.13)}$$

$$to_{j,i}^r = 1 \Rightarrow leave_{j,i}^r = 0 \qquad j\in\mathcal{K}, r\in\mathcal{R}, p_i^r\in s_r \text{ (F.14)}$$

$$(from_{j,i+1}^r = 0 \wedge to_{j,i+1}^r = 0) \iff leave_{j,i}^r = leave_{j,i+1}^r \qquad j\in\mathcal{K}, r\in\mathcal{R}, p_i^r\in s_r \text{ (F.15)}$$

$$(leave_{j,i}^r = 0 \wedge leave_{j,i+1}^r = 1) \Rightarrow from_{j,i+1}^r = 1 \qquad j\in\mathcal{K}, r\in\mathcal{R}, p_i^r\in s_r \text{ (F.16)}$$

$$(leave_{j,i}^r = 1 \wedge leave_{j,i+1}^r = 0) \Rightarrow to_{j,i+1}^r = 1 \qquad j\in\mathcal{K}, r\in\mathcal{R}, p_i^r\in s_r \text{ (F.17)}$$

$$\sum_{r\in\mathcal{R}}\sum_{p_i^r\in s_r|p_i^r=p} leave_{j,i}^r \leq \alpha_j \qquad j\in\mathcal{K}, p\in\mathcal{P} \text{ (F.18)}$$

$$\sum_{r\in\mathcal{R}}\sum_{p_i^r\in s_r|p_i^r=pol(j)} leave_{j,i}^r = 0 \qquad j\in\mathcal{K} \text{ (F.19)}$$

Figure 3 Cargo flow constraints (used in model $\mathcal{M}_1$).

$$\texttt{Element}(\langle \kappa(v) | v \in \mathcal{V} \rangle, v_r) = \kappa_r \qquad r \in \mathcal{R} \text{ (V.1)}$$

$$\texttt{Element}(\langle \rho(v) | v \in \mathcal{V} \rangle, v_r) = \rho_r \qquad r \in \mathcal{R} \text{ (V.2)}$$

$$\sum_{j \in \mathcal{K}} q_j.leave^r_{j,i} \leq \kappa_r \qquad r \in \mathcal{R}, p^r_i \in s_r \text{ (V.3)}$$

$$\sum_{j \in \mathcal{K}_{reefers}} q_j.leave^r_{j,i} \leq \rho_r \qquad r \in \mathcal{R}, p^r_i \in s_r \text{ (V.4)}$$

Figure 4 Capacity constraints (used in models $\mathcal{M}_1$ and $\mathcal{M}_2$).

$$\max \sum_{j \in \mathcal{K}} q_j \text{ (O.1)}$$

$$\max \sum_{j \in \mathcal{K}} q_j.rev(j) \text{ (O.2)}$$

$$\max \sum_{j \in \mathcal{K}} q_j.rev(j) - \sum_{j \in \mathcal{K}, r \in \mathcal{R} | p^r_i \neq pol(j)} to^r_{j,i}.q_j.ts(p^r_i) \text{ (O.3)}$$

$$\max \sum_{j \in \mathcal{K}, \ell \in routes(j)} q_j.\beta^\ell_j.cost_{ts}(route(j,\ell)) \text{ (O.4)}$$

Figure 5 Possible objective functions (used in models $\mathcal{M}_1$ and/or $\mathcal{M}_2$).

capable of carrying the most containers may not necessarily have the most plugs), we must ensure the consistency of these two quantities based on the types of vessels available. To achieve this, we introduce, for each rotation r, a variable v_r specifying the type of vessels considered, and two integer variables κ_r and ρ_r representing the maximum capacity in terms of containers and the maximum number of available plugs, respectively.

Constraints (V.1) and (V.2) ensure the consistency of the maximum capacities based on the type of vessels, while constraints (V.3) and (V.4) ensure compliance with these maximum capacities upon departure from each port ($\mathcal{K}_{reefers}$ denotes the set of parts j corresponding to reefer containers).

3.4 Objective Functions

The function (O.1) (see Figure 5) aims to maximize the number of commodities transported, while function (O.2) maximizes the revenue generated by the commodities transported. Function (O.3) aims to minimize the benefits where the benefits are defined as the revenue generated by commodities minus the transshipment costs. It should be noted that the other costs (like fuel cost) cannot be taken into account at this level, but are more a matter of the network design problem.

4 A Second CP Model

In this second model (denoted $\mathcal{M}_2$), we consider as input each of the possible routes for each commodity. To determine these routes, we first construct the rotation graph. This graph has a vertex for each rotation. There is an edge between two vertices if the two rotations share at least one port. The edges are then labeled by the shared ports. This graph allows us to quickly identify the ports where transshipments can take place. For each

$$\sum_{\ell \in routes(j)} \beta_j^\ell = \alpha_j \qquad j \in \mathcal{K} \text{ (R.1)}$$

$$\sum_{p_i^r \in route(j,\ell)} \beta_j^\ell = leave_{j,i}^r \qquad j \in \mathcal{K}, r \in \mathcal{R}, p_i^r \in s_r, p_i^r \neq pod(j) \text{ (R.2)}$$

Figure 6 Constraints related to routes (used in model $\mathcal{M}_2$).

commodity k, we compute the set of possible routes by calculating all paths starting from the rotations containing the port $pol(k)$ and arriving at a rotation containing the port $pod(k)$ and containing at most $ts_{max}(k)$ transshipments.

Given a commodity k, a route is a sequence $(s_1^k, s_2^k, \ldots, s_{t_k}^k)$ of port sequences such that:

- s_ℓ^k is a sequence of ports $(p_1^{r_\ell}, p_2^{r_\ell}, \ldots, p_{|s_\ell^k|}^{r_\ell})$ (with $|s_\ell^k| \geq 2$) associated with the rotation r_ℓ such that $p_m^{r_\ell}$ is the port preceding the port $p_{m+1}^{r_\ell}$ in the rotation r_ℓ. The first port in the sequence s_ℓ^k corresponds to the port where the commodity enters rotation r_ℓ while the last one is where it leaves this rotation.
- $p_1^{r_1} = pol(k)$.
- $p_{|s_\ell^k|}^{r_\ell} = p_1^{r_{\ell+1}}$, this means that this port is used for transshipment.
- $p_{|s_{t_k}^k|}^{r_{t_k}} = pod(k)$.

Note that t_k minus one is the number of transshipments of commodity k. We denote $routes(k)$ the set of all possible routes for transporting commodity k and $route(k,i)$ the i-th route of $routes(k)$. We extend these notations to each part j of each commodity k.

This second model includes the variables q_j, α_j, $leave_{j,i}^r$, v_r, κ_r and ρ_r from the first model, as well as the constraints presented in Figures 2 and 4. The difference between the two models lies simply in the way the routes are represented. In this second model, for each part j, we consider a Boolean variable β_j^ℓ which equals 1 if route $route(j,\ell)$ is used to transport part j, 0 otherwise. Constraint (R.1) (see Figure 6) ensures that a part j uses at most one route, while constraint (R.2) establishes the link with the variables $leave_{j,i}^r$. Regarding the objective functions, we can exploit functions (O.1) and (O.2) from Figure 5 respectively for maximizing the number of carried TEUs and the revenue. Function (O.4) allows for maximizing the benefits where $cost_{ts}(route(j,\ell))$ denotes the total transshipment cost per TEU for the ℓ-th route of commodity j.

5 An Incomplete Method

A real instance of the cargo routing problem can easily involve several dozen ports and several hundred of commodities for a total volume of more than 100,000 TEUs. The ability of complete solvers to solve such instances in a reasonable time can therefore quickly become limited. In this section, we present an incomplete method that is intended to be generic, in the sense that it does not depend on the criterion to be optimized. This method (called *Route*, see Algorithm 1) takes, as inputs, the set $\mathcal{R}$ of rotations and the set $\mathcal{K}_0$ of commodities to be transported, and will return the best solution $\mathcal{K}_p^b$ that it has found according to the criterion considered. A solution is represented here by a set of triplets (k,n,r) representing the assignment of n TEUs of commodity k to route r. The *Route* method begins with an initialization phase (lines 1-5) during which it computes all possible routes for each commodity k in $\mathcal{K}_0$ (line 2) according to the same process as model $\mathcal{M}_2$. A first solution $\mathcal{K}_p^{init}$ is used to initialize the current solution $\mathcal{K}_p$ (lines 4 and 11). For example, it can be provided by the user or constructed from scratch using the *Assign* method (see Algorithm 2).

Algorithm 1 $Route(\mathcal{R}, \mathcal{K}_0)$.

```
1  foreach k ∈ K_0 do
2      Compute all the possible routes for k
3      nb[k] ← q(k)
4  K_p ← K_p^init
5  K_p^b ← K_p
6  for j from 1 to #rst do
7      for i from 1 to #trial do
8          Apply Move(K_p) or Remove(K_p, nb)
9          Assign(K_p, nb)
10         if K_p better than K_p^b then K_p^b ← K_p
11     if j < #rst then K_p ← Select(K_p^init, K_p^m)
12 return K_p^b
```

Algorithm 2 $Assign(\mathcal{K}_p, nb)$.

```
1  foreach k ∈ K_0 do
2      s ← |{r|(k, x, r) ∈ K_p}|
3      while nb[k] > 0 and s ≤ s_max and routes(K_p, k) ≠ ∅ do
4          (n, r) ← best_route(K_p, k)
5          K_p ← K_p ∪ {(k, n, r)}
6          nb[k] ← nb[k] − n
7          s ← s + 1
```

Algorithm 3 $Move(\mathcal{K}_p)$.

```
1  foreach (k, n, r) ∈ K_p do
2      s ← |{r|(k, x, r) ∈ K_p}|
3      while n > 0 and s ≤ s_max and routes(K_p, k) ≠ ∅ do
4          (n', r') ← best_route(K_p, k)
5          if n ≤ n' then
6              K_p ← (K_p \ {(k, n, r}) ∪ {(k, n', r')}
7              n ← 0
8          else
9              if s < s_max then
10                 K_p ← K_p ∪ {(k, n', r')}
11                 n ← n − n'
```

Algorithm 4 $Remove(\mathcal{K}_p, nb_{rem})$.

```
1  while |K_p| > 0 and nb_rem > 0 do
2      Choose randomly (k, n, r) ∈ K_p
3      K_p ← K_p − {(k, n, r)}
4      nb_rem ← nb_rem − n
5      nb[k] ← nb[k] + n
```

Then, the *Route* method tries to improve the quality of the current solution by alternating between a phase where it moves or removes commodities (line 8) and a phase where it places the commodities on routes (line 9). This alternation is repeated $\#trial$ times (lines 7-10). At the end of each iteration, the best solution $\mathcal{K}_p^b$ is updated if necessary (line 10). Once the allotted number of trials for a given initial solution has been completed, we restart the search from the initial solution or the best solution found to avoid the problem of local optima (line 11). Note that heuristics can be used to choose between moving or removing commodities (line 8) or restarting from the initial solution or the best solution found (line 11). We give some examples of such heuristics in Section 6.

The objective of the *Assign* method is to assign as many commodities as possible to the available routes within the limit of the number s_{max} of possible divisions of the same commodity k. Thus, for each commodity k for which there are still containers to be placed, this method will identify the possible routes among the routes initially available for the commodity k. A route is possible if the rotations it uses are able to accept commodity in terms of load (for reefer, dry or empty containers) or in terms of electrical plugs (for reefer containers). The function *routes* returns the list of possible routes for a commodity k with respect to the current solution $\mathcal{K}_p$. To do this, it relies on the types of vessels available for each rotation. The available types are constantly evolving due to the addition, move or removal of commodities, ensuring that at each step, there is at least one possible vessel type for each rotation. The *best_route* function returns the available route r that allows transporting the most containers of commodity k depending on its nature as well as this number n of containers. The *Move* method (see Algorithm 3) is based on a similar principle. The main difference is that a quantity n of commodity k is initially transported via route r and that a part or all of this quantity will be transported via a new route r'. The underlying idea is to free up space (e.g. on a potentially saturated route) to allow the routing of another commodity. Finally, the *Remove* method (see Algorithm 4) allows the removal of some commodities from some routes. With this aim in view, we randomly select a commodity and a route from $\mathcal{K}^p$ and remove the related number of TEUs from this route. We repeat this process until at least nb_{rem} TEUs have been removed.

Different heuristics can be used to decide the order in which the commodities are processed in *Assign* and *Move* methods. For example, for the *Assign* method, we can start by processing the commodities that have the fewest possible routes, or sort them according to the remaining quantity to be routed. For the *Move* method, we can prioritize commodities transported via rotations where vessels are full when leaving certain ports. Furthermore, several different criteria can be considered (line 10 of Algorithm 1) such as, for example, maximizing the number of containers transported, the revenue generated or the benefits. The approach presented here is therefore relatively general and can easily be specialized according to needs. It is parameterized by the number nb_{rem} of commodities to remove at each call to the *Remove* method, the number $\#rst$ of restarts and the number $\#trial$ of iterations per restart performed in the *Route* method. Finally, as we will see in the next section, it can be used both to help design a network and to route commodities at the operational level.

Algorithm 1 can be considered as a case of the large neighborhood search (LNS) [18, 19]. LNS operates by partially destroying and then repairing a solution while controlling the search strategy for optimization problems. In Algorithm 1, the move/remove operations (line 8) can be seen as destroying part of the solution, while the assignment step (line 9) serves as the repairing phase. Restarts from a new solution and moving commodities help diversify the search. Intensification occurs during the assignment steps of commodities or when restarting from the best solution, which is saved or accepted throughout the search process.

6 Experiments

6.1 Experimental Protocol

6.1.1 Benchmark

In the absence of a public dataset for the cargo routing problem, we construct instances as realistically as possible using data from LINER-LIB [5], which serves as the benchmark for experiments on LSNDP. We consider the ports and vessel types from LINER-LIB. However, for vessels, their loading capacity is randomly drawn between 75% and 100% of their maximum capacity, as they are rarely operated at full capacity due to various factors. Furthermore, as LINER-LIB does not take into account reefer containers, we randomly set the number of plugs between 5% and 10% of the vessel's maximum capacity, to match values typically observed in practice. Then, concerning the commodities, we use the (POL, POD) pairs from the LINER-LIB instances. For a given pair, we generate a number of containers and associated revenues by randomly modulating the values provided by the LINER-LIB. Between 5% and 12% of these containers will be reefer containers, the rest being full ones. Finally, we randomly add a certain number of empty containers so that for each port, the balance between the number of incoming and outgoing containers is not unbalanced by more than 25%.

In our experiments, we randomly generate 40 instances using data from the `Baltic`, `WAF`, `Mediterranean` and `Pacific` from the LINER-LIB (10 instances per LINER-LIB instance). These instances have between 12 and 45 ports, from 46 to 1,409 commodities, for a total number of TEUs between 12,898 and 124,808. For each of these instances, we consider four different sets of rotations. Each set contains between 3 and 20 rotations, each involving between 2 and 10 ports. For each rotation, the number of associated vessel types depends on the use we make of the cargo routing problem. If we consider cargo routing as a step of network design (i.e. the tactical level), we use all vessel types compatible with the ports visited by the rotation. In contrast, for operational level, we fix arbitrarily a type of a vessel consistent with the ports of the rotation. In both cases, we thus obtain 160 instances of the cargo routing problem[1]. Each of these instances is solved by considering, on the one hand, a complete method based on one of the two models $\mathcal{M}_1$ or $\mathcal{M}_2$ and on the other hand, our incomplete method.

6.1.2 Implementation Details and Settings

The $\mathcal{M}_1$ and $\mathcal{M}_2$ models are implemented in OR-Tools (version 9.12.4544[2]) via its Python interface and solved using the CP-SAT solver of OR-Tools. This choice is guided by the solver's efficiency and the ability to use multiple processes. Except for the number of threads set to 16, we use the solver's default parameters. The incomplete method *Route* is implemented in Python 3 and launched in a single thread. Its number of trials varies between 10 and 250, and the number of commodities removed is fixed to 20% of the number of commodities in the current solution. The number of restarts ranges between 1 and 25. In order to sort commodities, we introduce two counters c_k and nb_k^f per commodity k. c_k represents the total value of containers of commodity k that are not transported in the

[1] These instances are available at `https://pageperso.lis-lab.fr/cyril.terrioux/Cargo_Routing/instances.zip`

[2] `https://developers.google.com/optimization/cp/cp_solver`

current solution. The value of containers depends on the criterion under consideration. For instance, it corresponds to the number of TEUs (resp. the revenue generated by the carried containers) when we maximize the number of containers carried (resp. the global revenue or the benefits). Initially, c_k is initialized with the number $nb(k)$ of containers of commodity k or the associated revenue $nb(k).rev(k)$ depending on the chosen criterion. nb_k^f is the cumulative number of TEUs of commodity k that are not transported in the solutions already produced. We initialize this counter with the number of possible routes for commodity k. In the *Assign* method, the remaining commodities are processed according to the decreasing ratio $\frac{c_k}{nb_k^f}$. By so doing, we try to identify the most valuable commodities that are the most difficult to transport, so that they can be prioritized for placement on board vessels. Regarding the choice between move and remove operations (line 8), we perform moves as long as they contribute to improve the current solution, and we carry out a removal phase when this is no longer the case. When restarting (line 11), we use the best solution found half of the time. The rest of the time and initially, we start with an empty solution. As some steps of our incomplete method involve randomness, each instance is solved 50 times. The experiments are conducted on servers equipped with Intel Xeon Gold 5218R processors clocked at 2.1 GHz and equipped with 192 Go of memory. For complete and incomplete approaches, the runtime is limited to one hour per instance. Finally, the commodities are divided into at most two parts (i.e. $s = 2$). This choice is made to allow comparison between the two models and the incomplete method. In fact, the runtime of a complete approach increases exponentially with s, whereas for our incomplete method, increasing s has little impact on runtime.

It should be noted that we cannot compare our results with those of methods from the literature. On the one hand, these methods do not address exactly the same variant of the problem. Specifically, they only consider one type of container (i.e. dry containers) and one type of vessel per rotation. On the other hand, their sources or executables are not available.

6.2 Study of the Behavior of the Incomplete Search

Before comparing the different approaches, we first analyze the behavior of the proposed incomplete method. First of all, our experiments have shown that the criterion to be optimized does not influence the time or quality of the results obtained. As the trends are often similar and due to lack of space, we will not present the results for each criterion considered hereafter. Such a result is not surprising since the criterion plays a minor role in Algorithm 1, impacting only line 10. Then, our method is relatively robust. Indeed, it can be observed that from one execution to another, the results are substantially the same, with a standard deviation of at most 4.3% in very rare cases and less than 1% in general. Figure 7(a) presents, for example, the dispersion of the values of the best solution for each of the ten `Pacific` instances when maximizing the quantity of transported commodities at the tactical level for one restart and 250 trials. Figure 7(b) does the same for 25 restarts and 10 trials per restart.

Next, we measure the impact of the number of trials by looking at the trial (out of 250) on which the best solution is found. Figure 8(a) presents the number of instances for which the best solution is found in a given interval of trials for a single restart. We consider here both tactical and operational instances. Clearly, for instances from `Baltic`, `WAF` and `Mediterranean`, whatever criterion, a number of trials of about 10 turns out to be relevant in most cases. For `Pacific` instances, it would seem that 50 trials would be more appropriate. As the runtime increases with the number of trials, the value 10 can generally be considered as a good trade-off between the runtime and the quality of the obtained solution. Then, we measure the impact of the number of restarts by looking at the run out of 25 on

Table 1 Number of instances (out of 40) for which optimality is proved for $\mathcal{M}_1$ and $\mathcal{M}_2$.

	Tactical level			Operational level		
	TEU	Revenue	Benefit	TEU	Revenue	Benefit
`Baltic`	39 / 38	40 / 40	40 / 40	39 / 38	40 / 40	40 / 40
`WAF`	40 / 40	40 / 40	40 / 40	40 / 40	40 / 40	40 / 40
`Mediterranean`	35 / 39	34 / 34	0 / 0	36 / 39	36 / 30	0 / 0
`Pacific`	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
Total	114 / 117	114 / 114	80 / 80	115 / 117	116 / 110	80 / 80

which the best solution is found. Figure 8(b) presents the number of instances for which the best solution is found in a given interval of restarts for 10 trials per restart. It appears that few restarts are often enough to obtain a good-quality solution. Again, the `Pacific` instances require an extra effort compared with those of other families. Indeed, the best results often require a number of restarts greater than 10. This can be explained by the size of the instances (the largest considered here), but also by the reduced number of iterations used here. Using 50 trials instead of 10 does imply a significant change (see Figure 7(c)). Figures 7(d) and (e) provide the dispersion for the quality of the solution and the runtime when maximizing the quantity of transported commodities for the instance `Pacific_9` (at tactical level). We can observe a slight improvement in the quality when the number of restarts increases. Here, a 1% improvement in solution quality requires a 9-fold increase in runtime. So, using a small number of restarts seems to be a good trade-off between runtime and solution quality when solving the problem of cargo routing at tactical level, as such a problem will need to be solved a large number of times. In an operational context, focusing on profitability would require to spend more time on problem-solving, which would involve using the method with several dozen restarts.

6.3 Comparisons of the Different Approaches

6.3.1 Tactical level

First, we consider the tactical level where the cargo routing problem is used as a liner network design step. The aim is to evaluate the quality of the network produced according to a given criterion, such as the number of TEUs transported or the revenue/benefits generated. It should be noted that network design as a whole is beyond the scope of this paper.

We first compare the $\mathcal{M}_1$ and $\mathcal{M}_2$ models. These two models are relatively close in terms of performance. Model $\mathcal{M}_1$ optimally solves 114 instances compared to 117 for model $\mathcal{M}_2$ when maximizing the quantity of transported commodities. For the two other criteria, both models optimally solve the same number of instances. Whatever the considered criterion, it turns out that, in general, instances based on `Baltic` and `WAF` are optimally solved. These instances are the smallest and correspond to feedering rotations (i.e. short rotations intended to supply main lines). In contrast, for instances based on `Pacific` (i.e. general instances integrating feedering and main lines), after one hour, the solver is still searching for the optimal solution, regardless of the model considered. We have observed that, most of the time, the last best solution is found around the time limit. This means that the solver has not yet begun the proof of optimality. Finally, we can note that maximizing the benefits (with a more complex objective function) seems to make the problem harder to solve. In particular, we can note that no `Mediterranean` instance is solved optimally.

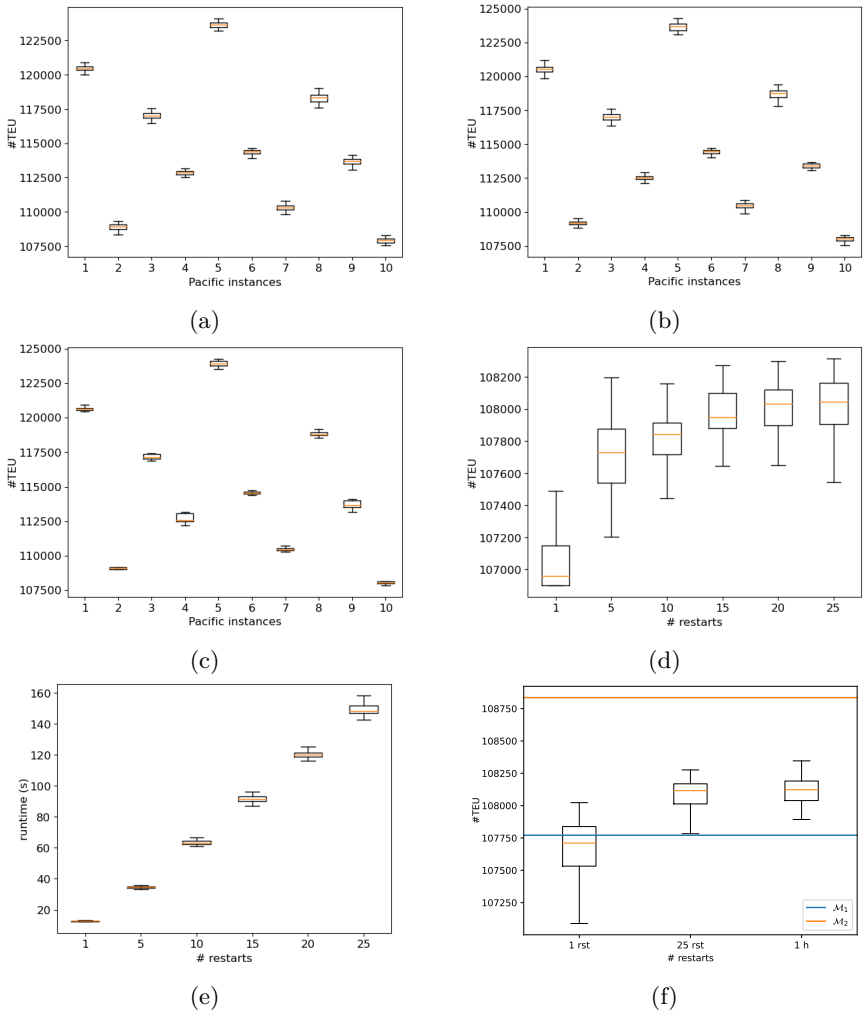


Figure 7 Quality dispersion of the incomplete method when maximizing the number of TEUs transported for a single restart and 250 trials (a) and 25 restarts and 10 trials per restart (b) and 50 trials (c), the quality (d) and runtime (e) for instance `Pacific_9` depending on the number of restarts and 10 trials per restart. Quality dispersion obtained by the incomplete method when it is respectively run for 1 restart, 25 restarts and 1 hour and comparison with the best solution obtained by models $\mathcal{M}_1$ and $\mathcal{M}_2$ when maximizing the number of TEUs for `Pacific_9` instance (f). All these figures consider the tactical level.

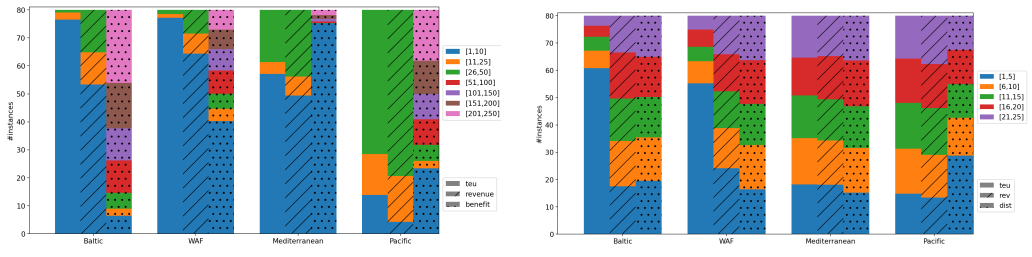
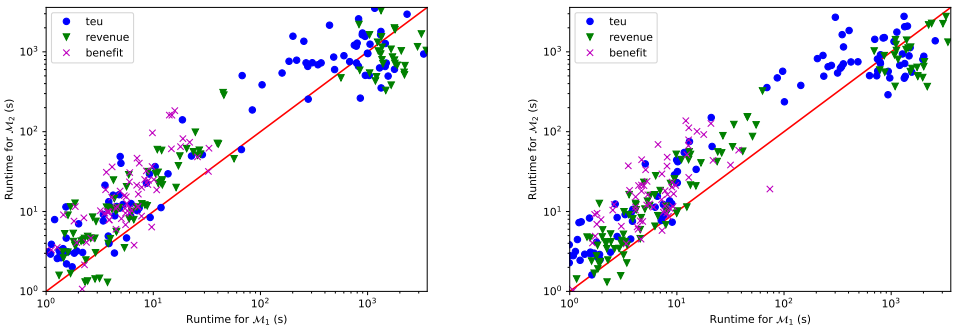
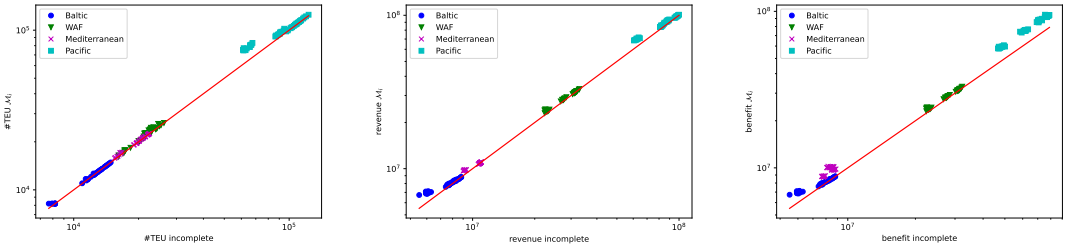


Figure 8 Impact of the number of trials (a) and restarts (b).



(a) (b)

Figure 9 Runtime comparison for models $\mathcal{M}_1$ and $\mathcal{M}_2$ for tactical (a) and operational use (b) for instance for which the optimality is proved.



(a) (b) (c)

Figure 10 Comparisons of the quality of solutions obtained by the incomplete method (for 25 restarts and 10 trials per restart) vs. the best solution obtained by models $\mathcal{M}_1$ and $\mathcal{M}_2$ when maximizing the number of carried TEUs (a) revenue (b) and benefit (c) for tactical level.

If we compare the solving runtime for each model on instances which are optimally solved (see Figure 9(a)), we note that the solving is often faster with $\mathcal{M}_1$ than with $\mathcal{M}_2$. However, if we consider the quality of the best found solution for the other instances, generally, $\mathcal{M}_2$ finds slightly better solutions whatever the criterion. Nevertheless, the gap between the two models does not exceed 3% and it is often less than 1%.

Now, we consider the incomplete *Route* method and compare the quality of the solution it finds with the best solution found by one of the two models. Figure 10 allows estimating the quality of the solutions found by comparing the result of the incomplete search with the best solution found by one of the two models for all the considered criterion. We consider here 25 restarts and 10 trials per restart. The reported result for the incomplete search is the worst result found over the 50 runs performed for each instance. By doing so, we want to assess the quality of our incomplete approach in the worst case[3]. This comparison shows that the incomplete method often obtains results close to those of a complete approach. We can see that, for some `Baltic` instances, the incomplete method performs slightly less well when maximizing the revenue or the benefits. This corresponds to instances for which the number

[3] Considering the best result over the 50 runs would only slightly improve the results, as the quality dispersion is low (see Figure 7).

of routes per commodity is reduced, making it more difficult for the incomplete method to find a good solution. Whatever the criterion, a similar trend can be observed for some `Pacific` instances with the same explanation. In terms of runtime, it takes a maximum of around 170 seconds for 25 restarts and 10 trials per restart for the largest instances, whereas Or-Tools systematically reaches the time-out for these instances although it uses 16 cores compared with one for the incomplete method. The incomplete method therefore delivers quality results in a reasonable time whatever the considered criterion. We can see from Figure 10(f) that a reasonable number of restarts and trials can do better, in some cases, than a complete approach, and that increasing the runtime still further does not lead to a significant improvement in results.

6.3.2 Operational use

We consider the case where the cargo routing problem is used in an operational context. At this stage, the network has already been built, i.e. the rotations have already been defined and a specific type of vessel is associated with each of them. In this case, the cargo routing problem is used to decide how commodities should be routed within the network. While this use can be seen as a special case of the tactical level, it does not make the solving easier. On the contrary, reducing the possible vessel types to just one per rotation further constrains the problem. As a result, we can note that model $\mathcal{M}_2$ the model solves slightly fewer instances optimally. Beyond that, the trends observed for the tactical level remain valid and reinforce the interest of the incomplete approach.

It should be noted that these experiments were carried out with at most two splittings of each commodity, to enable comparison between the complete approaches and our incomplete method. Increasing the number of splittings significantly increases the difficulty of the problem when solved with a complete search. In contrast, our incomplete method can divide commodities into as many parts as desired without any noticeable increase in time. It is therefore better suited to use in real-life conditions, in particular to enable shipowners to achieve good values for quality indicators such as vessel filling factor.

7 Conclusions and Perspectives

In this paper, we have proposed two CP-based models for solving the cargo routing problem, as well as an incomplete method. The results we have obtained show the difficulty of optimally solving the cargo routing problem with a complete approach and so highlight the practical interest of incomplete methods such as the one we propose. This latter turns to be robust and provides relevant solutions within a very reasonable runtime whatever the criterion we consider. Our results also point out the difference in difficulty depending on the optimization criterion considered or the nature of the instances (feedering or general). On this latter point, a possible improvement for general instances would be to processes feedering and main line rotations separately.

References

1 R. Agarwal and Ö. Ergun. Ship Scheduling and Network Design for Cargo Routing in Liner Shipping. *Transportation Science*, 42(2):175–196, 2008. `doi:10.1287/trsc.1070.0205`.

2 J. F. Álvarez. Joint routing and deployment of a fleet of container vessels. *Maritime Economics & Logistics*, 11(2):186–208, 2009. `doi:10.1057/mel.2009.5`.

3 M. Ameln, J. S. Fuglum, K. Thun, H. Andersson, and M. Stålhane. A new formulation for the liner shipping network design problem. *International Transactions in Operational Research*, 28(2):638–659, 2021. doi:10.1111/itor.12659.

4 B. D. Brouer, G. Desaulniers, C. V. Karsten, and D Pisinger. A Matheuristic for the Liner Shipping Network Design Problem with Transit Time Restrictions. In *Computational Logistics - 6th International Conference, ICCL*, volume 9335 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2015. doi:10.1007/978-3-319-24264-4_14.

5 B. D. Brouer, J. F. Álvarez, C. E. M. Plum, D. Pisinger, and M. M. Sigurd. A base integer programming model and benchmark suite for liner-shipping network design. *Transportation Science*, 48(2):281–312, 2014. doi:10.1287/trsc.2013.0471.

6 M. Christiansen, E. O. Hellsten, D. Pisinger, D. Sacramento, and C. Vilhelmsen. Liner shipping network design. *European Journal of Operational Research*, 286, 2020. doi:10.1016/j.ejor.2019.09.057.

7 D. F. Koza. Liner shipping service scheduling and cargo allocation. *EJOR*, 275, December 2018. doi:10.1016/j.ejor.2018.12.011.

8 Y. El Ghazi, D. Habet, and C. Terrioux. A CP Approach for the Liner Shipping Network Design Problem. In *CP*, pages 16:1–16:21, 2023. doi:10.4230/LIPICS.CP.2023.16.

9 M. Giovannini and H. Psaraftis. The profit maximizing liner shipping problem with flexible frequencies: logistical and environmental considerations. *Flexible Services and Manufacturing Journal*, 31, September 2019. doi:10.1007/s10696-018-9308-z.

10 C. V. Karsten, B. D. Brouer, G. Desaulniers, and D. Pisinger. Time constrained liner shipping network design. *Transportation Research. Part E: Logistics and Transportation Review*, 105:152–162, 2017. doi:10.1016/j.tre.2016.03.010.

11 S. Lee, T. Boomsma, and K. Holst. Approximate dynamic programming for liner shipping network design. *Annals of Operations Research*, pages 1–40, July 2024. doi:10.1007/s10479-024-06106-1.

12 Q. Meng, S. Wang, H. Andersson, and K. Thun. Containership routing and scheduling in liner shipping: overview and future research directions. *Transportation Science*, 48(2):265–280, January 2014. doi:10.1287/trsc.2013.0461.

13 R. Neamatian Monemi and S. Gelareh. Network design, fleet deployment and empty repositioning in liner shipping. *Transportation Research Part E: Logistics and Transportation Review*, 108, July 2017. doi:10.1016/j.tre.2017.07.005.

14 J. Mulder and R. Dekker. Methods for strategic liner shipping network design. *European Journal of Operational Research*, 235:367–377, June 2014. doi:10.1016/j.ejor.2013.09.041.

15 S. Ozcan, D. Eliiyi, and L. Reinhardt. Cargo allocation and vessel scheduling on liner shipping with synchronization of transshipments. *Applied Mathematical Modelling*, 77, July 2019. doi:10.1016/j.apm.2019.06.033.

16 L. B. Reinhardt and D. Pisinger. A branch and cut algorithm for the container shipping network design problem. *Flexible Services and Manufacturing Journal*, 24(3):349–374, 2012. doi:10.1007/s10696-011-9105-4.

17 F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

18 A. Santini, S. Ropke, and L. M. Hvattum. A comparison of acceptance criteria for the adaptive large neighbourhood search metaheuristic. *J. Heuristics*, 24(5):783–815, 2018. doi:10.1007/S10732-018-9377-X.

19 P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *CP*, volume 1520, pages 417–431, 1998. doi:10.1007/3-540-49481-2_30.

20 K. Thun, H. Andersson, and M. Christiansen. Analyzing complex service structures in liner shipping network design. *Flexible Services and Manufacturing Journal*, 29(3-4):535–552, 2017. doi:10.1007/s10696-016-9262-6.

21 P. Van Hentenryck and J.-P. Carillon. Generality versus Specificity: An Experience with AI and OR Techniques. In *Proceedings of the National Conference on Artificial Intelligence*, pages 660–664, 1988.

22 S. Wang and Q. Meng. Liner shipping network design with deadlines. *Computers & Operations Research*, 41:140–149, 2014. doi:10.1016/j.cor.2013.08.014.

23 S. Wang, X. Qu, T. Wang, and W. Yi. Optimal container routing in liner shipping networks considering repacking 20 ft containers into 40 ft containers. *Journal of Advanced Transportation*, 2017:1–9, January 2017. doi:10.1155/2017/8608032.

An Expansion-Based Approach for Quantified Integer Programming

Michael Hartisch
University of Passau, Germany
FAU Erlangen-Nürnberg, Germany

Leroy Chew
TU Wien, Austria

Abstract

Quantified Integer Programming (QIP) bridges multiple domains by extending Quantified Boolean Formulas (QBF) to incorporate general integer variables and linear constraints while also generalizing Integer Programming through variable quantification. As a special case of Quantified Constraint Satisfaction Problems (QCSP), QIP provides a versatile framework for addressing complex decision-making scenarios. Additionally, the inclusion of a linear objective function enables QIP to effectively model multistage robust discrete linear optimization problems, making it a powerful tool for tackling uncertainty in optimization.

While two primary solution paradigms exist for QBF – search-based and expansion-based approaches – only search-based methods have been explored for QIP and QCSP. We introduce an expansion-based approach for QIP using Counterexample-Guided Abstraction Refinement (CEGAR), adapting techniques from QBF. We extend this methodology to tackle multistage robust discrete optimization problems with linear constraints and further embed it in an optimization framework, enhancing its applicability. Our experimental results highlight the advantages of this approach, demonstrating superior performance over existing search-based solvers for QIP in specific instances. Furthermore, the ability to model problems using linear constraints enables notable performance gains over state-of-the-art expansion-based solvers for QBF.

2012 ACM Subject Classification Mathematics of computing → Discrete mathematics; Mathematics of computing → Solvers; Theory of computation → Discrete optimization; Theory of computation → Constraint and logic programming; Theory of computation → Algorithm design techniques; Applied computing → Operations research; Theory of computation → Abstraction

Keywords and phrases Quantified Integer Programming, Quantified Constraint Satisfaction, Robust Discrete Optimization, Expansion, CEGAR

Digital Object Identifier 10.4230/LIPIcs.CP.2025.12

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2506.04452`

Supplementary Material
Software (Source Code): `https://github.com/MichaelHartisch/EQuIPS` [36]
archived at `swh:1:dir:29f0d2d80f69841007fd3a315a47c2865669f3a6`

Funding Furthermore, we acknowledge that this research was partially funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 534441421 and Austrian Science Fund FWF Project ESP197.
Michael Hartisch: Partially funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 534441421.
Leroy Chew: funded by Austrian Science Fund FWF Project ESP197.

Acknowledgements We thank the three anonymous reviewers for their helpful comments.

© Michael Hartisch and Leroy Chew;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

1.1 Motivation

Solving Quantified Boolean Formulas (QBF) typically relies on two complementary approaches: Quantified Conflict-Driven Clause Learning (QCDCL) and expansion-based solving. QCDCL extends classical SAT solving techniques by incorporating learning and backjumping in a search-based framework – explicitly traversing the assignment tree respecting the quantifier order – making it particularly effective for QBF instances with deep quantifier alternations. In contrast, expansion-based solving employs a controlled form of Shannon expansion, duplicating the formula to eliminate quantifiers, at the cost of a larger formula with more variables. While expansion methods excel on formulas with few quantifier alternations, they struggle as the number of alternations increases. QCDCL solvers exhibit the opposite behavior: they handle deeply nested quantifications well but can be inefficient for problems where expansion-based approaches are advantageous.

Quantified Integer Programming generalizes QBF by introducing integer variables and linear constraints, placing it within the broader domain of Quantified Constraint Satisfaction Problems (QCSP). Existing QIP solvers follow a QCDCL-inspired approach. However, no expansion-based QIP solver exists, leaving a gap in the landscape of QIP solution methods. In this work, we address this gap by introducing and investigating a new expansion-based QIP solver, enabling efficient resolution of problems where search-based methods struggle.

Expansion-based methods are refutationally complete: given that QIP requires bounded variable domains, a full expansion reduces satisfiability to a finite Integer linear Programming (IP) problem. However, this naive approach is computationally prohibitive, as the expansion tree grows exponentially with the number of variables. In practice, proving unsatisfiability does not require full expansion – an unsatisfiable core often involves only a fraction of the search space. Inspired by techniques from QBF solving, such as those employed in `RAReQS` and `QFUN`, we leverage Counterexample Guided Abstraction Refinement (CEGAR) to construct a more targeted expansion. Our method iteratively refines the search space by alternating between satisfiability and unsatisfiability checks, incorporating counterexamples (countermoves) to guide the exploration process efficiently.

By bridging the gap between expansion-based solving and QIP, our approach broadens the applicability of QIP solvers and introduces new avenues for tackling complex quantified constraint problems.

1.2 Related Work

The study of quantified integer variables dates back at least to the 1990s [27], with the term QIP being coined in [62]. Early research focused on the complexity of QIP [62, 19, 53] and treated QIP as a satisfiability problem until optimization aspects were incorporated [47].

Building on these foundations, a search-based approach for solving QIP was proposed [23, 40], followed by several algorithmic enhancements, including expansion-related relaxation techniques [35] and pruning heuristics [39]. Extensions enabled restricting universal variable assignments, confining them beyond their default bounds [37, 38], similar to robust optimization under polyhedral uncertainty [31], budgeted uncertainty [7, 32], or decision-dependent uncertainty [56, 55]. Furthermore, the link to multistage robust discrete optimization has been established, demonstrating practical applicability [30].

In multistage robust discrete optimization, approaches such as scenario generation [31] and row-and-column generation (e.g. [1]) iteratively refine relaxed formulations, intrinsically incorporating optimization. Unlike these, in order to optimize, we repeatedly solve QIP

feasibility problems in a binary search environment to determine the optimal objective value. For general multistage optimization approaches, most available methods rely on approximation techniques, typically using either sampling methods [5, 48] or decision rules [64, 57, 6], which restrict the solution space to strategies with a predefined (often linear) structure.

Our approach builds upon the expansion-based methods developed for QBF [42, 43, 44], contrasting with traditional search-based approaches [66, 45], though efforts have been made to combine both ideas [12]. Theoretical [9] and practical [46] results suggest that expansion-based solving is advantageous when there are few quantifier alternations and several limitations of search-based QBF methods have been documented [10, 8, 14, 20].

In QCSP, numerous search-based algorithms exist [49, 26, 65, 54, 15], with only a few approaches employing ideas related to expansion – one repair-based [61] and one relaxation-based [25] approach. Early attempts to also incorporate optimization date back to the early 2000s [16]. These efforts primarily relied on search-based methods [4] and were later extended to more general problem formulations [51]. Similar developments occurred in QBF, where optimization versions have also been explored [41].

Finally, several extensions of satisfiability modulo theories (SMT) have been developed to address quantified reasoning [3, 13, 58, 2]. These techniques differ from our QIP-specific approach, which leverages the linear structures for greater efficiency. A recent approach in [63] seems related, given the very similar title, but differs in scope, focusing on semi-infinite problems – optimization problems with finitely many (existentially quantified) variables and infinitely many (quantified) constraints.

2 Preliminaries

2.1 Quantified Integer Programming

A Quantified Integer Program is a natural extension of an integer linear program in which each variable is subject to either existential or universal quantification. Consider n integer variables $x_1, x_2, \dots, x_n$ arranged in order such that if $i < i'$, variable x_i, is said to lie to the left of $x_{i'}$. Each variable x_i, $i \in [n]$[1], takes values in a bounded domain $D_i = \{x_i \in \mathbb{Z} \mid l_i \leq x_i \leq u_i\}$, where l_i and u_i are the lower and upper integer bounds, respectively. In addition, every variable x_i is associated with a quantifier $Q_i \in \{\exists, \forall\}$. The *quantification level* of a variable is defined as the number of alternations of quantifiers to its left plus one. If there are $k \leq n$ such levels, then all variables sharing the same quantification level are grouped together. For each such level j, the common quantifier $\mathcal{Q}_j \in \{\exists, \forall\}$ applies to the vector of variables $\boldsymbol{X}_j$, which collectively range over the domain $\mathcal{D}_j$ (the Cartesian product of their individual domains). We sometimes omit the level index and simply write $\mathcal{D}$, which implies that the variables must remain integral and adhere to their prescribed bounds.

A *QIP feasibility problem* can be written in the compact form $\mathcal{Q}_1\boldsymbol{X}_1 \in \mathcal{D}_1\ \mathcal{Q}_2\boldsymbol{X}_2 \in \mathcal{D}_2\ \cdots\ \mathcal{Q}_k\boldsymbol{X}_k \in \mathcal{D}_k : A^\exists \boldsymbol{x} \leq \boldsymbol{b}^\exists$, where the *existential constraint system* is given by matrix $A^\exists \in \mathbb{Q}^{m \times n}$ and right-hand side vector $\boldsymbol{b}^\exists \in \mathbb{Q}^m$, with $\boldsymbol{x} = (\boldsymbol{X}_1, \dots, \boldsymbol{X}_k)$. Note, that throughout the paper we use bold letters, to indicate vectors. We sometimes write $\mathcal{Q}\boldsymbol{X} \in \mathcal{D} : \Phi$, decomposing the problem into the problem of finding an assignment for the first level variables $\boldsymbol{X}$ and the remaining QIP Φ that either starts with quantifier $\bar{Q}$ or only consists of a constraint system. For QIP Φ and variables $\boldsymbol{X}$ of quantification level $j \in [k]$,

[1] We use $[n] = \{1, \dots, n\}$ to denote index sets.

we write $\Phi[\boldsymbol{\tau}]$ to denote the modified QIP obtained by removing $\boldsymbol{X}$ from the quantification sequence and assigning $\boldsymbol{X}$ to $\boldsymbol{\tau} \in \mathcal{D}_j$ in the constraint system of Φ leading to a change in the right-hand side vector. Then, the new constraint system is described by $A^{\exists}_{(-\boldsymbol{X})}\boldsymbol{x}_{(-\boldsymbol{X})} \leq \boldsymbol{b}^{\exists}_{\boldsymbol{X}=\boldsymbol{\tau}}$, where $A^{\exists}_{(-\boldsymbol{X})}$ contains all columns as $A^{\exists}$ without the columns corresponding to $\boldsymbol{X}$, $\boldsymbol{x}_{(-\boldsymbol{X})}$ corresponds to the variable vector $\boldsymbol{x}$ without variables $\boldsymbol{X}$, and $\boldsymbol{b}^{\exists}_{\boldsymbol{X}=\boldsymbol{\tau}} = \boldsymbol{b}^{\exists} - A^{\exists}_{(\boldsymbol{X})}\boldsymbol{\tau}$, where $A^{\exists}_{(\boldsymbol{X})}$ is the matrix comprising of the columns of $A^{\exists}$ corresponding to $\boldsymbol{X}$.

2.2 QIP Game Semantics

We can think of a QIP as a game between the *universal* ($\forall$) and *existential* ($\exists$) player, where in move j, player $\mathcal{Q}_j$ assigns variables $\boldsymbol{X}_j$ of level j with values from domain $\mathcal{D}_j$. The existential player wins, if the existential constraint system is satisfied after all variables have been assigned. The universal player wins, if at least one constraint is violated. More formally, a *play* is an assignment of all variables $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_k$. A strategy for the assignment of variables $\boldsymbol{X}_j$ of level j, is a function $s_j : \mathcal{D}_1 \times \cdots \times \mathcal{D}_{j-1} \to \mathcal{D}_j$. A strategy $S = (s_i)_{i\in[k],\mathcal{Q}_i=Q}$ for player Q consists of a strategy for all variables associated to her. For a given QIP, S is a *winning strategy* for the existential player, if

$$\forall \boldsymbol{X}_i \in \mathcal{D}_i, i \in [k], Q_i = \forall : \; A^{\exists}\boldsymbol{x} \leq \boldsymbol{b}^{\exists} \bigwedge_{\substack{i\in[k]:\\ \mathcal{Q}_i=\exists}} \boldsymbol{X}_i = s_i(\boldsymbol{X}_1, \ldots, \boldsymbol{X}_{i-1})$$

is true, i.e., for every assignment of the universally quantified variables, the strategy results in a satisfied constraint system. Conversely, S is a winning strategy for the universal player, if

$$\forall \boldsymbol{X}_i \in \mathcal{D}_i, i \in [k], Q_i = \exists : \; A^{\exists}\boldsymbol{x} \not\leq \boldsymbol{b}^{\exists} \bigwedge_{\substack{i\in[k]:\\ \mathcal{Q}_i=\forall}} \boldsymbol{X}_i = s_i(\boldsymbol{X}_1, \ldots, \boldsymbol{X}_{i-1})$$

is true, i.e., if regardless of the assignment of the existentially quantified variables, the strategy of the universal player leads to a violation of the constraint system.

For QIP $\mathcal{Q}\boldsymbol{X} \in \mathcal{D} : \Phi$, a strategy s for assigning $\boldsymbol{X}$ (which is just an assignment $\boldsymbol{\tau} \in \mathcal{D}$), is called a *winning move*, if there exists a winning strategy S containing s. Furthermore, for $\mathcal{Q}\boldsymbol{X}_1 \in \mathcal{D}_1 \bar{\mathcal{Q}}\boldsymbol{X}_2 \in \mathcal{D}_2 : \Phi$ and assignment $\boldsymbol{\tau} \in \mathcal{D}_1$ to $\boldsymbol{X}_1$, the assignment $\boldsymbol{\mu} \in \mathcal{D}_2$ is called a *countermove* to $\boldsymbol{\tau}$, if $\boldsymbol{\mu}$ is a winning move for the QIP $\bar{\mathcal{Q}}\boldsymbol{X}_2 \in \mathcal{D}_2 : \; \Phi[\boldsymbol{\tau}]$. We also adopt the notion of a multi-game as used in [44]. A QIP *multi-game* is given by $\mathcal{Q}\boldsymbol{X} \in \mathcal{D} : \{\Phi_1 \ldots \Phi_\ell\}$, where each Φ_i, referred to as a *subgame*, is a QIP beginning with $\bar{Q}$ or that solely contains a constraint system. A multi-game is won by player $\mathcal{Q}$, if there exists a move $\boldsymbol{\tau} \in \mathcal{D}$ such that $\mathcal{Q}$ has a winning strategy for all subgames $\Phi_i[\boldsymbol{\tau}]$.

2.3 Additional Constraints

To further enhance the modeling power of a QIP, we explicitly enable the restriction of universally quantified variables. This has been introduced in [37] for QIP and has a similar notion as QCSP$^+$ [65] and Quantified Linear Implication [24]. In robust optimization, this corresponds to introducing an uncertainty set [31], while in the context of QBF it is related to including cubes in the initial formula. To this end, let $A^{\forall}\boldsymbol{x} \leq \boldsymbol{b}^{\forall}$ be the *universal constraint system* with $A^{\forall} \in \mathbb{Q}^{\bar{m}\times n}$ and $\boldsymbol{b}^{\forall} \in \mathbb{Q}^{\bar{m}}$, $\bar{m} \in \mathbb{Z}_{\geq 0}$. To ensure that the universal player's constraints remain independent of the existential variables, we require that $A^{\forall}_{\ell,i} = 0$ for every $i \in [n]$ with $Q_i = \exists$ and for all $\ell \in [\bar{m}]$. We also assume that $\{\boldsymbol{x} \in \mathcal{D} \mid A^{\forall}\boldsymbol{x} \leq \boldsymbol{b}^{\forall}\} \neq \emptyset$, i.e., the

"*uncertainty set*" is nonempty. In presence of a universal constraint system we speak of a *QIP with polyhedral uncertainty*. To this end, we redefine the domains of universally quantified variables, by making them depended on assignments $\tilde{\boldsymbol{X}}_1, \dots, \tilde{\boldsymbol{X}}_{j-1}$ of the preceding levels. Specifically, for $j \in [k]$ with $\mathcal{Q}_j = \forall$, we define (with a slight abuse of notation)

$$\mathcal{D}_j(\tilde{\boldsymbol{X}}_1, \dots, \tilde{\boldsymbol{X}}_{j-1}) = \left\{ \boldsymbol{Y} \in \mathcal{D}_j \;\middle|\; \begin{array}{l} \exists\, \boldsymbol{X}_{j+1} \in \mathcal{D}_{j+1}, \dots, \boldsymbol{X}_k \in \mathcal{D}_k \text{ such that } A^{\forall}\boldsymbol{x} \leq \boldsymbol{b}^{\forall}, \\ \text{with } \boldsymbol{x} = (\tilde{\boldsymbol{X}}_1, \dots, \tilde{\boldsymbol{X}}_{j-1}, \boldsymbol{Y}, \boldsymbol{X}_{j+1}, \dots, \boldsymbol{X}_k) \end{array} \right\},$$

while for existentially quantified variables, the domain remains unchanged. Note that, due to the structure of the universal constraint system, only universal variables from preceding levels can influence this domain. This definition guarantees that when the universal player selects an assignment $\boldsymbol{Y}$ at level j, it obeys the lower and upper bounds and there exists an extension in later levels that results in the satisfaction of the universal constraint system. This way, no play can result in a violation of the universal constraint system. If $\bar{m} = 0$, i.e., if there are no universal constraints, using the domain described above, boils down to the bounded domain and we obtain a standard QIP. Hence, for the remainder of the paper, whenever we write $\mathcal{D}$ for a universal domain, it may also be subject to a universal constraint system and we omit stating the dependence on variables of previous levels.

We also allow for a linear objective function $\boldsymbol{c}^\top \boldsymbol{x}$, $\boldsymbol{c} \in \mathbb{Z}^n$, which changes the nature of the game for the existential player, who now has to both satisfy the constraint system and minimize the objective value. Suppose $\mathcal{Q}_1 = \exists$ and $\mathcal{Q}_k = \forall$. Then, the *QIP optimization problem* can be stated as

$$\min_{\boldsymbol{X}_1 \in \mathcal{D}_1} \max_{\boldsymbol{X}_2 \in \mathcal{D}_2} \cdots \min_{\boldsymbol{X}_{k-1} \in \mathcal{D}_{k-1}} \max_{\boldsymbol{X}_k \in \mathcal{D}_k} \boldsymbol{c}^\top \boldsymbol{x} \quad : \quad A^{\exists}\boldsymbol{x} \leq \boldsymbol{b}^{\exists}.$$

In particular, this optimization problem is feasible with optimal objective value $z^\star \in \mathbb{Z}$, if and only if the following holds: the QIP feasibility problem $\mathcal{Q}_1 \boldsymbol{X}_1 \in \mathcal{D}_1\ \mathcal{Q}_2 \boldsymbol{X}_2 \in \mathcal{D}_2\ \cdots\ \mathcal{Q}_k \boldsymbol{X}_k \in \mathcal{D}_k : \ A^{\exists}\boldsymbol{x} \leq \boldsymbol{b}^{\exists} \wedge \boldsymbol{c}^\top \boldsymbol{x} \leq z^\star$ can be won by the existential player, while for the QIP $\mathcal{Q}_1 \boldsymbol{X}_1 \in \mathcal{D}_1\ \mathcal{Q}_2 \boldsymbol{X}_2 \in \mathcal{D}_2\ \cdots\ \mathcal{Q}_k \boldsymbol{X}_k \in \mathcal{D}_k : \ A^{\exists}\boldsymbol{x} \leq \boldsymbol{b}^{\exists} \wedge \boldsymbol{c}^\top \boldsymbol{x} \leq z^\star - 1$ no winning strategy for the existential player exists. The notion of a winning strategy remains the same and we only add the term *optimal winning move* for the existential player, which is the winning move that attains the best worst-case objective.

▶ **Example 1.** Let us consider an example with $n = 5$ variables, with domains $D_1 = \{0, 1, 2\}$ and $D_i = \{0, 1\}$, for $i \in \{2, 3, 4, 5\}$. The quantification sequence (without the bounding domains for sake of presentation) is given by $\exists x_1 \forall x_2 \exists x_3 \forall x_4 \exists x_5$ and we have a universal constraint system consisting of a single constraint $x_2 + x_4 \leq 1$. Hence, $\mathcal{D}_2 = \{0, 1\}$, $\mathcal{D}_4(x_2 = 0) = \{0, 1\}$, and $\mathcal{D}_4(x_2 = 1) = \{0\}$. Let the QIP optimization problem be given by

$$\min_{x_1 \in \mathcal{D}_1} \max_{x_2 \in \mathcal{D}_2} \min_{x_3 \in \mathcal{D}_3} \max_{x_4 \in \mathcal{D}_4} \min_{x_5 \in \mathcal{D}_5} -x_1 + 2x_2 - 3x_3 + x_4 + 2x_5 : \begin{array}{rrrrrl} 2x_1 & & +x_3 & & -x_5 & \leq 4 \\ x_1 & -x_2 & +x_3 & & -x_5 & = 1 \\ & +x_2 & +x_3 & -x_4 & -x_5 & \leq 2 \\ x_1 & +x_2 & +x_3 & +x_4 & & \leq 3. \end{array}$$

Here, $x_1 = 0$ is not a winning move, as there is a countermove $x_2 = 1$ which renders the second constraint violated. There are two winning strategies $S = (s_1, s_3, s_5)$ and $T = (t_1, t_3, t_5)$ for the existential player: $s_1 = 1$, $s_3 = 1$, and $s_5 = 1 - x_2$ as well as $t_1 = 2$, $t_3 = 0$, and $t_5 = 1 - x_2$. Note, that if the universal constraint was not present, neither of them would be a winning strategy, as the fourth existential constraint is always violated in case of universal strategy $x_2 = x_4 = 1$.

The worst-case objective value of T is equal to 1, stemming from the worst case scenario $x_2 = 0$, $x_4 = 1$. The same universal assignment also defines the worst-case for S, resulting in an objective value of -1. Hence, S is a better strategy than T and in fact $x_1 = 1$ is the optimal winning move. To further clarify the used notation, $A^{\exists}_{(-x_1)}\boldsymbol{x}_{(-x_1)} \leq \boldsymbol{b}^{\exists}_{x_1=1}$ is

$$\begin{array}{cccccc} & +x_3 & & -x_5 & \leq 2 \\ -x_2 & +x_3 & & -x_5 & = 0 \\ +x_2 & +x_3 & -x_4 & +x_5 & \leq 2 \\ +x_2 & +x_3 & +x_4 & & \leq 2, \end{array}$$

which is the constraint system of the remaining QIP $\Phi[\tau]$ after making the optimal winning move $\tau = x_1 = 1$.

3 Expansion-Based Quantified Integer Programming Solver

3.1 The Framework

In this section we present our novel solution approach – the *Expansion-based Quantified Integer Programming Solver* (`EQuIPS`) – that is able to solve QIP problems with polyhedral uncertainty. `EQuIPS` is based on an algorithm that iteratively solves abstractions until convergence is achieved. The idea is essentially the same as in the solvers `RAReQS` and `QFUN` and our pseudo-code is based on the one shown in [44]. We adapt their QBF approach to general integer domains and linear functions by accounting for semantic and technical differences, introducing an expansion rule tailored to QIP, and defining a refinement step specific to QIP. The solution process starts with an empty abstraction of the full QIP – a trivial problem where only compliance with variable domains (possibly including universal constraints) have to be followed – and recursively refines the abstraction by adding found countermoves. This way, it partially expands the inner quantifiers until it is sufficient to solve the original QIP. The pseudocode in Algorithm 1 shows the main function that calls itself in a nested fashion. We refer to Appendix A.1 for a formal proof on correctness and insights regarding the underlying proof system.

Algorithm 1 `EQuIPS`$(Q\boldsymbol{X} \in \mathcal{D} : \{\Phi_1 \dots \Phi_\ell\})$ – Find Winning Move for Multi-Game.

Input: multi-game $\mathcal{Q}\boldsymbol{X} \in \mathcal{D} : \{\Phi_1 \dots \Phi_\ell\}$
Output: assignment of $\boldsymbol{X}$ that wins the multi-game or $\bot$ if no winning move exists

1: **if** each Φ_l is quantifier free **then return** `wins1`$(\mathcal{Q}\boldsymbol{X} \in \mathcal{D} : \{\Phi_1 \dots \Phi_\ell\})$
2: $\alpha \leftarrow \mathcal{Q}\boldsymbol{X} \in \mathcal{D} : \varnothing$ ▷ start with empty abstraction
3: **while** True **do**
4: $\quad\boldsymbol{\tau}' \leftarrow$ `EQuIPS`(α) ▷ find winning move for abstraction
5: $\quad$**if** $\boldsymbol{\tau}' = \bot$ **then return** $\bot$ ▷ no move for abstraction ⇒ no move for multi-game
6: $\quad\boldsymbol{\tau} \leftarrow$ `extract`$(\boldsymbol{\tau}', \boldsymbol{X})$
7: $\quad\lambda \leftarrow -1$
8: $\quad$**for** $l \in [\ell]$ **do**
9: $\quad\quad\boldsymbol{\mu} \leftarrow$ `EQuIPS` $(\Phi_l[\boldsymbol{\tau}])$ ▷ find countermove to $\boldsymbol{\tau}$
10: $\quad\quad$**if** $\boldsymbol{\mu} \neq \bot$ **then** $\lambda \leftarrow l$
11: $\quad$**if** $\lambda = -1$ **then return** $\boldsymbol{\tau}$ **else** $\alpha \leftarrow$`refine` $(\alpha, \Phi_l, \boldsymbol{\mu})$ ▷ refine abstraction

The initial input of Algorithm 1 is the QIP $\mathcal{Q}\boldsymbol{X} \in \mathcal{D} : \Phi$, i.e., a multi-game with a single subgame. In Line 1, we deal with the case where each subgame of the multi-game is quantifier free. It is noteworthy, that the `wins1` function call significantly differs from the

one used in [44]. The main reason is that in case of QBF and $\mathcal{Q} = \forall$, a winning move for the universal player can be found by solving a SAT problem consisting of the conjunction of the negated Boolean formulas of each subgame. Recall, that the goal for the existential player is to identify a move that ensures each constraint system of every subgame to be violated. In the case of QIP, it is not immediately clear what the counterpart of a negated Boolean formula would be for a system of linear constraints. We discuss the `wins1` function in more detail in Section 3.2.

In Line 2 of Algorithm 1 we initialize the abstraction, containing no subgames. A move that wins the current abstraction is found in (Line 4). If α is the empty abstraction, this call to `EQuIPS` will immediately invoke `wins1`, returning any assignment that satisfies the domain $\mathcal{D}$. When $\mathcal{Q} = \exists$, any assignment from the bounded domain may be returned. However, when $\mathcal{Q} = \forall$, compliance with the universal constraint system must also be ensured.

A technical detail to note is that the abstraction may include copies of later-stage variables (due to the subsequent `refine` call), necessitating the extraction of only those assignments corresponding to the variables of interest, $\boldsymbol{X}$, in Line 6. After obtaining the corresponding move $\boldsymbol{\tau}$, in case a countermove $\boldsymbol{\mu}$ exists, the abstraction is refined by adding the subgame that corresponds to $\boldsymbol{\mu}$ (Lines 8–11). We will provide more insights into the `refine` function in Section 3.3. If we are not able to find a move that wins the abstraction, we know by construction, that there cannot exist a winning move for the initial multi-game, and return $\perp$ in Line 5. Similarly, if no countermove to $\boldsymbol{\tau}$ can be found, this means that $\boldsymbol{\tau}$ is not only a winning move for the abstraction but also for the entire multi-game. In this case, we return $\boldsymbol{\tau}$ (see Line 11).

3.2 `wins1` on Integer Linear Programs

One crucial aspect of our algorithm is the call `wins1`$(\mathcal{Q}\boldsymbol{X} \in \mathcal{D} : \{\Phi_1 \dots \Phi_\ell\})$, where all subgames $\Phi_1 \dots \Phi_\ell$ are quantifier free, i.e., they each only contain a constraint system with variables $\boldsymbol{x} = \boldsymbol{X}$. Simply speaking, this call tries to answer the question, whether player $\mathcal{Q}$ can find an assignment of $\boldsymbol{X}$, which is a winning move for all subgames $\Phi_1, \dots, \Phi_\ell$. The notation of `wins1` is borrowed from `QFUN`, and in `QFUN`, it is more or less a call to a SAT solver. Our subroutine of `wins1`, however requires some non trivial modification. In the case of a QBF, this problem can be stated in a straight-forward manner as if $\mathcal{Q} = \exists$ one has to find an assignment satisfying the conjunction of the ℓ copies of the Boolean function, while if $\mathcal{Q} = \forall$ a solution to the conjunction of all negated Boolean formulas must be found. But in particular regarding the latter case, there is no counterpart in integer programming: there is no notion of a "negated constraint system".

For $l \in [\ell]$, let $A_l^\exists \boldsymbol{x} \leq \boldsymbol{b}_l^\exists$ be the constraint system corresponding to subgame Φ_l. Recall, that if $\mathcal{Q} = \exists$, a winning move ensures that the constraint system of each subgame is satisfied. Hence, in order to find a winning move, we need to find a solution to Problem (1):

$$A_l^\exists \boldsymbol{x} \leq \boldsymbol{b}_l^\exists \quad \forall l \in [\ell] \tag{1a}$$

$$\boldsymbol{x} \in \mathcal{D} \tag{1b}$$

This is a standard integer program and can be solved using any standard solver.

If $\mathcal{Q} = \forall$, we need to find an assignment of $\boldsymbol{x} \in \mathcal{D}$, such that all constraint systems are violated, while obeying the own domain, i.e., the uncertainty set. Violating a constraint system means that at least one of its constraints is not satisfied. To the end of modeling this as an integer linear program, let $\boldsymbol{L}^l \in \mathbb{Q}^m$ be a vector for each $l \in [\ell]$, where

$$L_j^l \leq \min_{\boldsymbol{x}\in\mathcal{D}} (A_l^\exists)_{j,\star}\boldsymbol{x} = \sum_{\substack{i\in[n]\\(A_l^\exists)_{j,i}<0}} (A_l^\exists)_{j,i}u_i + \sum_{\substack{i\in[n]\\(A_l^\exists)_{j,i}\geq 0}} (A_l^\exists)_{j,i}l_i$$

for every $j \in [m]$. In other words, the j-th entry of $\boldsymbol{L}^l$ provides a lower bound that is not larger than the minimum possible value of the left-hand side of row j in constraint system l. As a result, the inequality $\boldsymbol{L}^l \leq A_l^\exists \boldsymbol{x}$ is trivially fulfilled for any $\boldsymbol{x} \in \mathcal{D}$.

These lower bounds only need to be computed once at the start of our solver since the constraint system remains (basically) the same across all subgames, only differing in the values of already assigned variables. One also could refine these bounds dynamically for each subgame by considering already assigned variables, aiming to accelerate the IP solution process through potentially improved relaxations. However, this approach comes at the cost of additional computational effort, as the bounds must be recomputed at each invocation of `wins1`. In our implementation, we opted for the weaker bounds that only need to be calculated once.

Furthermore, we need to be able to certify a violation of a constraint. To this end, we need the following lemma.

▶ **Lemma 2.** *Given a linear constraint $\sum_{i\in[n]} a_i x_i \leq b$ with rational coefficients $a_i, b \in \mathbb{Q}$ and integer variables x_i. Then, for any integer assignment $\tilde{\boldsymbol{x}}$, it holds that $\sum_{i\in[n]} a_i\tilde{x}_i \not\leq b \Leftrightarrow \sum_{i\in[n]} a_i\tilde{x}_i \geq b + r$, where $r = \frac{1}{d}$, for $d = lcd\{a_1, \ldots, a_n, b\}$, being the reciprocal of the lowest common denominators of the a_i and b.*

Proof. Let d be the lowest common denominator of the $n+1$ parameters. Then the constraint can be rewritten as $\sum_{i\in[n]} \frac{\tilde{a}_i}{d} x_i \leq \frac{\tilde{b}}{d}$, with integers $\tilde{a}_i$ and $\tilde{b}$. For any assignment $\tilde{\boldsymbol{x}}$ with $\sum_{i\in[n]} a_i\tilde{x}_i > b$ the gap between the right-hand side and the left-hand side can be stated as $\left|\frac{\tilde{b}-\sum_{i\in[n]}\tilde{a}_i x_i}{d}\right| > 0$, where the numerator is integer. Hence, a lower bound for this gap is attained if the numerator is equal to one, i.e., $\frac{1}{d}$ is a lower bound on the violation of the constraint, which proves the claim. ◀

This lemma shows the need for integrality of universally quantified variables in our approach, as otherwise, we would not be able to specify a value of minimal violation, as a continuous variable could violate the right-hand side by an arbitrarily small value. For existentially quantified variables this in principle is not necessary in our current setting. Now, let $\boldsymbol{r}^l \in \mathbb{Q}^m$ be a vector with positive entries less than or equal to the reciprocals of the lowest common denominators of the rows of $A_l^\exists$ and $\boldsymbol{b}_l^\exists$, ensuring for any row $j \in [m]$ and any $\boldsymbol{x} \in \mathcal{D}$ that $(A_l^\exists)_{j,\star}\boldsymbol{x} \not\leq (\boldsymbol{b}_l^\exists)_k \Leftrightarrow (A_l^\exists)_{j,\star}\boldsymbol{x} \geq (\boldsymbol{b}_l^\exists)_k + r_j^l$. Note that the lowest common denominator of the $n+1$ rational numbers can be computed in polynomial time, using the Euclidean algorithm, assuming that the denominators of the coefficients are know. But also note that smaller values to bound the gap between right-hand side and left-hand side are allowed. E.g. for the constraint $0.5x_1 + 2x_2 <= 4$ the designated value would be 0.5, as the lowest common denominator of 0.5, 2 and 4 is 2 with a reciprocal of 0.5. On the other hand, finding the number with the most decimal places also is valid. Let p be this number. Then setting $r_j^l = 10^{-p}$ also suffices. In the above example this equates to setting $r_j^l = 0.1$. The latter case is what we implemented. Further note, that that if all coefficients are integers, this value always can be set to one.

Now, consider the following integer linear program (2):

$$-A_l^{\exists}\boldsymbol{x} - (\boldsymbol{L}^l - \boldsymbol{b}_l^{\exists} - \boldsymbol{r}^l)\boldsymbol{y}^l \leq -\boldsymbol{L}^l \qquad \forall l \in [\ell] \tag{2a}$$

$$v_l \leq \sum_{j\in[m]} y_j^l \qquad \forall l \in [\ell] \tag{2b}$$

$$v \leq v_l \qquad \forall l \in [\ell] \tag{2c}$$

$$v \geq 1 \tag{2d}$$

$$A^{\forall}\boldsymbol{x} \leq \boldsymbol{b}^{\forall} \tag{2e}$$

$$\boldsymbol{x} \in \mathcal{D} \tag{2f}$$

$$v, v_1, \ldots, v_\ell \in \{0,1\} \tag{2g}$$

$$\boldsymbol{y}^l \in \{0,1\}^m \qquad \forall l \in [\ell] \tag{2h}$$

The idea is that the indicator variable v only can be set to 1, if all ℓ systems are violated by an assignment of $\boldsymbol{x}$. We can immediately see, that in order to set v to 1, all v_l must be set to 1. Hence, for any $l \in [\ell]$ there has to exist at least one constraint $j \in [m]$ for which $y_j^l = 1$. So let us consider the setting of y_j^l. Setting $y_j^l = 0$ can never result in a violation of the respective constraint, as $A_l^{\exists}\boldsymbol{x} \geq \boldsymbol{L}^l$ is always true. On the other hand, it is only feasible to set y_j^l to 1, if $(A_l^{\exists})_{k,*}\boldsymbol{x} \geq (b_l^{\exists})_j + r_j^l$ holds, which is only the case, if $\boldsymbol{x}$ violates constraint j of system l. Consequently, v_l is an indicator whether constraint system l is violated. Only if we find some $\boldsymbol{x}$ that violates all constraint systems while at the same time obeys the uncertainty set given by $A^{\forall}\boldsymbol{x} \leq \boldsymbol{b}^{\forall}$, we can also set $v = 1$. In other words: If and only if there exists an assignment of $\boldsymbol{x} \in \mathcal{D}$ with $A_l^{\exists}\boldsymbol{x} \not\leq \boldsymbol{b}_l^{\exists}$ for all $l \in [\ell]$, Problem (2) has a feasible solution.

▶ **Example 3.** Consider the constraint $0.5x_1 + 2x_2 <= 4$ and variable bounds $-2 \leq x_1 \leq 2$ and $0 \leq x_2 \leq 3$. The lower bound L for the left-hand side is -1. We set $r = 0.5$ to indicate the minimal violation of this constraint. Consequently, the corresponding Constraint (2a) is given by $-0.5x_1 - 2x_2 + 5.5y \leq 1$ and in particular, in case of assigning $y = 1$, $0.5x_1 + 2x_2 \geq 4.5$ must be true, indicating the violation of the original constraint.

The pseudocode of the `wins1` function is presented in Algorithm 2. In our implementation we utilize the solver `GUROBI` [33] to solve Problems (1) and (2).

Algorithm 2 `wins1`$(\mathcal{Q}\boldsymbol{X} \in \mathcal{D} : \{\Phi_1 \ldots \Phi_\ell\})$ – Solve Multi-Game with a Single Move.

Input: multi-game $\mathcal{Q}\boldsymbol{X} \in \mathcal{D} : \{\Phi_1 \ldots \Phi_\ell\}$ with all Φ_l quantifier free
Output: assignment of $\boldsymbol{X}$ that wins the multi-game or $\perp$ if no winning move exists
1: **if** $\mathcal{Q} = \exists$ **then** $\pi \leftarrow$ Problem (1) **else** $\pi \leftarrow$ Problem (2)
2: **if** π is feasible **then return** solution on π **else return** $\perp$

3.3 Refinement by Expansion

It can be argued that the unique aspect of `RAReQS` and `QFUN` that sets it apart from other CEGAR approaches is that they extend the number of variables being looked at in the abstraction via expansion. This is much easier to spot in the original `RAReQS` than in `QFUN` (our description in Algorithm 1 is based off `QFUN`). In `QFUN` and our description this is achieved by having multiple listed subproblems after the outer quantifier block, as these subgames technically each have separate variables. The function `refine` adds an additional subgame to the abstraction, based on the countermove $\boldsymbol{\mu}$, that was found to beat our move $\boldsymbol{\tau}$ in one of the original subgames.

Given an abstraction $\alpha = \mathcal{Q}\boldsymbol{X} \in \mathcal{D}_{\boldsymbol{X}} : \{\Psi_1, \ldots, \Psi_n\}$, with subgames Ψ_i, $i \in [n]$, for which a winning move $\boldsymbol{\tau}$ was found. Let Φ be another subgame, in which $\boldsymbol{\tau}$ is not a winning move, i.e., there exists a countermove $\boldsymbol{\mu}$ that wins $\Phi[\boldsymbol{\tau}]$. Let the quantification sequence of Φ start with $\bar{\mathcal{Q}}\boldsymbol{Y}$. If $\Phi = \bar{\mathcal{Q}}\boldsymbol{Y} \in \mathcal{D}_{\boldsymbol{Y}} : A^{\exists}\boldsymbol{x} \leq \boldsymbol{b}^{\exists}$, then

$$\texttt{refine}(\alpha, \Phi, \boldsymbol{\mu}) = \mathcal{Q}\boldsymbol{X} \in D_{\boldsymbol{X}} : \left\{\Psi_1, \ldots, \Psi_n, A^{\exists}_{(-\boldsymbol{Y})}\boldsymbol{x}_{(-\boldsymbol{Y})} \leq \boldsymbol{b}^{\exists}_{\boldsymbol{Y}=\boldsymbol{\mu}}\right\},$$

i.e., the refined abstraction contains an additional constraint system accounting for the scenario of the found countermove. If $\Phi = \bar{\mathcal{Q}}\boldsymbol{Y} \in \mathcal{D}_{\boldsymbol{Y}}\mathcal{Q}\boldsymbol{Z} \in \mathcal{D}_{\boldsymbol{Z}} : \Lambda$ for a QIP Λ, then

$$\texttt{refine}(\alpha, \Phi, \boldsymbol{\mu}) = \mathcal{Q}\boldsymbol{X} \in \mathcal{D}_{\boldsymbol{X}}\boldsymbol{Z}_{\Phi}^{(\boldsymbol{Y}=\boldsymbol{\mu})} \in \mathcal{D}_{\boldsymbol{Z}} : \left\{\Psi_1, \ldots, \Psi_n, \Lambda_{\Phi}^{(\boldsymbol{Y}=\boldsymbol{\mu})}[\boldsymbol{\mu}]\right\},$$

where $\boldsymbol{Z}_{\Phi}^{(\boldsymbol{Y}=\boldsymbol{\mu})}$ is a copy of $\boldsymbol{Z}$ that represents the move of $\boldsymbol{Z}$ in case $\boldsymbol{Y}$ is set to $\boldsymbol{\mu}$, and the variables of $\Lambda_{\Phi}^{(\boldsymbol{Y}=\boldsymbol{\mu})}$ are copies of the variables of Λ having the same annotation as $\boldsymbol{Z}_{\Phi}^{(\boldsymbol{Y}=\boldsymbol{\mu})}$. When $\boldsymbol{Z}$ is universal, we also need to make sure it satisfies the uncertainty set and thus $\mathcal{D}_{\boldsymbol{Z}}$ is meant to include the respective constraints on the annotated variables. We sometimes write $\boldsymbol{Z}^{(\boldsymbol{\mu})}$ instead of $\boldsymbol{Z}^{(\boldsymbol{Y}=\boldsymbol{\mu})}$.

▶ **Example 4.** Consider the QIP $\exists x_1 x_2 \forall z_1 z_2 \exists t d : (x_1 + x_2 + t - 2d = 0) \wedge (z_1 + z_2 + t \geq 1) \wedge (-z_1 - z_2 - t \geq -2)$ with all binary domains. Consider the outer level. Initially α, the abstraction, will be empty, which means that any binary assignment of the variables x_1 and x_2 is feasible. If we choose $\boldsymbol{\tau} = (0, 0)$, the universal response is to assign $z_1 = z_2 = 0$, at which point the universal player wins. Hence, a countermove $\boldsymbol{\mu} = (0, 0)$ to $\boldsymbol{\tau}$ is found and consequently the abstraction α is refined to become $\alpha = \exists x_1 x_2 t^{(00)} d^{(00)} : (x_1 + x_2 + t^{(00)} - 2d^{(00)} = 0) \wedge (t^{(00)} \geq 1) \wedge (-t^{(00)} \geq -2)$. The `EQuIPS` call on this refined abstraction will end up in a call of the `wins1` function, as after the initial existential quantifier, no further quantifiers follow. A solution to the respective constraint system is $\boldsymbol{\tau}' = (0, 1, 1, 1)$, which contains assignments of x_1, x_2, $t^{(00)}$, and $d^{(00)}$. Extracting the values of the relevant variables x_1 and x_2 we obtain $\boldsymbol{\tau} = (0, 1)$. We again check whether we find a countermove to $\boldsymbol{\tau}$, in which case $z_1 = z_2 = 1$ is produced. We once again adapt the abstraction by calling

$$\texttt{refine}(\alpha, \forall z_1 z_2 \exists t d : (x_1 + x_2 + t - 2d = 0) \wedge (z_1 + z_2 + t \geq 1) \wedge (-z_1 - z_2 - t \geq -2), (1, 1)),$$

yielding the refined abstraction

$$\exists x_1 x_2 t^{(00)} d^{(00)} t^{(11)} d^{(11)} : \left\{\begin{array}{r}\Big((x_1 + x_2 + t^{(00)} - 2d^{(00)} = 0) \wedge (t^{(00)} \geq 1) \wedge (-t^{(00)} \geq -2)\Big), \\ \Big((x_1 + x_2 + t^{(11)} - 2d^{(11)} = 0) \wedge (t^{(11)} \geq -1) \wedge (-t^{(11)} \geq 0)\Big)\end{array}\right\},$$

with two subgames. As each subgames is quantifier free, another call to `wins1` is invoked and the IP solver is called on the constraint system

$$\begin{array}{lll} x_1 + x_2 + t^{(00)} - 2d^{(00)} = 0 & t^{(00)} \geq 1 & -t^{(00)} \geq -2 \\ x_1 + x_2 + t^{(11)} - 2d^{(11)} = 0 & t^{(11)} \geq -1 & -t^{(11)} \geq 0 \\ x_1, x_2, t^{(00)}, d^{(00)}, t^{(11)}, d^{(11)} \in \{0, 1\}. & & \end{array}$$

As this IP is infeasible, we know that there is no move for (x_1, x_2) that wins the abstraction, and hence, there cannot exist a move for (x_1, x_2) that wins the game.

3.4 Optimization

3.4.1 Optimization Method 1: Binary Search

In Section 2, we introduced the QIP optimization problem, for which a search-based solution approach exists [40]. We now want to utilize the presented expansion-based approach, to obtain another solution tool for the optimization problem. To this end, we assume that a linear objective is given with objective coefficients $\boldsymbol{c} \in \mathbb{Z}^n$. For clarity of presentation, we assume an existential starting player in this case. We have already drawn the connection between the QIP optimization problem and its decision version, where the objective function is moved to the constraints and one asks for the existence of a solution with objective value less than or equal to some value z. As all variables are bounded and the objective value only attains integer values, we can compute lower and upper bounds on the objective value. In particular, for the optimal objective value we know $z^\star \in [\min_{\boldsymbol{x}\in\mathcal{D}} \boldsymbol{c}^\top \boldsymbol{x}, \max_{\boldsymbol{x}\in\mathcal{D}} \boldsymbol{c}^\top \boldsymbol{x}]$. Now, we can conduct a binary search to close in on the optimal value as shown in Algorithm 3.

Algorithm 3 Optimization Framework utilizing `EQuIPS`.

Input: QIP optimization problem
Output: $\perp$, if instance is infeasible and otherwise the optimal objective value.

$LB \leftarrow \min_{\boldsymbol{x}\in\mathcal{D}} \boldsymbol{c}^\top \boldsymbol{x}$
$UB \leftarrow \max_{\boldsymbol{x}\in\mathcal{D}} \boldsymbol{c}^\top \boldsymbol{x}$
$z \leftarrow UB$
run `EQuIPS` on QIP decision problem with additional constraint $\boldsymbol{c}^\top \boldsymbol{x} \leq z$
if $\perp$ **then return** *"Instance is infeasible"*
while $UB - LB > 0$ **do**
    $z \leftarrow (LB + UB)/2$
    run `EQuIPS` on QIP decision problem with additional constraint $\boldsymbol{c}^\top \boldsymbol{x} \leq z$
    if feasible **then** $UB \leftarrow z$ **else** $LB \leftarrow z + 1$
return UB

3.4.2 Optimization Method 2: Mixing Methods

Repeatedly running the solver in a binary search can end up being more expensive than necessary. Our working hypothesis is that the existing search-based solver `Yasol` [40] is good at finding solutions, but slow at verifying optimality, while `EQuIPS` suffers from the opposite problem: it is much better at showing inconsistencies from objective value bounds that are too tight, but slow at finding good initial solutions. Therefore we propose to combine the approaches, by letting `Yasol` search for good solutions and use `EQuIPS` to verify optimality. In particular, every time `Yasol` finds a new solution – an improved value for UB – we can call `EQuIPS` and try to verify that no solution with objective value at most $UB - 1$ exists by adding the respective constraint on the objective function. Then, if and only if, UB is optimal, no winning move for the existential player will be found. Doing this sequentially, i.e., stopping the search process of `Yasol` while `EQuIPS` tries to verify optimality, of course can be detrimental, as in the early phase of the optimization process, `Yasol` tends to find better solutions quickly. Thus, calling `EQuIPS` for every newly found solution can become inefficient. We instead think of initiating a parallel process of `EQuIPS` while `Yasol` continues its search. If `EQuIPS` certifies that no better solution exists, `Yasol` can terminate early. Conversely, if `Yasol` finds a new solution, the current `EQuIPS` process can be terminated and restarted with the updated bound.

It is noteworthy, that if EQuIPS returns a winning move with an objective function bounded by z, it cannot be concluded that a solution with objective value z exists, but only that a solution with some value less than or equal to z exists. Hence, Yasol cannot extract a newly found solution from a call to EQuIPS that does not verify optimality.

In this combined approach we also see options to exploit further synergies, where not only Yasol profits from EQuIPS, but also the other way around. When Yasol continues the search process after finding a new incumbent solution, it would incorporate all information it has gathered during the solution process. Running EQuIPS with a new bound on the objective function, on the other hand, is essentially starting again from scratch, which we want to avoid. Since constraints are learned during the search process of Yasol, adding them to EQuIPS is one way to transfer learned information from one solver to the other. Any learned constraint holds information on what variable assignments will not lead to a (improved) solution. It is worth mentioning that Yasol does not explicitly track all its progress through learning constraints, but also implicitly through branching decisions. But each branch of the search tree that is completed without a found (better) solution can be interpreted as a learned constraint. Adding such constraints to the verification instance, has the potential to improve the runtime of EQuIPS, with the risk of increasing the size of the instance too much, making it harder for the underlying IP solver to solve Problems (1) and (2). This issue has similarities to the selection of cutting planes for solving integer programs (see, e.g., [22]) and further research needs to be done for our special case.

▶ **Theorem 5.** *Let $\Pi\phi$ be a QIP with Π a quantifier prefix and ϕ an IP and assume* Yasol *is a correct clause learning algorithm for QIP and also complete, in that it will eventually find the optimal solution. Suppose* Yasol *learns clauses $C_1 \dots C_n$ on the way to learning a solution with objective function value of v. Then $\Pi\phi \wedge F$ is false if and only if $\Pi\phi \wedge C_1 \dots C_n \wedge D \wedge F$ is false, where F is a constraint saying the objective function is strictly less than v.*

Proof. Suppose Yasol has learned clauses $C_1 \dots C_n$ and a solution with value v. Let us consider the QIP feasibility problem $\Pi\phi \wedge C_1 \dots C_n \wedge F$. This QIP is either true, resulting in the existence of a strategy with value less than v, or there is no such strategy, rendering the QIP false. If $\Pi\phi \wedge C_1 \dots C_n \wedge F$ is true, then obviously $\Pi\phi \wedge F$ also must be true, as it contains less existential constraints. Now assume $\Pi\phi \wedge C_1 \dots C_n \wedge F$ is false. Assume $\Pi\phi \wedge F$ is true, which means that there exists a solution with objective value strictly less than v. As Yasol is complete, it eventually has to find this solution. However, $\Pi\phi \wedge C_1 \dots C_n \wedge F$ being false, implies that Yasol can no longer find the solution if it restarted after learning $C_1 \dots C_n$ (which is an option for Yasol to perform after clause learning). Consequently, $\Pi\phi \wedge F$ also must be false. ◀

4 A New Challenging Problem Class: QRandomParity

QRandomParity is a combination of QParity, which are known to be hard for QCDCL based QBF solvers [10], and RandomParity which are hard for CDCL based SAT solvers [17].

Given an integer n and a random permutation σ on $[n]$. Consider the Quantified Boolean problem

$$\begin{aligned}\exists x_1 \dots x_n \forall z \exists t_2 \dots t_n \exists s_2 \dots s_n.\ & t_2 = (x_1 \oplus x_2) \wedge \dots \wedge t_i = (t_{i-1} \oplus x_i), \dots \wedge \\ & s_2 = (x_{\sigma(1)} \oplus x_{\sigma(2)}) \wedge \dots \wedge s_i = (s_{i-1} \oplus x_{\sigma(i)}), \dots \wedge \\ & (z \to \neg t_n) \wedge (\neg z \to s_n)\end{aligned}$$

Both t_n and s_n compute the parity of the x variables but use a different ordering. In particular, $t_n = s_n$ must be fulfilled. If we take the full expansion we get a contradiction, because parity is associative and commutative. These families have been shown hard for CDCL solvers like `CaDiCaL` in both theory and experiments [17]. This is because of the standard encodings of the parity problems into clauses. Typically one encodes $a = (b \oplus c)$ using four clauses $(\neg a \vee b \vee c)$, $(a \vee \neg b \vee c)$, $(a \vee b \vee \neg c)$, and $(\neg a \vee \neg b \vee \neg c)$.

Note that, extension variables can be used to produce formulas easy for resolution. In this case it has been shown [18] that only $O(n \log n)$ many extension variables are needed before we can get linear size resolution refutation. In pseudo-Boolean constraints the extension variables come more naturally into a standard encoding of parity, but require a new variable e for each constraint. Therefore $a = (b \oplus c)$ can be represented by the constraint $a + b + c = 2e$.

This encoding allows a short cutting planes proof. We simply can add all constraints to get $x_1 + \cdots + x_n + t_n + 2t_2 \cdots + 2t_{n-1} - 2e_2 \cdots - 2e_n = 0$ and this can be repeated for $x_1 + \cdots + x_n + s_n + 2s_2 \cdots + 2s_{n-1} - 2f_2 \cdots - 2f_n = 0$.

Subtracting one from the other we get $t_n - s_n + 2\sum_{i=2}^{n}(t_i - e_i - (s_i - f_i)) = 0$. The only integer solution to this equality is to have $t_n = s_n$, as otherwise an odd number would be on the left-hand side of the constraint, and cutting planes finds this via the division rule [29]. The idea is, that IP solvers, which have access to several preprocessing techniques that utilize constraint aggregation (e.g., see [28, 50, 52]), may be able to handle this as well.

The QRandomParity problem in clausal form (as QBF) is given by the quantification sequence $\exists x_1 \dots x_n \forall u \exists t_2 \dots t_n, s_1 \dots s_n$ (with all Boolean variables) and matrix

$$(\bar{x}_1 \vee x_2 \vee t_2)\ (x_1 \vee \bar{x}_2 \vee t_2)\ (x_1 \vee x_2 \vee \bar{t}_2)\ (\bar{x}_1 \vee \bar{x}_2 \vee \bar{t}_2) \tag{3a}$$

$$(\bar{x}_i \vee t_{i-1} \vee t_i)\ (x_i \vee \bar{t}_{i-1} \vee t_i)\ (x_i \vee t_{i-1} \vee \bar{t}_i)\ (\bar{x}_i \vee \bar{t}_{i-1} \vee \bar{t}_i) \text{ for } i = 3 \text{ to } n \tag{3b}$$

$$(\bar{x}_{\sigma(1)} \vee x_{\sigma(2)} \vee s_2)\ (x_{\sigma(1)} \vee \bar{x}_{\sigma(2)} \vee s_2)\ (x_{\sigma(1)} \vee x_{\sigma(2)} \vee \bar{s}_2)\ (\bar{x}_{\sigma(1)} \vee \bar{x}_{\sigma(2)} \vee \bar{s}_2) \tag{3c}$$

$$(\bar{x}_{\sigma(i)} \vee s_{i-1} \vee s_i)\ (x_{\sigma(i)} \vee \bar{s}_{i-1} \vee s_i)\ (x_{\sigma(i)} \vee s_{i-1} \vee \bar{s}_i)\ (\bar{x}_{\sigma(i)} \vee \bar{s}_{i-1} \vee \bar{s}_i) \text{ for } i = 3 \text{ to } n \tag{3d}$$

$$(\bar{u} \vee \bar{t}_n)\ (u \vee s_n), \tag{3e}$$

where each (3a)-(3d) encodes an XOR relation. In linear form (as QIP), all variables are binary and their sequence is given by $\exists x_1 \dots x_n \forall u \exists t_2 \dots t_n, d_2 \dots d_n, s_1 \dots s_n, e_2 \dots e_n$, where a linear encoding of the XOR relation is used, needing auxiliary variables d and e:

$$x_1 + x_2 + t_2 = 2d_2 \qquad x_{\sigma(1)} + x_{\sigma(2)} + s_2 = 2e_2 \tag{4a}$$

$$t_{i-1} + x_i + t_i = 2d_i \qquad s_{i-1} + x_{\sigma(i)} + s_i = 2e_i \qquad \text{for } i \in \{3, \dots, n\} \tag{4b}$$

$$-u - t_n \geq -1 \qquad u + s_n \geq 1. \tag{4c}$$

5 Experimental Evaluation

For our experiments, we compiled `EQuIPS` with `GUROBI` 12.0 and `Yasol`, which utilizes a linear programming solver, is compiled with `CPLEX` 22.1. When installing `QFUN` with the respective SAT solvers, we used the provided script and we used `DepQBF` 6.01 and the latest version of `z3`.

5.1 QRandomParity

As argued in Section 4 we expect our expansion-based solver to perform well on instances of type QRandomParity, compared to search-based algorithms like `Yasol` [40] and `DepQBF` [45]. We also tested the solver `z3` [21], which is capable of handling such instances. Furthermore, we are interested in the comparison of `EQuIPS` against state-of-the-art expansion-based solvers

from the QBF community such as QFUN [44]. QFUN allows the integration of several SAT solvers, and we compiled it once with CaDiCaL [11] and once with CryptoMiniSat [60], which we refer to as $\texttt{QFUN}_{\text{cadi}}$ and $\texttt{QFUN}_{\text{cms}}$, respectively. Here we expect $\texttt{QFUN}_{\text{cms}}$ to perform better on QRandomParity instances, due to the better handling of XOR clauses by CryptoMiniSat compared to other SAT solvers. For Yasol and EQuIPS, we use the QIP Encoding (4), while for the other solvers we use QBF Encoding (3). Notably, we tested both encodings with z3 and observed that it performed better on the QBF formulation than on the QIP formulation. Consequently, we report only its performance on Encoding (3).

For each of the following experiments, we created 100 instances for each n (only varying in the random permutation of the variables). Experiments were conducted on an AMD Ryzen 9 5900X processor (3.70 GHz) with 128 GB RAM, imposing a 1800 seconds time limit per instance and restricting each process to a single thread. In a first experiment, for $n \in \{10, 12, \ldots, 26\}$ we compare all solvers and show the results in a Cactus plot in Figure 1.

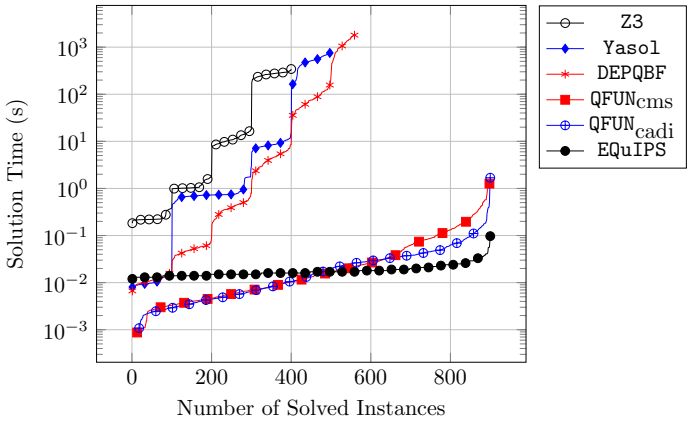


Figure 1 Cactus plot. Algorithms that appear further to the right and closer to the bottom solve more instances faster, indicating better performance.

As expected, we can observe the expansion-based solvers outclass the search-based solvers DepQBF and Yasol as well as z3. $\texttt{QFUN}_{\text{cadi}}$ and $\texttt{QFUN}_{\text{cms}}$ basically have the same performance, while our approach has the most constant behavior, outperforming all solvers for $n \geq 22$.

For even larger values of n, $\texttt{QFUN}_{\text{cadi}}$ quickly reaches its limit, incapable of solving these instances before timeout, while $\texttt{QFUN}_{\text{cms}}$ still can solve such instances easily. In particular, for $n = 100$, no instance was solved by $\texttt{QFUN}_{\text{cadi}}$ before the timeout. This is likely due to the special handling of XOR formulations by CryptoMiniSat, that CaDiCaL lacks. Hence, for larger values of n we restrict our comparison to $\texttt{QFUN}_{\text{cms}}$ and EQuIPS, as shown in Table 1.

Table 1 Median runtimes and (if existent) number of not solved QRandomParity instances.

n	100	200	300	400	500	600	700	800	900	1000
EQuIPS	0.06	0.18	0.37	0.68	0.99	1.31	1.72	2.13	2.64	3.44/10
$\texttt{QFUN}_{\text{cms}}$	1.67	2.21	2.09	2.18	2.37	2.13/1	2.51/2	2.22/3	3.04/4	-/100

For an increasing value of n, our approach consistently outperforms $\texttt{QFUN}_{\text{cms}}$. However, for very large n, both solvers sometimes fail to return a solution within the time limit. This is quite surprising, as runtime trends did not indicate a sharp increase in solution times. We further investigated this by analyzing the behavior of GUROBI on the fully expanded

problem for large values of n. It became evident that for smaller instances, GUROBI could solve them entirely during preprocessing, without initiating a search. However, for larger instances, preprocessing terminated prematurely before infeasibility was detected, forcing the solver into a search phase from which it never returned. For some instances, we were able to fine tune GUROBI parameters, but not to the extend of solving instances with $n \geq 1500$. We suspect a similar phenomenon occurs with CryptoMiniSat. These observations align with our hypothesis that aggregation techniques efficiently prove infeasibility for smaller instances, but as the problem size grows, identifying the right constraints for aggregation becomes increasingly difficult. This, in turn, leads GUROBI to halt preprocessing prematurely and initiate an exhaustive search instead.

For all experiments, QFUN and DepQBF were given instances in the QDIMACS file format. Surprisingly, testing the QCIR format – where we expected $\texttt{QFUN}_{\text{cms}}$ to perform even better – resulted in worse performance. Additionally, we evaluated EQuIPS on the QBF formulation of QRandomParity, where all constraints are clauses (see Encoding (3) in the appendix). While EQuIPS could still solve instances of size 100 within seconds, it failed to solve any instance of size 200. This further supports our claim that leveraging the modeling capabilities of linear constraints can be advantageous.

5.2 Multilevel Critical Node Problem

To evaluate our approach on optimization instances, we consider the multilevel critical node problem (MCN) as introduced in [1]. Details on the problem and the QIP encoding can be found in Appendix A.2. We compare the performance of Yasol, EQuIPS (utilizing binary search), and the baseline column- and row-generation approach ($\texttt{MCN}_{CR}$) from [1]. Note that $\texttt{MCN}_{CR}$ is essentially a scenario generation approach that uses dualization techniques to approximate the optimal solution of the adversary problem. In contrast, the approach provided by Yasol and EQuIPS is much more straightforward and easy to use, requiring only a problem encoding as QIP. Notably, we do not compare our solver to the enhanced techniques from [1], as our goal is to demonstrate the model-and-run potential of QIPs as a baseline for such instances.

The $\texttt{MCN}_{CR}$ algorithm was executed using Python 2.7.18, with mixed-integer linear programs solved via IBM CPLEX 12.9. Experiments were conducted on an AMD EPYC 9474F 48-Core Processor (3.60 GHz) with 256 GB RAM, imposing a two-hour time limit per instance and restricting each process to a single thread. We used the provided instances[2], consisting of randomly generated undirected graphs with $|V| \in \{20, 40, 60, 80, 100\}$, a density of 5%, and various budget limit configurations (Ω, Φ, and Δ). Figure 2 presents a Cactus plot, revealing key performance trends. Our approach exhibits higher run times on instances that are generally solved quickly. However, EQuIPS performs well on larger instances with inherently higher run times. Overall, our method solves the second-highest number of instances (465), compared to 431 solved by Yasol and 475 by $\texttt{MCN}_{CR}$.

5.3 Further Experiments

To further evaluate our expansion-based approach, we conducted additional experiments (see Appendix A.3 for details). The key findings are as follows. First, using EQuIPS to verify the optimality of incumbent solutions found during the search process of Yasol is promising.

[2] Instances, optimal solutions, and algorithms from [1] were provided at https://github.com/mxmmargarida/Critical-Node-Problem.

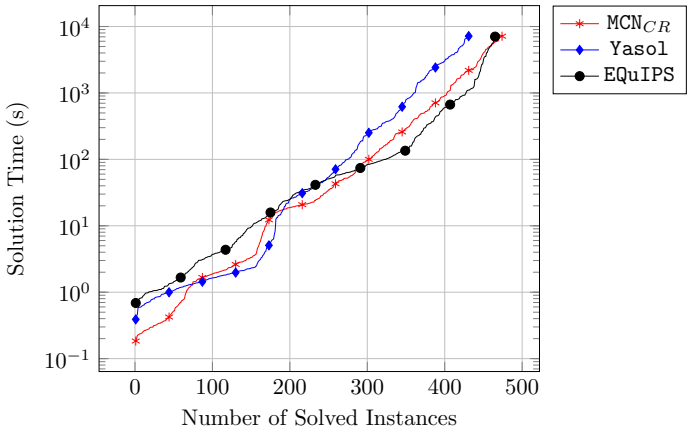


Figure 2 Cactus plot for experiments on MCN test set.

This hybrid approach can be enhanced by incorporating selected learned constraints from `Yasol` into the `EQuIPS` verification instance. Second, `EQuIPS` does not always outperform `Yasol`– especially on instances with multiple quantifier alternations – a result that aligns with similar observations in QBF. Finally, several heuristic improvements remain to be explored. For example, solving the IP relaxation to obtain an initial winning move shows potential but may increase run times when the full expansion is eventually required.

6 Conclusion

In this paper, we presented an expansion-based approach for quantified integer programming, a rarely explored direction in quantified constraint programming. Our method leverages the power of state-of-the-art integer linear programming solvers to handle abstractions – partial expansions of the quantified program.

`EQuIPS` offers advantages over existing QIP solvers like `Yasol` and various QBF approaches. It inherits the benefits of expansion-based solvers over search-based solvers in QBF while also utilizing the modeling flexibility of linear constraints. Additionally, we observe that QBF solvers incorporating XOR reasoning and expansion, such as `QFUN` with `CryptoMiniSat`, perform competitively with `EQuIPS`.

Our experiments show that `EQuIPS` offers advantages in certain cases. We demonstrate performance improvements on different families that other approaches neglect, strengthening the case for combining methods and illustrating how such combinations can be effective in practice.

Future improvements could integrate machine learning techniques to enhance abstraction construction. While `QFUN` employs decision trees, our integer programming setting allows for numerical machine learning methods such as support vector machines. Additionally, the counterexample generation in `wins1` can be heuristically guided by adding an objective to the underlying integer programs, e.g., choosing universal assignments that maximize constraint violation or existential assignments that optimize the objective. To further improve optimization capabilities, strategy extraction as well as directly incorporating optimization aspects into our framework are also promising directions. Furthermore, parallelizing both the exploration of countermoves in each sub-game and the objective-value checks during binary search should improve performance.

References

1 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.

2 Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark Barrett. Extending SMT solvers to higher-order logic. In *Automated Deduction–CADE 27: 27th International Conference on Automated Deduction, Natal, Brazil, August 27–30, 2019, Proceedings 27*, pages 35–54. Springer, 2019. doi:10.1007/978-3-030-29436-6_3.

3 Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. *Handbook of model checking*, pages 305–343, 2018. doi:10.1007/978-3-319-10575-8_11.

4 Marco Benedetti, Arnaud Lallouet, and Jérémie Vautard. Quantified constraint optimization. In *International Conference on Principles and Practice of Constraint Programming*, pages 463–477. Springer, 2008. doi:10.1007/978-3-540-85958-1_31.

5 Dimitris Bertsimas and Iain Dunning. Multistage robust mixed-integer optimization with adaptive partitions. *Operations Research*, 64(4):980–998, 2016. doi:10.1287/opre.2016.1515.

6 Dimitris Bertsimas and Angelos Georghiou. Binary decision rules for multistage adaptive mixed-integer optimization. *Mathematical Programming*, 167:395–433, 2018. doi:10.1007/s10107-017-1135-6.

7 Dimitris Bertsimas and Melvyn Sim. Robust discrete optimization and network flows. *Mathematical programming*, 98(1):49–71, 2003. doi:10.1007/s10107-003-0396-4.

8 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.

9 Olaf Beyersdorff, Leroy Chew, Judith Clymo, and Meena Mahajan. Short proofs in QBF expansion. In *Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22*, pages 19–35. Springer, 2019. doi:10.1007/978-3-030-24258-9_2.

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

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 Nikolaj Bjørner, Mikolás Janota, and William Klieber. On conflicts and strategies in QBF. In *LPAR (short papers)*, pages 28–41, 2015. doi:10.29007/4sk1.

13 Nikolaj S Bjørner and Mikolás Janota. Playing with quantified satisfaction. *LPAR (short papers)*, 35:15–27, 2015. doi:10.29007/vv21.

14 Benjamin Böhm and Olaf Beyersdorff. QCDCL vs QBF resolution: Further insights. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPIcs*, pages 4:1–4:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SAT.2023.4.

15 Hubie Chen. Beyond Q-resolution and prenex form: A proof system for quantified constraint satisfaction. *Logical Methods in Computer Science*, 10, 2014. doi:10.2168/LMCS-10(4:14)2014.

16 Hubie Chen and Martin Pál. Optimization, games, and quantified constraint satisfaction. In *International Symposium on Mathematical Foundations of Computer Science*, pages 239–250. Springer, 2004. doi:10.1007/978-3-540-28629-5_16.

17 Leroy Chew, Alexis de Colnet, Friedrich Slivovsky, and Stefan Szeider. Hardness of random reordered encodings of parity for resolution and CDCL. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(8):7978–7986, March 2024. doi:10.1609/aaai.v38i8.28635.

18 Leroy Chew and Marijn J. H. Heule. Sorting parity encodings by reusing variables. 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 1–10. Springer, Springer, 2020. doi:10.1007/978-3-030-51825-7_1.

19 Dmitry Chistikov and Christoph Haase. On the complexity of quantified integer programming. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 94:1–94:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.94.

20 Abhimanyu Choudhury and Meena Mahajan. Dependency schemes in CDCL-based QBF solving: A proof-theoretic study. *J. Autom. Reason.*, 68(3):16, 2024. doi:10.1007/s10817-024-09707-4.

21 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, Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

22 Santanu S Dey and Marco Molinaro. Theoretical challenges towards cutting-plane selection. *Mathematical Programming*, 170:237–266, 2018. doi:10.1007/s10107-018-1302-4.

23 Thorsten Ederer, Michael Hartisch, Ulf Lorenz, Thomas Opfer, and Jan Wolf. Yasol: an open source solver for quantified mixed integer programs. In *Advances in Computer Games: 15th International Conferences, ACG 2017, Leiden, The Netherlands, July 3–5, 2017, Revised Selected Papers 15*, pages 224–233. Springer, 2017. doi:10.1007/978-3-319-71649-7_19.

24 Pavlos Eirinakis, Salvatore Ruggieri, K Subramani, and Piotr Wojciechowski. On quantified linear implications. *Annals of Mathematics and Artificial Intelligence*, 71(4):301–325, 2014. doi:10.1007/s10472-013-9332-3.

25 Alex Ferguson and Barry O'Sullivan. Relaxations and explanations for quantified constraint satisfaction problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 690–694. Springer, 2006. doi:10.1007/11889205_52.

26 Ian P Gent, Peter Nightingale, Andrew Rowley, and Kostas Stergiou. Solving quantified constraint satisfaction problems. *Artificial Intelligence*, 172(6-7):738–771, 2008. doi:10.1016/j.artint.2007.11.003.

27 Richard Gerber, William Pugh, and Manas Saksena. Parametric dispatching of hard real-time tasks. *IEEE transactions on computers*, 44(3):471–479, 1995. doi:10.1109/12.372041.

28 Fred Glover. Surrogate constraints. *Operations Research*, 16(4):741–749, 1968. doi:10.1287/opre.16.4.741.

29 Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(5):3768–3777, 2021. doi:10.1609/aaai.v35i5.16494.

30 Marc Goerigk and Michael Hartisch. Multistage robust discrete optimization via quantified integer programming. *Computers & Operations Research*, 135:105434, 2021. doi:10.1016/j.cor.2021.105434.

31 Marc Goerigk and Michael Hartisch. An introduction to robust combinatorial optimization. *International Series in Operations Research and Management Science*, 2024. doi:10.1007/978-3-031-61261-9.

32 Marc Goerigk, Jannis Kurtz, and Michael Poss. Min–max–min robustness for combinatorial problems with discrete budgeted uncertainty. *Discrete Applied Mathematics*, 285:707–725, 2020. doi:10.1016/j.dam.2020.07.011.

33 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024. URL: https://www.gurobi.com.

34 Michael Hartisch. *Quantified integer programming with polyhedral and decision-dependent uncertainty*. PhD thesis, University of Siegen, Germany, 2020. doi:10.25819/ubsi/4841.

35 Michael Hartisch. Adaptive relaxations for multistage robust optimization. In *Pacific Rim International Conference on Artificial Intelligence*, pages 485–499. Springer, 2021. doi:10.1007/978-3-030-89188-6_36.

36 Michael Hartisch and Leroy Chew. MichaelHartisch/EQuIPS. Software, DFG-534441421, FWF Project ESP197, swhId: swh:1:dir:29f0d2d80f69841007fd3a315a47c2865669f3a6 (visited on 2025-07-23). URL: https://github.com/MichaelHartisch/EQuIPS, doi:10.4230/artifacts.24095.

37 Michael Hartisch, Thorsten Ederer, Ulf Lorenz, and Jan Wolf. Quantified integer programs with polyhedral uncertainty set. In *Computers and Games: 9th International Conference, CG 2016, Leiden, The Netherlands, June 29–July 1, 2016, Revised Selected Papers 9*, pages 156–166. Springer, 2016. doi:10.1007/978-3-319-50935-8_15.

38 Michael Hartisch and Ulf Lorenz. Mastering uncertainty: towards robust multistage optimization with decision dependent uncertainty. In *PRICAI 2019: Trends in Artificial Intelligence: 16th Pacific Rim International Conference on Artificial Intelligence, Cuvu, Yanuca Island, Fiji, August 26–30, 2019, Proceedings, Part I 16*, pages 446–458. Springer, 2019. doi:10.1007/978-3-030-29908-8_36.

39 Michael Hartisch and Ulf Lorenz. A novel application for game tree search-exploiting pruning mechanisms for quantified integer programs. In *Advances in Computer Games: 16th International Conference, ACG 2019, Macao, China, August 11–13, 2019, Revised Selected Papers 16*, pages 66–78. Springer, 2020. doi:10.1007/978-3-030-65883-0_6.

40 Michael Hartisch and Ulf Lorenz. A general model-and-run solver for multistage robust discrete linear optimization. *arXiv preprint arXiv:2210.11132*, 2022. doi:10.48550/arXiv.2210.11132.

41 Alexey Ignatiev, Mikoláš Janota, and Joao Marques-Silva. Quantified maximum satisfiability. *Constraints*, 21:277–302, 2016. doi:10.1007/s10601-015-9195-9.

42 Mikoláš Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. In Alessandro Cimatti and Roberto Sebastiani, editors, *Proc. 15th International Conference on Theory and Applications of Satisfiability Testing*, volume 7317, pages 114–128. Springer, 2012. doi:10.1007/978-3-642-31612-8_10.

43 Mikoláš 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 Mikoláš Janota. Towards generalization in QBF solving via machine learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1):6607–6614, April 2018. doi:10.1609/aaai.v32i1.12208.

45 Florian Lonsing and Uwe Egly. Depqbf 6.0: A search-based QBF solver beyond traditional QCDCL. In *Automated Deduction–CADE 26: 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6–11, 2017, Proceedings*, pages 371–384. Springer, 2017. doi:10.1007/978-3-319-63046-5_23.

46 Florian Lonsing and Uwe Egly. Evaluating QBF solvers: Quantifier alternations matter. In John Hooker, editor, *Principles and Practice of Constraint Programming*, pages 276–294, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-98334-9_19.

47 Ulf Lorenz and Jan Wolf. Solving multistage quantified linear optimization problems with the alpha–beta nested benders decomposition. *EURO Journal on Computational Optimization*, 3:349–370, 2015. doi:10.1007/s13675-015-0038-7.

48 Francesca Maggioni, Fabrizio Dabbene, and Georg Ch. Pflug. Sampling methods for multistage robust optimization problems. *Ann. Oper. Res.*, 347(3):1385–1423, 2025. doi:10.1007/s10479-025-06545-4.

49 Nikos Mamoulis and Kostas Stergiou. Algorithms for quantified constraint satisfaction problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 752–756. Springer, 2004. doi:10.1007/978-3-540-30201-8_60.

50 Hugues Marchand and Laurence A Wolsey. Aggregation and mixed integer rounding to solve MIPs. *Operations research*, 49(3):363–371, 2001. doi:10.1287/opre.49.3.363.11211.

51 Toshihiro Matsui, Hiroshi Matsuo, Marius Calin Silaghi, Katsutoshi Hirayama, Makoto Yokoo, and Satomi Baba. A quantified distributed constraint optimization problem. In *Proc. 9th Int'l. Conf. on Autonomous Agents and Multiagent Systems (AAMAS2010)*, volume 1, pages 1023–1030. Nagoya Institute of Technology, 2010. URL: https://dl.acm.org/citation.cfm?id=1838344.

52 George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Wiley interscience series in discrete mathematics and optimization. Wiley, 1988. doi:10.1002/9781118627372.

53 Danny Nguyen and Igor Pak. The computational complexity of integer programming with alternations. *Mathematics of Operations Research*, 45(1):191–204, 2020. doi:10.1287/moor.2018.0988.

54 Peter Nightingale. Non-binary quantified CSP: algorithms and modelling. *Constraints*, 14:539–581, 2009. doi:10.1007/s10601-009-9068-1.

55 Jérémy Omer, Michael Poss, and Maxime Rougier. Combinatorial robust optimization with decision-dependent information discovery and polyhedral uncertainty. *Open Journal of Mathematical Optimization*, 5:1–25, 2024. doi:10.5802/ojmo.33.

56 Michael Poss. Robust combinatorial optimization with variable cost uncertainty. *European Journal of Operational Research*, 237(3):836–845, 2014. doi:10.1016/j.ejor.2014.02.060.

57 Krzysztof Postek and Dick den Hertog. Multistage adjustable robust mixed-integer optimization via iterative splitting of the uncertainty set. *INFORMS Journal on Computing*, 28(3):553–574, 2016. doi:10.1287/ijoc.2016.0696.

58 Andrew Reynolds, Tim King, and Viktor Kuncak. Solving quantified linear arithmetic by counterexample-guided instantiation. *Formal Methods in System Design*, 51:500–532, 2017. doi:10.1007/s10703-017-0290-y.

59 Alexander Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999.

60 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending sat solvers to cryptographic problems. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 244–257. Springer, 2009. doi:10.1007/978-3-642-02777-2_24.

61 Kostas Stergiou. Repair-based methods for quantified CSPs. In *International Conference on Principles and Practice of Constraint Programming*, pages 652–666. Springer, 2005. doi:10.1007/11564751_48.

62 K Subramani. Analyzing selected quantified integer programs. In *International Joint Conference on Automated Reasoning*, pages 342–356. Springer, 2004. doi:10.1007/978-3-540-25984-8_26.

63 Kerian Thuillier, Anne Siegel, and Loïc Paulevé. CEGAR-based approach for solving combinatorial optimization modulo quantified linear arithmetics problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(8):8146–8153, March 2024. doi:10.1609/aaai.v38i8.28654.

64 Phebe Vayanos, Daniel Kuhn, and Berç Rustem. A constraint sampling approach for multistage robust optimization. *Automatica*, 48(3):459–471, 2012. doi:10.1016/j.automatica.2011.12.002.

65 Guillaume Verger and Christian Bessiere. Guiding search in QCSP+ with back-propagation. In *International Conference on Principles and Practice of Constraint Programming*, pages 175–189. Springer, 2008. doi:10.1007/978-3-540-85958-1_12.

66 Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified Boolean satisfiability solver. In *ICCAD*, pages 442–449, 2002. doi:10.1145/774572.774637.

A Appendix

A.1 Underlying Proof System and Correctness

It has been well established that RAReQS works with the QBF proof system ∀Exp+Res [43]. It comes as no surprise that the underlying proof system of EQuIPS acts much the same, instead of describing the SAT oracle as a resolution system, we describe the IP call as a cutting planes proof. Figure 3 describes the ∀Exp+Cutting Planes proof systems which is the underlying proof system for refutations where the first quantifier is existential.

$$\frac{\sum_{k\in[n]:\ Q_k=\exists} a_k x_k + \sum_{k\in[n]:\ Q_k=\forall} a_k x_k \leq b \text{ in matrix}}{\sum_{k\in[n]:\ Q_k=\exists} a_k x_k^{[\boldsymbol{\tau}]} + \sum_{k\in[n]:\ Q_k=\forall} a_k \boldsymbol{\tau}(x_k) \leq b} \text{ (Axiom)}$$

- $\boldsymbol{\tau}$ is a complete assignment to universal variables that satisfies the universal constraint system
- For $x_k^{[\boldsymbol{\tau}]}$, $[\boldsymbol{\tau}]$ takes only the part of $\boldsymbol{\tau}$ that is left of x_k

- **Addition**: From $\sum_{k\in[n]} a_k x_k \leq b$ and $\sum_{k\in[n]} \alpha_k x_k \leq \beta$, derive $\sum_{k\in[n]} (a_k + \alpha_k) x_k \leq b + \beta$.
- **Multiplication**: From $\sum_{k\in[n]} a_k x_k \leq b$, derive $\sum_{k\in[n]} d a_k x_k \leq db$, where $d \in \mathbb{Z}^+$.
- **Division**: From $\sum_{k\in[n]} a_k x_k \leq b$, derive $\sum_{k\in[n]} \frac{a_k}{d} x_k \leq \left\lceil \frac{b}{d} \right\rceil$, where $d \in \mathbb{Z}^+$ divides each a_k.

Figure 3 The proof system ∀Exp+Cutting Planes.

▶ **Theorem 6.** *∀Exp+Cutting Planes is a sound and complete proof system for QIP.*

Proof. Given a QIP, we take a full Shannon expansion on all universal quantifiers. In case a universal constraint system is present, only universal variable assignments satisfying this system are considered. Every potential axiom line is found as a conjunct in this expansion. The full expansion is satisfiability equivalent with the original QIP and contains no universal quantifiers. Therefore given the completeness and soundness of the Cutting Planes proof system, we prove ∀Exp+Cutting Planes is a sound and complete proof system for QIP. ◀

We will now give an overview on the soundness of EQuIPS. We start with an observation about Algorithm 1 and the description of refinement in Section 3.3.

▶ **Observation 7.** *At any point in a run of* EQuIPS *with outer block $\mathcal{Q}\boldsymbol{X} \in D_{\boldsymbol{X}}$, for every subgame of the abstraction there is an assignment $\boldsymbol{\mu}$ to the first inner block variables $\boldsymbol{Y}$ such that the subgame is of one of two forms:*

1. $\Psi_i = A^{\exists}_{(-\boldsymbol{Y})}\boldsymbol{x}_{(-\boldsymbol{Y})} \leq \boldsymbol{b}^{\exists}_{\boldsymbol{Y}=\boldsymbol{\mu}}$ *when* $\Phi_{l_i} = \bar{\mathcal{Q}}\boldsymbol{Y} \in \mathcal{D}_{\boldsymbol{Y}} : A^{\exists}\boldsymbol{x} \leq \boldsymbol{b}^{\exists}$, *or*
2. $\Psi_i = \Lambda^{(\boldsymbol{Y}=\boldsymbol{\mu})}_{\Phi_{l_i}}[\boldsymbol{\mu}]$ *when* $\Phi_{l_i} = \bar{\mathcal{Q}}\boldsymbol{Y} \in \mathcal{D}_{\boldsymbol{Y}}\mathcal{Q}\boldsymbol{Z} \in \mathcal{D}_{\boldsymbol{Z}} : \Lambda$, *where* Λ *itself is a QIP.*

In the first case we have a QIP with only an outer block, in the second case we have a QIP with at least one inner block, Λ *is a QIP representing the rest of blocks and the constraints.*

▶ **Lemma 8.** *If* EQuIPS *returns* $\bot$ *on multi-game* $\forall\boldsymbol{X} \in \mathcal{D}_{\boldsymbol{X}} : \{\Phi_1 \dots \Phi_n\}$ *then this multi-game is won by the existential player.*

Proof. We prove this by induction on the quantifier depth. The base case is if $\Phi_1 \dots \Phi_n$ are quantifier free, and we solve integer linear program (2) as `wins1` is invoked. Then, $\bot$ is returned if and only if the universal player is not able to violate the constraint systems of all subgames as argued in Section 3.2. Therefore $\forall\boldsymbol{X} \in \mathcal{D}_{\boldsymbol{X}} : \{\Phi_1 \dots \Phi_n\}$ is won by the existential player.

Now suppose some Φ_j contains a quantifier. Then, we build an abstraction α of the multi-game and $\bot$ is only returned, if the call to `EQuIPS` in Line 4 on the abstraction returns $\bot$. Note that the first call to Line 4 on the empty abstraction always returns a move $\boldsymbol{\tau}$, as $\mathcal{D}_{\boldsymbol{X}}$ is always non-empty. Thus, a refinement of the abstraction must have led to a returned $\bot$. Hence, we have to show that a call to `EQuIPS` for a refined abstraction $\forall\boldsymbol{X} \in \mathcal{D}_{\boldsymbol{X}}\forall\boldsymbol{Z}^{(\boldsymbol{Y}=\boldsymbol{\mu}_1)}_{\Phi_{l_1}} \in \mathcal{D}_{\boldsymbol{Z}} \dots \boldsymbol{Z}^{(\boldsymbol{Y}=\boldsymbol{\mu}_k)}_{\Phi_{l_k}} \in \mathcal{D}_{\boldsymbol{Z}} : \{\Psi_1 \dots \Psi_k\}$ returns $\bot$, only if the original formula is won by the existential player. We know that the abstraction is existentially feasible by induction hypothesis, in other words there is no assignment to the outer block that makes all subgames infeasible. We now argue that for any $\boldsymbol{\tau} \in \mathcal{D}_{\boldsymbol{X}}$, there must be one subgame that is won by the existential player. Otherwise for each Φ_j, $j \in [n]$ there exists an assignment $\boldsymbol{\tau}_i \in \mathcal{D}_{\boldsymbol{Z}}$ for which $\Psi_i[\boldsymbol{\tau}][\boldsymbol{\tau}_i]$ is lost for the existential player. Then we can construct $\boldsymbol{\tau}' = (\boldsymbol{\tau}, \boldsymbol{\tau}_{l_1}, \dots, \boldsymbol{\tau}_{l_k})$. This is well defined because $\boldsymbol{X}$ (corresponding to $\boldsymbol{\tau}$) are the only shared outer variables between the subgames. $\boldsymbol{\tau}'$ is also a winning move of the abstraction and in particular, each subgame Ψ_i is won by the universal player, against our assumption. Therefore for each $\boldsymbol{\tau} \in \mathcal{D}_{\boldsymbol{X}}$ there is a subgame Ψ_i that is won by the existential player under the assignment to the remaining outer variables. From Observation 7, subgames Ψ_i can have one of the following structures:

1. $\Psi_i = A^{\exists}_{(-\boldsymbol{Y})}\boldsymbol{x}_{(-\boldsymbol{Y})} \leq \boldsymbol{b}^{\exists}_{\boldsymbol{Y}=\boldsymbol{\mu}}$ when $\Phi_{l_i} = \exists\boldsymbol{Y} \in \mathcal{D}_{\boldsymbol{Y}} : A^{\exists}\boldsymbol{x} \leq \boldsymbol{b}^{\exists}$, or
2. $\Psi_i = \Lambda^{(\boldsymbol{Y}=\boldsymbol{\mu})}_{\Phi_{l_i}}[\boldsymbol{\mu}]$ when $\Phi_{l_i} = \exists\boldsymbol{Y} \in \mathcal{D}_{\boldsymbol{Y}}\forall\boldsymbol{Z} \in \mathcal{D}_{\boldsymbol{Z}} : \Lambda$, where Λ itself is a QIP.

We argue that one of these being feasible means that its associated $\boldsymbol{\mu}$ is countermove to $\boldsymbol{\tau}$ in the original game. If $\boldsymbol{\tau}$ allows Φ_{l_i}, then Φ_{l_i} is in the original games. In the first case, $\Psi_i[\boldsymbol{\tau}] = A^{\exists}_{(-\boldsymbol{X}-\boldsymbol{Y})}\boldsymbol{x}_{(-\boldsymbol{X}-\boldsymbol{Y})} \leq \boldsymbol{b}^{\exists}_{\boldsymbol{X}=\boldsymbol{\tau}\boldsymbol{Y}=\boldsymbol{\mu}}$ is feasible so $\boldsymbol{\mu}$ is a countermove in the original game that satisfies $\Phi_{l_i}[\boldsymbol{\tau}]$. In the second case $\Lambda^{(\boldsymbol{X}=\boldsymbol{\tau},\boldsymbol{Y}=\boldsymbol{\mu})}_{\Phi_{l_i}}[\boldsymbol{\tau}, \boldsymbol{\mu}]$ is feasible so $\boldsymbol{\mu}$ must be a countermove in the original game that satisfies $\Phi_{l_i}[\boldsymbol{\tau}]$. Therefore every $\boldsymbol{\tau}$ has an existential countermove. ◀

In the existential case, we can understand soundness through the theoretical existence of proofs. We unfortunately cannot extract proofs at this stage, as we would need to extract cutting planes proofs from `Gurobi` which is not supported.

▶ **Lemma 9.** *If* EQuIPS *returns* $\bot$ *on multi-game* $\exists\boldsymbol{X} \in \mathcal{D}_{\boldsymbol{X}} : \{\Phi_1 \dots \Phi_n\}$ *then there is a* $\forall$*Exp+Cutting Planes refutation of* $\exists\boldsymbol{X} \in \mathcal{D}_{\boldsymbol{X}}.\Phi_1 \wedge \dots \wedge \Phi_n$.

Proof. We can prove this via induction on the quantifier depth of $\exists \boldsymbol{X} \in \mathcal{D}_{\boldsymbol{X}} : \{\Phi_1 \dots \Phi_n\}$.

For the base case, if all $\Phi_1 \dots \Phi_n$ are quantifier free, `wins1` is invoked. Given the outer quantifier is $\exists$, we solve the integer program (1), which is the conjunction of all constraint systems $\Phi_1 \dots \Phi_n$. Since cutting planes is a complete refutational system for Integer Programming [59] there is a cutting planes proof of this refutation.

Now consider the inductive step. Suppose some Φ_i has a quantifier. Then `EQuIPS` goes into the CEGAR loop and only returns $\perp$ if the abstraction α returns $\perp$. We have to show if the abstraction $\exists \boldsymbol{X} \in \mathcal{D}_{\boldsymbol{X}} \exists \boldsymbol{Z}^{(\boldsymbol{Y}=\boldsymbol{\mu}_1)}_{\Phi_{l_1}} \in \mathcal{D}_{\boldsymbol{Z}} \dots \exists \boldsymbol{Z}^{(Y=\boldsymbol{\mu}_k)}_{\Phi_{l_k}} \in \mathcal{D}_{\boldsymbol{Z}} : \{\Psi_1 \dots \Psi_k\}$ returns $\perp$ then we can construct a $\forall$Exp+Cutting Planes refutation of the original formula.

By assuming the induction hypothesis we have an $\forall$Exp+Cutting Planes refutation π of $\exists \boldsymbol{X} \in \mathcal{D}_{\boldsymbol{X}} \exists \boldsymbol{Z}^{(Y=\boldsymbol{\mu}_1)}_{\Phi_{l_1}} \in \mathcal{D}_{\boldsymbol{Z}} \dots \exists \boldsymbol{Z}^{(Y=\boldsymbol{\mu}_k)}_{\Phi_{l_k}} \in \mathcal{D}_{\boldsymbol{Z}} : \{\Psi_1 \dots \Psi_k\}$. Note that the quantified variables in each $\Psi_1 \dots \Psi_k$ are different. The only variables they share are in the outer block variables $\boldsymbol{X}$.

Consider an individual axiom step C of π, which involves taking a constraint from Ψ_i, and a complete assignment $\boldsymbol{\beta}$ to the universal variables of Ψ_i that satisfies the universal constraint system. Note that Ψ_i can only be a subgame in the abstraction for one of two reasons:

1. $\Psi_i = A^{\exists}_{(-\boldsymbol{Y})} \boldsymbol{x}_{(-\boldsymbol{Y})} \leq \boldsymbol{b}^{\exists}_{\boldsymbol{Y}=\boldsymbol{\mu}}$ when $\Phi_{l_i} = \forall \boldsymbol{Y} \in \mathcal{D}_{\boldsymbol{Y}} : A^{\exists} \boldsymbol{x} \leq \boldsymbol{b}^{\exists}$, when $\boldsymbol{\mu} \in \mathcal{D}_{\boldsymbol{Y}}$.
2. $\Psi_i = \Lambda^{(\boldsymbol{Y}=\boldsymbol{\mu})}_{\Phi_{l_i}}[\boldsymbol{\mu}]$ when $\Phi_{l_i} = \forall \boldsymbol{Y} \in \mathcal{D}_{\boldsymbol{Y}} \exists \boldsymbol{Z} \in \mathcal{D}_{\boldsymbol{Z}} : \Lambda$, when $\boldsymbol{\mu} \in \mathcal{D}_{\boldsymbol{Y}}$.

In each case, for every Ψ_i there is a $\boldsymbol{Y} = \boldsymbol{\mu}$ statement that corresponds to it. We take π and rename all the existential variables appearing in the inner blocks to create π', we rename w appearing in Ψ_i to be $w^{(\boldsymbol{Y}=\boldsymbol{\mu})}_{\Psi_i}$.

In the first case, we can take the single row j from $A^{\exists}_{(-\boldsymbol{Y})} \boldsymbol{x}_{(-\boldsymbol{Y})} \leq \boldsymbol{b}^{\exists}_{\boldsymbol{Y}=\boldsymbol{\mu}}$ that was combined with $\boldsymbol{\beta}$ to get C. Note that $\boldsymbol{\beta}$ must be empty, because there are no universals left. We take the same row j of $A^{\exists} \boldsymbol{x} \leq \boldsymbol{b}^{\exists}$ and combine it with $\boldsymbol{\mu}$ (which satisfies the uncertainty set) in an axiom step from our original formula. Then $A^{\exists}_j \boldsymbol{x} \leq \boldsymbol{b}^{\exists}_j$ instantiates to $(A^{\exists}_{(-\boldsymbol{Y})})_j \boldsymbol{x}_{(-\boldsymbol{Y})} \leq (\boldsymbol{b}^{\exists}_{(\boldsymbol{Y}=\boldsymbol{\mu})})_j$, exactly the same as C and the transformation in π' does not change this, because there are no inner existential variables.

In case 2, the inner part of $\Lambda^{(\boldsymbol{Y}=\boldsymbol{\mu})}_{\Phi_{l_i}}$ must have a row j that appears as constraint

$$\sum_{k \in [n]:\ Q_k = \exists} a^{\exists}_{j,k} x_k + \sum_{k \in [n]:\ Q_k = \forall} a^{\exists}_{j,k} x_k \leq (\boldsymbol{b}^{\exists}_{(\boldsymbol{\mu})})_j$$

that combines with $\boldsymbol{\beta}$ to get axiom

$$\sum_{k \in [n]:\ Q_k = \exists} a^{\exists}_{j,k} x^{[\boldsymbol{\beta}]}_k + \sum_{k \in [n]:\ Q_k = \forall} a^{\exists}_{j,k} \boldsymbol{\beta}(x_k) \leq (\boldsymbol{b}^{\exists}_{(\boldsymbol{\mu})})_j.$$

In π' we can write it as

$$\sum_{\substack{k \in [n] \\ x_k \in \boldsymbol{X}}} a^{\exists}_{j,k} x_k + \sum_{\substack{k \in [n] \\ x_k \in \boldsymbol{Z}^{(\boldsymbol{Y}=\boldsymbol{\mu})}_{\Phi_{l_i}}}} a^{\exists}_{j,k} x_k + \sum_{\substack{k \in [n]:\ Q_k = \exists \\ x_k \text{ inner}}} a^{\exists}_{j,k} x_{k\,\Phi_{l_i}}^{(\boldsymbol{Y}=\boldsymbol{\mu})[\boldsymbol{\beta}]} + \sum_{k \in [n]:\ Q_k = \forall} a^{\exists}_{j,k} \boldsymbol{\beta}(x_k) \leq (\boldsymbol{b}^{\exists}_{(\boldsymbol{\mu})})_j.$$

But notice that all $x_k \in \boldsymbol{Z}^{(\boldsymbol{Y}=\boldsymbol{\mu})}_{\Phi_{l_i}}$ match the renaming annotation in π'. Simplifying again gets us

$$\sum_{\substack{k \in [n] \\ x_k \in \boldsymbol{X}}} a^{\exists}_{j,k} x_k + \sum_{\substack{k \in [n]:\ Q_k = \exists \\ x \notin \boldsymbol{X}}} a^{\exists}_{j,k} x_{k\,\Phi_{l_i}}^{(\boldsymbol{Y}=\boldsymbol{\mu})[\boldsymbol{\beta}]} \leq (\boldsymbol{b}^{\exists}_{(\boldsymbol{\mu} \sqcup \boldsymbol{\beta})})_j.$$

We now take the same row in the inner part of Λ and combine it with $\boldsymbol{\mu} \sqcup \boldsymbol{\beta}$ to get the axiom

$$\sum_{k \in [n]:\ Q_k = \exists} a_k x_k^{[\boldsymbol{\mu} \sqcup \boldsymbol{\beta}]} + \sum_{k \in [n]:\ Q_k = \forall} a_k \boldsymbol{\beta}(x_k) \leq (\boldsymbol{b}^{\exists})_j.$$

Note that $\boldsymbol{\mu} \sqcup \boldsymbol{\beta}$ is a disjoint union because $\boldsymbol{Y}$ is already assigned in Ψ_i, and each universal block satisfies its domain given by bounds and universal constraint system. Subtracting both sides ends us with the axiom exactly as it was in π', because $\boldsymbol{X}$ variables are not changed under $[\boldsymbol{\mu} \sqcup \boldsymbol{\beta}]$ and the $\boldsymbol{\mu}$ annotation is already present in the π' proof. π' is therefore a refutation in the original formula. ◀

A.2 Multilevel Critical Node Problem

We consider the multilevel critical node problem [1]. Given a directed graph $G = (V, E)$. Two agents act on G: The *attacker* selects a set of nodes she wants to infect and the *defender* tries to maximize the number of saved nodes. The defender can *vaccinate* nodes before any infection occurs and *protect* a set of nodes after the attack. An infection triggers a cascade of further infections that propagates via the graph neighborhood, only stopped by vaccinated or protected nodes. For each action (vaccination, infection, protection) a budget (Ω, Φ, Λ, respectively) exists limiting the number of chosen nodes. For any node $v \in V$ binary variables z_v, y_v, and x_v are used to indicated its vaccination, infection, and protection, respectively. Variables $\alpha_v \in \{0, 1\}$ indicate whether node $v \in V$ is saved eventually. Only the variables $\boldsymbol{y}$ are universally quantified. Their domain is restricted by a budget constraint, i.e., $\mathcal{U}_{\boldsymbol{y}} = \{\boldsymbol{y} \in \{0,1\}^V \mid \sum_{v \in V} y_v \leq \Phi\}$. Hence, the universal constraint system only contains the single budget constraint. A QIP with polyhedral uncertainty set can be stated as follows:

$$\max_{\boldsymbol{z} \in \{0,1\}^V} \min_{\boldsymbol{y} \in \mathcal{U}_{\boldsymbol{y}}} \max_{\substack{\boldsymbol{x} \in \{0,1\}^V \\ \boldsymbol{\alpha} \in \{0,1\}^V}} \sum_{v \in V} \alpha_v \tag{5a}$$

$$\text{s.t. } \exists \boldsymbol{z} \in \{0,1\}^V\ \forall \boldsymbol{y} \in \mathcal{U}_{\boldsymbol{y}}\ \exists \boldsymbol{x} \in \{0,1\}^V\ \boldsymbol{\alpha} \in \{0,1\}^V : \tag{5b}$$

$$\sum_{v \in V} z_v \leq \Omega \tag{5c}$$

$$\sum_{v \in V} x_v \leq \Lambda \tag{5d}$$

$$\alpha_v \leq 1 + z_v - y_v \qquad \forall v \in V \tag{5e}$$

$$\alpha_v \leq \alpha_u + x_v + z_v \qquad \forall (u, v) \in E \tag{5f}$$

Constraint (5e) ensures that infected nodes cannot be saved, unless they were vaccinated and Constraint (5f) describes the propagation of the infection to neighboring nodes that are neither vaccinated nor protected. This model corresponds exactly to the trilevel program presented in [1] with the key difference, that we are able to directly plug this model into our solver to obtain the optimal solution, without having to dualize, reformulate or develop domain specific algorithms. The same is true for the QIP solver `Yasol`.

A.3 Further Experiments

A.3.1 Mixing Methods

As outlined in Section 3.4.2, there is potential to link search-based and expansion-based approaches. To demonstrate this, we conducted an experiment using multilevel critical node instances. 1. Run `Yasol` and record the time t_{opt} at which the optimal solution $z^\star$ is first

found (its existence is verified, not its optimality). 2. Run `EQuIPS`, separately, with the objective function constraint bound $z^\star + 1$ (note the maximization objective function) and record the verification time t_{ver}. This hypothetical solver – running `Yasol` in parallel with `EQuIPS` for verification – with runtime $t_{\text{opt}} + t_{\text{ver}}$ solves 27 more instances than `Yasol` alone and has strictly lower runtime on 196 instances.

We also tested whether adding learned constraints from `Yasol`'s search benefits the expansion-based solver. We modified `Yasol` to extract every detected conflict as a constraint until the optimal solution was found (extracting beyond this point might prune the optimal solution). For a single instance, we obtained 73 learned constraints and created three variations of the verification instance: one without added constraints, one with all 73 constraints, and one with three hand-picked constraints. Table 2 reports the runtimes and the number of IP calls in the `wins1` function. For this instance, incorporating learned

Table 2 Comparison of three verification instances.

instance type	original verification instance	org. + 73 constraints	org. + 3 constraints
runtime	36.3s	44.4s	22.2s
calls to IP solver	1034	648	550

constraints reduced the iterations (and thus IP calls) for the expansion-based solver. However, too many constraints may increase IP solver runtime; therefore, selectively transferring the "most beneficial" constraints to `EQuIPS` can significantly enhance the verification process. Although we expected the 73-constraint instance to have fewer IP calls than the one with three constraints, this discrepancy likely results from testing only a single instance. Averaged over a larger set, more constraints should decrease the number of IP calls.

A.3.2 Performance of `EQuIPS` vs. `Yasol` on other optimization test sets

While the computational experiments in Sections 5.1 and 5.2 show promising results – suggesting that our expansion-based approach can effectively compete with the search-based solver `Yasol`– this cannot be stated as a general conclusion, particularly for optimization instances. We conducted tests on 1800 multistage robust assignment instances from [30] and 270 multistage robust scheduling instances from [34]. The former involve combinatorial matching problems under cost uncertainty, while the latter model aircraft scheduling with uncertain arrival times. Both datasets include instances with up to seven decision stages. In both cases, instances can be encoded either with or without polyhedral uncertainty, labeled with QIP_{PU} and QIP, respectively.

Table 3 Number of solved instances and median runtimes on different test sets.

	solved instances					median run times (seconds)				
	Assignment		Scheduling		MCN	Assignment		Scheduling		MCN
	QIP	QIP_{PU}	QIP	QIP_{PU}		QIP	QIP_{PU}	QIP	QIP_{PU}	
`Yasol`	1800	1800	262	264	431	0.4	0.4	56.7	29.0	247.8
`EQuIPS`	1599	1760	194	247	465	10.6	2.7	287.9	51.3	83.8

Table 3 presents the number of instances solved within a 1800-second time limit as well as the median runtime, highlighting that `EQuIPS` struggles with these problem types. Several factors may explain this behavior. First, for assignment instances without a universal

constraint system, the structure requires existentially quantified variables to adapt to any changes in universally quantified variables. As a result, nearly the entire expansion must be constructed before a solution can be determined, drastically increasing computational complexity. Additionally, many of the tested instances feature multiple levels of universally quantified variables, unlike the MCN instances, which contain only a single universal level.

This aligns with expansion-based solvers in the QBF domain, which perform well with few quantifier alternations but struggle as the number of universal levels increase.

A.3.3 Adapted `wins1` Function in Case of Empty Abstraction

Several aspects of Algorithm 1 allow for a choice between standard techniques and more sophisticated implementations. One such aspect is selecting a winning move for the empty abstraction, occurring at Line 4. The call to `EQuIPS` leads to `wins1`, which, for $\mathcal{Q} = \exists$, returns any assignment satisfying the domain constraints. In our implementation, we initially returned the lower bounds of existentially quantified variables, avoiding the IP solver. While computationally inexpensive, this can result in an assignment that violates the existential constraints, making the first countermove less meaningful.

An obvious alternative is to use the IP relaxation, where we solve the existential constraint system without considering quantification. The goal is to obtain stronger moves that lead to more relevant countermoves, ultimately reducing the number of subgames to be considered. Note, that as we observed superior runtimes when using this existential IP relaxation, all reported results so far, are based on this implementation.

However, solving an IP is more computationally expensive than assigning lower bounds. We investigated this trade-off using the same test sets as in Appendix A.3.2. Results in Table 4 show that fewer instances of the assignment test set are solved within the 1800-second time limit with the existential IP relaxation. However, for other test sets, `EQuIPS` performance improves, with median runtimes decreasing overall.

Table 4 Number of solved instances and median runtimes, with `EQuIPS` using lower bounds (LB) vs. solving the existential IP relaxation (exist. IP) to find a winning move for the empty abstraction.

	solved instances					median run times (seconds)				
	Assignment		Scheduling		MCN	Assignment		Scheduling		MCN
	QIP	QIP_{PU}	QIP	QIP_{PU}		QIP	QIP_{PU}	QIP	QIP_{PU}	
LB	1600	1770	175	223	461	12.3	3.8	590.9	127.8	121.9
exist. IP	1599	1760	194	247	465	10.6	2.7	287.9	51.3	83.8

Disjunctive Scheduling in Tempo

Emmanuel Hebrard
LAAS-CNRS, Université de Toulouse, France

Abstract

In this paper we introduce a constraint programming lazy clause and literal generation solver embarking ideas from SAT Modulo theories. A key aspect of the solver are Boolean variables with an associated semantic in difference logic, i.e., systems of binary numeric difference constraints or *edges*, making it particularly adapted to scheduling and other temporal problems. We apply this solver to disjunctive scheduling problems, where edges are used as branching variables, can be inferred via the edge finding rule as well as by transitivity reasoning, and can in turn strengthen propagation via temporal graph reasoning. Our experiments on job-shop scheduling show that a deep integration of these techniques makes our solver competitive with state-of-the-art approaches on these problems.

2012 ACM Subject Classification Applied computing; Computing methodologies → Discrete space search

Keywords and phrases Scheduling, Constraint solvers, Clause-learning

Digital Object Identifier 10.4230/LIPIcs.CP.2025.13

Supplementary Material *Software (Source Code)*: `https://gitlab.laas.fr/roc/emmanuel-hebrard/tempo` [23], archived at `swh:1:dir:bbab6415fb8f44fef04878de35e7b50e797596bd`

Funding *Emmanuel Hebrard*: supported by the ANR Project AD-Lib (ANR-22-CE23-0014); and the European Union's Horizon Europe Research and Innovation program TUPLES No101070149.

1 Introduction

Constraint solvers are constantly evolving. In particular, even tough some restricted form of nogood learning had been used in CP for a long time [19, 43], *conflict-driven clause learning* (CDCL) has had a significant impact on the design of constraint solvers. Bacchus and Katsirelos proposed to compute *explanations* of constraints in order to apply the principles of CDCL on constraint programs [25, 26]. Most hybrid CP/SAT solvers now implement the approach known as *Lazy Clause Generation* (LCG) [17, 36]: domains (and domain changes) are mapped to logical literals (including membership and bound literals), and constraints are propagated using dedicated algorithms as in CP although they generate *explanation clauses* in order to apply CDCL's conflict analysis method. However, even though there are successful and established LCG solvers such as Chuffed [12], their implementation choices are less grounded than in pure CP solvers such as ACE [31], Choco [40, 41][1] or Gecode [44], and questions about their design remain open.

We introduce Tempo, yet another CP/SAT hybrid solver whose implementation of numeric variables is inspired from *difference logic*. This design choice makes it particularly suited for temporal and in particular scheduling problems. For instance, similar approaches have been shown to be very efficient on flexible job-shop [45] or on problems with reservoir resources[46], although such approaches are not limited to this domain. We discuss state-of-the-art solvers for this type of problems in Section 3 where we argue that in many cases the gains in performance came together with a significant complexity growth, such as the recourse to portfolios of algorithms. The overall architecture of Tempo, described in Section 4, is on

[1] Note that recent versions of Choco also implements clause learning.

© Emmanuel Hebrard;
licensed under Creative Commons License CC-BY 4.0

the contrary relatively simple. One of its key aspects is that bound literals are purely *lazy*, that is, a literal $[\![x \leq k]\!]$ is created in a branch of the search tree only when necessary and it "exists" in the data structure only in the subtree rooted at this branch. The solver Aries [6, 7] is built on the same or extremely similar ideas. However, Tempo implements constraint propagation, including global constraints via a constraint queue with configurable priority while Aries is restricted to a few simple constraints. The key differences with respect to Aries are described in Sections 5 and 6.

We report the results in Section 8 of extensive experiments on disjunctive scheduling demonstrating that it is competitive on these benchmarks with toolkits such as CP Optimizer [24], OR-Tools [37] or OptalCP [39], although it is an arguably much simpler approach.

2 Background

In this section we briefly introduce the necessary background on Boolean Satisfiability (SAT), hybrid CP/SAT solvers, Difference Logic and scheduling.

2.1 Algorithms for Boolean Satisfiability

The *Satisfiability* problem is to decide if there exists an assignment of a set $\mathcal{A}$ of atoms such that a propositional logic formula φ evaluates to `true`, that is, whether there exists a solution to: $\exists \mathcal{A}.\varphi$. We shall use the symbol $a \in \mathcal{A}$ to denote both the Boolean variable and the *positive literal* standing for this variable being true, while $\neg a$ is the *negative literal*, standing for this variable being false. It is often assumed that the formula φ is in *Conjunctive Normal Form* (CNF). A CNF is a conjunction of *clauses*, i.e., a disjunction of *literals*.

Conflict-driven Clause Learning (CDCL) solvers [34] make branching decisions as other tree search algorithms, by adding an arbitrary new true literal to the formula. However, when encountering a failure, instead of undoing the most recent decision, a *conflict clause*, or nogood, is generated and added to the clause set. Let $\mathcal{F}$ be the list, in chronological order, of literals currently true in some branch of the search tree, let $rk(l)$ be the rank of literal l in $\mathcal{F}$ and let the *explanation* $(reason(l) \implies l)$ for l be the clause that unit-propagated l (and we call $reason(l)$ the *reason* for l). *Conflict analysis* usually computes a conflict clause with a *unique implication point* (UIP) whose rank is higher than or equal to that of the last decision. Let a *conflict set* c be a conjunction of literals that entails unsatisfiability of the formula φ, i.e., such that $\varphi \wedge c \models \bot$. The conflict set is initialized as the negation of the clause that triggered the failure. Then the literal $l \in c$ with maximum rank $rk(l)$ is removed from c and replaced by its reason $reason(l)$ until exactly one literal in c has a rank superior or equal to that of the most recent decision (this literal is the first UIP, i.e., 1-UIP). This process is called *conflict-driven clause resolution*.[2] Then, every branching decision whose rank is higher than the highest rank of any literal in c but the 1-UIP is undone, the conflict clause $\neg c$ is added to the formula, and unit-propagation infers the negation of the 1-UIP literal.

2.2 Hybrid Solvers

Hybrid CP-SAT solvers work on the same principle, with two key differences. Firstly, there is a mapping between Boolean literals and *restriction events* on the domains of numeric (often integer) variables. For instance, a literal $a_{i,v}$ may stand for the assignment event

[2] We define it using conflict sets because it is more intuitive and convenient, however, those are indeed resolution steps when adopting the clausal view.

$x_i = v$ and $\neg a_{i,v}$ to the removal event $x_i \neq v$. Secondly, arbitrary constraint propagation algorithms can be used as well as clauses to infer new true literals. In order to use conflict analysis as described above, general constraints must be able to produce *explanation clauses*. Given a literal l inferred by some constraint propagation algorithm in a search state defined by the set of true literals $\mathcal{F}$, an explanation clause is a clause c entailed by the model and such that unit-propagation on $\mathcal{F} \wedge c$ produces the true literal l. The name *Lazy Clause Generation* comes from the fact that explanation clauses are generated during search rather than proactively as an encoding. However, this should not be confused with another type of laziness: explanation clauses can be generated during constraint propagation, or in a true "lazy" way, only when we want to replace a literal in the conflict set with its reason. In Tempo the two options are used, depending on the constraint.

2.3 Difference Logic

Satisfiability Modulo Theories (SMT) solvers decide the satisfiability of formula of the form $\exists \mathcal{X}.\varphi$ with φ a propositional logic formula whose atoms have a semantic in an underlying *theory*. *Difference Logic* is a fragment of linear arithmetics where atoms stand for binary difference constraints of the form $[\![x - y \leq k]\!]$ with $x, y \in \mathcal{X}$ and k a constant.

The duality between shortest path and systems of difference constraints is well know [4]. A conjunction of difference logic constraints $\mathcal{E}$ can be mapped to a *temporal network*, i.e., a directed graph $G_{\mathcal{E}}$ with one vertex per numeric variable, and a labeled edge $y \xrightarrow{k} x$ for every difference constraint $[\![x - y \leq k]\!] \in \mathcal{E}$. The system $\mathcal{E}$ is satisfiable if and only if $G_{\mathcal{E}}$ does not contain any negative cycle, and the difference constraint $[\![x - y \leq k]\!]$ is *entailed* by $\mathcal{E}$ if and only if the length $d_{\mathcal{E}}(y, x)$ of the shortest path from y to x in $G_{\mathcal{E}}$ is less than or equal to k. For this reason, we shall refer to difference constraints as *edge* in this paper.

Given the set of edges $\mathcal{A}$ that appear in a formula, and given a subset $\mathcal{E} \subseteq \mathcal{A}$ of *true edges* (edge literals assigned to "true"), a difference logic reasoner is expected to decide if $\mathcal{E}$ is consistent. If so, it should produce the set $\mathcal{E}' \subseteq (\mathcal{A} \setminus \mathcal{E})$ of edges entailed by $\mathcal{E}$, along with their explanations. If not, it should produce an explanation for the inconsistency. It is possible to compute all the entailed edges in polynomial time, by computing all-pair shortest paths in the graph $G_{\mathcal{E}}$ with e.g., the Floyd-Warshall algorithm. State-of-the-art implementations do not compute the full transitive closure. Instead, when a new true edge is added to the system, only the edges that appear in the formula and have become entailed by the system are computed. This computation can be done incrementally, by maintaining a potential function and using Dijkstra's algorithm, after relabeling the edges in the same way as in Johnson's algorithm [13, 16].

2.4 Scheduling

The problem of scheduling tasks under resource constraints has been an extremely successful domain for constraint programming. We are given a set of tasks $\mathcal{I}$, where $i \in \mathcal{I}$ is defined by an interval of time of possibly variable length p_i; where s_i denotes the start time of task i; and where e_i is its end time. Tasks can be subject to precedence constraints, set between their start or end times. They can also be subject to resource constraints. In this paper we consider only disjunctive constraints, stating that two tasks using the same resource may not overlap. Difference logic captures well these problems since precedence constraints are edges and the NoOverlap constraint (also referred to as *disjunctive* or *unary* resource) is defined a 2-CNF whose atoms are edges:

$$\text{NoOverlap}(\mathcal{I}) \iff \forall i, j \in \mathcal{I},\ [\![e_i - s_j \leq 0]\!] \vee [\![e_j - s_i \leq 0]\!] \tag{1}$$

Moreover, this formulation should make it clear that if all difference logic atoms are assigned a truth value, and if the resulting difference system is consistent, then the problem is satisfiable. Indeed, all constraints (precedences and resources) are defined exclusively using difference logic atoms. Therefore, if we maintain the consistency of the difference logic system, there is no need to branch on start times, end times or durations in the search tree. In this paper we only consider problems where the objective function to minimize is the maximum of all the end times (i.e., the *makespan*), although several other objectives are of interest.

3 Related Work

IBM's CP Optimizer [29] has been considered as the state of the art on a range of scheduling problems for a long time. It is a commercial toolkit, its code is not open and if there is a significant amount of literature by various contributors, it is difficult to know with a high degree of certainty which techniques are used. However, the main components are described in [29]. With respect to constraint propagation, the default parameters are not known, but it is safe to assume that most if not all propagation rules for the NoOverlap constraint are implemented in CP Optimizer, and may be switched on or off during search or at preprocessing. Moreover, CP Optimizer uses the idea of gathering edges into a temporal network that can be used to make inference. Using Bellman-Ford on this graph to propagate the bounds of temporal variable is equivalent to using classic constraint propagation on precedences except that negative cycle can be found more quickly on very large domains. Although this is not confirmed in [29], we believe that it very likely uses this technique in conjunction with resource constraints to adjust the bounds of temporal variable in a similar way as we describe in Section 4.2, since these ideas were described in [27]. A significant effort has been given to the default search strategy. Large neighborhood search [47] is used to quickly find high quality solutions. Neighborhood selection (among a portfolios) as well as completion strategies selection relies on machine learning and some adaptive criterion [28]. When trapped in local minima the default strategy switches to Failure-Directed Search (FDS) [54]. In this strategy, branching choices are binary splits on the tasks' start times. Moreover, it implements a rather complex system of *rating* for the choicepoints, with similarities to impact-based search [42]. It can also be compared to the notion of *weight* used in the *weighted-degree* heuristic [8] or of the *activity* used in the VSIDS heuristic [35], since it tries to focus on (and select choicepoints leading to) failures. CP Optimizer does not use clause-learning, however FDS includes storing and propagating nogoods from restarts [32]. Finally, it uses a limited form of *strong branching* [2] by propagating both branches of a subset of binary choicepoints before actually making a branching choice. This technique is used only on a subset of promising choicepoints and applied only near the root of the search tree. Google's OR-Tools [38] implements many of the features in CP Optimizer, including the integration of a simplex algorithm, dedicated scheduling algorithms from operation research and CP and large neighborhood search as well as other local search techniques [14]. However, unlike CP Optimizer, it is an hybrid CP/SAT solver implementing lazy clause generation techniques taking inspiration from Chuffed. OptalCP is a commercially released constraint solver, which shares many traits with CP Optimizer, and a focus on implementation and design choices geared toward a more efficient use of CPU and caches [53]. It was shown to be even more efficient than CP Optimizer, in particular on disjunctive scheduling problems.

Those three toolkits are designed to be usable by non-experts. Therefore, they typically implement highly complex search and parameter tuning strategies as well as inprocessing techniques in order to be robust, instead of leaving these choices to the user. Finally, they are

designed to get the most out of multiprocessing, through the use of portfolios of algorithms and communication between threads. This approach has proven extremely effective and lead to several successful applications. On the other hand, simpler approaches have been shown to be very effective on disjunctive scheduling problems: although using a pure CP solver (Mistral), an early approach using the observation that we only need to branch on edge variables was used with success on disjunctive scheduling problems [20]. It was in particular the first to solve every Open-shop instance in Brucker's data set [9] to optimality [21]. Since this approach did not use any dedicated resource constraint propagation algorithm, its efficiency had two main explanations: it explores a search tree whose choicepoints are edges, as explained above; and the *weighted-degree* heuristic was shown to be very effective in finding the important variables to branch on. An extension of this method with a limited form of clause learning was later shown to be slightly more effective on job-shop benchmarks [48]. More recently, a very efficient solver, called Aries, based on the same principles, but implementing clause-learning as well as all the standard CDCL techniques was proposed. [7] This approach used many of the same ideas on difference logic and edges developed in this paper. With respect to the constraint model and inference mechanisms, as far as we are concerned with disjunctive scheduling, Aries can be seen as Mistral plus clause-learning and the approach we propose, Tempo, can be seen as Aries plus global constraints. The variable heuristics are different though working on the same principles: Aries uses *learning-rate* [33] as default heuristic while the version of Tempo used in our experiments *variable state independent decaying sum* [35].

4 Principles of the Solver

The key difference between a pure CP and a hybrid CP/SAT solver is that in the latter, the information about the current domains, and its changes thereof, must be encoded in atomic referenceable *literals*. We detail the different types of literals in Tempo in Sections 4.1 and 4.2, and then we show how they affect a core technique in SAT solver: unit-propagation.

4.1 Variables and Literals

There are two primitive types of variables in the solver: *Boolean* and *numeric* variables. The domain of a numeric variable is defined by its lower and upper bounds. Bound literals are constraints of the form $[\![\pm x \leq k]\!]$. We use signed variables so that bound literals are all in the canonical form of difference logic constraints ($[\![x \geq k]\!]$ is written $[\![-x \leq -k]\!]$), and so that a smaller right-hand side always corresponds to a tighter literal. We say that $[\![\pm x \leq k]\!]$ is *active* if it was created (e.g. by constraint propagation) at a decision level lower than or equal to the current one in the same branch. In other words, a literal $[\![\pm x \leq k]\!]$ can be true without being active, if there is an active literal $[\![\pm x \leq k']\!]$ with $k' < k$. For every signed variable, the list of active literals maintained in decreasing order of their right-hand sides. For instance, if the lower bound of a numeric variable x changes (to take the value k) because of constraint c, the literal $[\![-x \leq -k]\!]$ is inserted at the end of the list for $-x$ and its explanation is c. Some solvers [26, 17] support lazy bound literal generation. A possible implementation is essentially equivalent to adding new Boolean variables during search with a bound semantic attached to it. This means that the corresponding Boolean variable must live as long as a clause involves such a literal. More generally, adding such literals during search is easy, however, making sure that the number of literals does not grow too large is difficult. Therefore, in Tempo, much like in Aries, the literal $[\![\pm x \leq k]\!]$ is not linked to any Boolean variable. It is created when the domain of variable x changes accordingly, and no longer exists when backtracking over that decision level in the search tree. However, this

incurs a cost when accessing the informations attached to such a literal. For instance if one wants to access the explanation for $[\![\pm x \leq k]\!]$, one must search for the highest value of $k' \leq k$ such that the literal $[\![\pm x \leq k']\!]$ is currently active. Since $[\![\pm x \leq k']\!]$ entails $[\![\pm x \leq k]\!]$, its explanation is also valid[3]. Similarly, if the literal $[\![\pm x \leq k]\!]$ is inferred e.g., by constraint propagation, unit-propagation algorithm would access the list of clauses that watch it in constant time. This is impossible if we do not allocate any memory for literals that are not active. Our implementation of the watched literals data structure is discussed in Section 4.3.

4.2 Temporal Network

As in SAT Modulo Theories solvers, Boolean variables may be given a semantic in difference logic, i.e., each literal of a Boolean variable a can be associated to an edge: $a \implies [\![x - y \leq k]\!]$. The negative literal $\neg a$ can stand for the negation of the edge ($[\![y - x < -k]\!]$). However, we allow for the positive and negative literals of the same variable to be associated to any edges, or no edge at all. This is useful to represent a binary disjuncts with a single variable, e.g.:

$$[\![e_i - s_j \leq 0]\!] \text{ XOR } [\![e_j - s_i \leq 0]\!] \tag{2}$$

The binary disjunct in Eq. 2, stating that tasks i and j do not overlap can be represented with a single variable $a_{i,j}$ with $a_{i,j} \implies [\![e_i - s_j \leq 0]\!]$ and $\neg a_{i,j} \implies [\![e_j - s_i \leq 0]\!]$. This is also useful for optional tasks, where we might want that a Boolean variable implies an edge, but its negation leaves the numeric variable free.

The difference logic system in implemented in Tempo uses a reversible directed graph $G_{\mathcal{E}}$. When the literal $[\![x - y \leq k]\!]$ is set to `true`, the edge is directly added to the difference logic system, modeled by the arc $y \xrightarrow{k} x$ in $G_{\mathcal{E}}$. Our implementation differs from the method described by Cotton and Maler [13] in the following ways: Firstly, as explained by Feydy et al., in typical constraint problems, many more bounds literals are propagated than edge literals, and hence it is important to have a dedicated procedure for each type [16]. Moreover, since bounds literals are generated lazily, they do not necessarily appear in the formula prior to search. We use the common approach to have a distinguished vertex standing for the constant 0 in the temporal network. This vertex is connected to every other numeric variable, however *implicitly*. That is, if $[\![x \leq k]\!]$ is the tightest true upper bound literal for variable x, then everything works as if the arc $0 \xrightarrow{k} x$ was in the graph, i.e., the difference constraint $[\![x - 0 \leq k]\!]$, and similarly, $[\![-x \leq -k]\!]$ indicates the arc $x \xrightarrow{-k} 0$. Those arcs are not actually added to avoid having several arcs between the same vertices. Therefore, given the set of edges $\mathcal{A}$ that appear in the constraint program and a subset $\mathcal{E} \subseteq \mathcal{A}$ of true edges, for each new edge $[\![x - y \leq k]\!]$, the reasoner achieves three tasks: decide consistency; compute the set $\mathcal{B}$ of bounds literals entailed by $\mathcal{E} \cup \{[\![x - y \leq k]\!]\}$; and compute a set $\mathcal{E}' \subseteq (\mathcal{A} \setminus (\mathcal{E} \cup \{[\![x - y \leq k]\!]\}))$ of entailed (non-bound) edges.

As in [16], given a new arc $y \xrightarrow{k} x$, the first two tasks are done via two calls to a shortest path algorithm. Let $d(0, x)$ be the length of the shortest path from the origin (0) to x, and $d(x, 0)$ be the length of the shortest path to the origin from x in $G_{\mathcal{E}}$. The bound literals $[\![-x \leq -d(x, 0)]\!]$ and $[\![x \leq d(0, x)]\!]$ are entailed by $\mathcal{E}$. In other words, all the upper (resp. lower) bound updates can be done by computing all the shortest paths from (resp. to) 0 in $G_{\mathcal{E}}$. Since a shortest path can have changed only if it goes through the arc $y \xrightarrow{k} x$ this can be done by one call from x and another from y in the converse graph. Notice that the shortest

[3] Note that $[\![\pm x \leq k]\!]$ cannot be in the explanation of $[\![\pm x \leq k']\!]$ because it would entail that a literal $[\![\pm x \leq k'']\!]$ with $k' < k'' \leq k$ is active but that would contradict k' is the highest such value.

path algorithm can be stopped early when reaching a vertex whose shortest path from (resp. to) 0 has not changed. Our approach has a worse time complexity than those described in [13, 16] because we use a version of Dijkstra with a simple FIFO queue but where the same vertex can enter the queue more than once. The worst-case time complexity of this algorithm is the same as Bellman-Ford. However, as opposed to Dijkstra's algorithm, it can work directly on graphs with negative arcs and is efficient in practice when the new edge does not entail many bound updates. If the shortest path from the origin to x is updated to a strictly lower value (say k) and y is its predecessor on the path, via the arc $y \xrightarrow{k'} x$, then the literal $[\![x \leq k]\!]$ is inferred, and its explanation is:

$$[\![x - y \leq k']\!] \wedge [\![y \leq k - k']\!] \implies [\![x \leq k]\!] \tag{3}$$

Moreover, there is a negative cycle if and only if the shortest path from 0 to y is updated in the first call, and the explanation is the set of edges in the negative cycle.

Finally, the main difference with respect to [13, 16] concerns the third task: we do not infer *all* the edges entailed by the current set of true edges. Indeed this requires to either store all the shortest paths and hence quadratic space, or to run the shortest paths algorithms as explained above, but without the early stop condition. Even though the worst-case time complexity is the same, it makes a significant difference on dense precedence networks. In our experiments this turned out to be too costly, and instead we compute only a subset of entailed edges. The rule to infer that the edge $[\![x - y \leq k]\!]$ is false corresponds to the standard propagation rule for binary disjunctions in existing scheduling models. Within the difference logic system implemented in Tempo, the rule is as follows:

$$d(x, 0) + d(0, y) < -k \implies \neg[\![x - y \leq k]\!] \tag{4}$$

Indeed the following difference logic sentence is unsatisfiable

$$d(x, 0) + d(0, y) < -k \wedge [\![0 - x \leq d(x, 0)]\!] \wedge [\![y - 0 \leq d(0, y)]\!] \wedge [\![x - y \leq k]\!] \tag{5}$$

since it can be reduced to $0 < 0$.

For every edge $[\![x - y \leq k]\!]$ in the difference logic system, there is a propagator that watches the lower bound literals of the numeric variable x and the upper bound literals of the y, and triggers exactly as shown in Eq. 4 to produce the literal $\neg[\![x - y \leq k]\!] = [\![y - x < -k]\!]$. The explanation can be:

$$[\![0 - x \leq d(x, 0)]\!] \wedge [\![y - 0 \leq d(0, y)]\!] \implies \neg[\![x - y \leq k]\!] \tag{6}$$

We use a slightly improved explanation. Assume that the most recent bound literal is $[\![0 - x \leq d(x, 0)]\!]$. Then the explanation for $\neg l \iff \neg[\![x - y \leq k]\!]$ is:

$$[\![0 - x \leq -k - 1 - d(0, y)]\!] \wedge [\![y - 0 \leq d(0, y)]\!] \implies \neg l \tag{7}$$

Since we only need $d(x, 0) + d(0, y) < -k$ and this explanation is therefore more general.

4.3 Unit-Propagation of Bound Literals

Since bound literals are generated lazily, we cannot have constant-time access to all data structures storing information on literals. Bound literals are simply stored in order, with one list for upper bounds and another list for lower bounds. Therefore, the current lower and upper bound of a given numeric variable can be accessed in constant time. However, accessing the explanation for a given literal $[\![x \leq k]\!]$ requires to traverse the list of upper bounds literals for the variable x. This could be done by binary search, however, since

conflict analysis usually does not explore literals beyond the current decision level, we chose to use a linear search starting from the most recent bound instead. Since conflict analysis is not as time consuming as constraint propagation, this is not a very important issue.

On the other hand, unit-propagation represents a significant proportion of the total runtime, and the *watched-literal* data-structure [35] must be adapted. In the standard method, we have a list of *watcher* clauses for every literal, containing every clause that watches it (every clause watches two of its literals). We use the standard implementation when the watched literal is Boolean. Instead, for bound literals, we have an array indexed by signed numeric variables, i.e., $\mathcal{W}_x$ is the watch-list for the upper bound of x and $\mathcal{W}_{-x}$ is the watch-list for its lower bound (see example in Figure 1). This array maps each signed variable to a list $\mathcal{W}_{\pm x}$ containing pairs $(k, W_{[\![\pm x \leq k]\!]})$ where k is a numeric value and $W_{[\![\pm x \leq k]\!]}$ is the list of clauses currently watching the literal $[\![\pm x \leq k]\!]$. Those pairs are ordered by increasing value of k. When a literal $[\![x \leq k']\!]$ (resp. $[\![-x \leq -k']\!]$) becomes true, we scan the list $\mathcal{W}_{-x}$ (resp. $\mathcal{W}_x$). When encountering a pair $(k, W_{[\![\pm x \leq k]\!]})$ with $k + k' \geq 0$ we know that the clauses in $W_{[\![\pm x \leq k]\!]}$ and in all the list after $W_{[\![\pm x \leq k]\!]}$ do not need to be unit propagated since they watch a literal weaker than $[\![\pm x \leq k']\!]$, and hence no more propagation can occur.

$$
\begin{array}{ll}
W_{[\![x \leq 2]\!]}: & [\![\mathbf{x \leq 2}]\!] \vee [\![-y_1 \leq -3]\!] \vee [\![y_3 \leq 6]\!];\ [\![\mathbf{x \leq 2}]\!] \vee [\![y_3 \leq 8]\!] \vee [\![-y_2 \leq -4]\!] \\
W_{[\![x \leq 5]\!]}: & [\![\mathbf{x \leq 5}]\!] \vee [\![-y_3 \leq -7]\!] \vee [\![-y_1 \leq -3]\!];\ [\![\mathbf{x \leq 5}]\!] \vee [\![-y_2 \leq -5]\!] \vee [\![y_3 \leq 7]\!] \\
W_{[\![x \leq 10]\!]}: & [\![\mathbf{x \leq 10}]\!] \vee [\![y_1 \leq 5]\!] \vee [\![-y_2 \leq -2]\!]
\end{array}
$$

Figure 1 Watch structure $\mathcal{W}_x$ for a set of clauses watching an upper bound literal on x. If $[\![-x \leq -1]\!]$ becomes true, we can stop at the first "sentinel" check since $2 - 1 \geq 0$. If $[\![-x \leq -8]\!]$ becomes true, the clauses in $W_{[\![x \leq 2]\!]}$ and $W_{[\![x \leq 5]\!]}$ will be unit propagated, but not those in $W_{[\![x \leq 10]\!]}$.

However, the list $\mathcal{W}_{\pm x}$ must be maintained sorted. Therefore, the additional cost of this data structure comes when moving a clause from a watch list to another watch list. That is, suppose that the literal $[\![\pm x \leq k]\!]$ watched by clause c is falsified. If the clause c contains an unwatched, unassigned literal (say $[\![y \leq k']\!]$) then c must be removed from $W_{[\![\pm x \leq k]\!]}$, which can be done in constant time, and added to the list $W_{[\![y \leq k']\!]}$. This can also be done in constant time, however, finding the list $W_{[\![y \leq k']\!]}$ within $\mathcal{W}_y$ takes a time linear in the size of $\mathcal{W}_y$. In other words, the insertion cost will increase linearly with the number of different bound literals currently watched for the same numeric variable.

5 Edge-Finding: propagation and explanation

The standard Edge-Finding [10] rule for the NoOverlap constraint (see Eq. 1) is implemented in Tempo. We write est_i for the *earliest start time* of a task i, that is, the lower bound of the start time variable s_i at a given point in search, and lct_i for its *latest completion time*, i.e., the upper bound of e_i. Moreover we extend this notation to set of tasks, with $est_\Omega = \min_{i \in \Omega} est_i$, $lct_\Omega = \max_{i \in \Omega} lct_i$ and $p_\Omega = \sum_{i \in \Omega} p_i$

The *overload* rule is a sufficient condition for unsatisfiability of NoOverlap($\mathcal{I}$):

$$\exists \Omega \subseteq \mathcal{I},\ p_\Omega > lct_\Omega - est_\Omega \quad \models \bot \tag{8}$$

The Edge-Finding rule uses Eq. 8 to infer precedence relations, i.e., edges:

$$\exists \Omega \subseteq \mathcal{I},\ \exists i \in \Omega,\ p_\Omega > lct_{\Omega \setminus \{i\}} - est_\Omega \quad \models \quad \forall j \neq i \in \Omega,\ [\![e_j - s_i \leq 0]\!] \tag{9}$$

It "finds" the set of edges on the right-hand side of the consequence 9, which are entailed by the fact that if task i is not the last one to be processed in the set Ω, then the latest completion time would be too early to process all the tasks in Ω. Since in the proposed

framework we have variables for edges, we can directly set the edge to true. In standard implementations, however, this inference cannot be directly stored and instead the bounds of the numeric variables are adjusted accordingly via the two following rules:

$$\forall j \neq i \in \Omega \quad e_j \leq \min(lct_j, lct_i - p_j) \tag{10}$$

$$s_i \geq est_\Omega + \sum_{j \neq i \in \Omega} p_j \tag{11}$$

Notice that Eq. 10 is redundant when edges are assigned, although Eq. 11 is not, because it uses the fact that tasks in $\Omega \setminus \{i\}$ must all be completed before i *and they cannot overlap*. This adjustment, however, is weaker than the propagation described in Section 6, and hence our implementation of Edge-Finding only assigns the corresponding edge literals.

Let $\Omega \subseteq \mathcal{I}, i \in \Omega$ such that Eq. 9 is triggered. For each $j \neq i \in \Omega$, the edge $[\![e_j - s_i \leq 0]\!]$ can be explained by the following clause:

$$\bigwedge_{k \in \Omega} [\![-s_k \leq -est_\Omega]\!] \wedge \bigwedge_{k \neq i \in \Omega} [\![e_k \leq lct_{\Omega \setminus \{i\}}]\!] \implies [\![e_j - s_i \leq 0]\!] \tag{12}$$

However, we use Vilím's algorithm [51] for Edge-Finding, which does not explicitly keep track of a minimal set Ω. For the sake of simplicity, we show here how to explain a failure of the overload rule (Eq. 8) as it is essentially the same construction in the general explanation.

One basic observation is that there are only $n(n-1)/2$ non-dominated sets Ω to consider in Equations 8 and 9 for a resource involving n tasks: given some values for est_Ω and lct_Ω we only need to consider the maximal set of tasks wholly within those bounds, and there are at most $n(n-1)/2$ possible distinct values for est_Ω and lct_Ω. Vilím's algorithm goes one step further and achieves all these tests in $O(n \log n)$ time thanks to the *Theta-Tree* data structure. This algorithm explores tasks in non-decreasing latest completion times. Assume that tasks $1, \ldots, n$ are numbered accordingly (i.e., so that $i < j$ implies $lct_i \leq lct_j$). Given task i it computes the "elastic" approximation $ect_{1,\ldots,i}$ of the *earliest completion time* of sets of tasks $\{1, \ldots, i\}$. An replace Eq. 8 with:

$$ect_{1,\ldots,i} > lct_{1,\ldots,i} \implies \bot \tag{13}$$

Within the Theta-tree structure, the following recursion is used to compute the elastic earliest completion time of a set of tasks $\Theta_l \cup \Theta_r$, from two disjoint sets such tasks in Θ_l have lower earliest start times than tasks in Θ_r (i.e., such that $\forall i \in \Theta_l, j \in \Theta_r,\ est_i \leq est_j$):

$$ect_{\Theta_l \cup \Theta_r} = \max(ect_{\Theta_r}, ect_{\Theta_l} + p_{\Theta_r}) \tag{14}$$

There is one such set of tasks per node in the Theta-tree, with each leaf standing for a singleton set containing one task in $\mathcal{I}$, and internal nodes are computed using Eq. 14. In order to extract sound and minimal explanations, we keep track of an additional information in each node of the Theta-tree: the *relevant earliest start time*, that we write est^*. For a leaf $\{i\}$ we have $est^*_{\{i\}} = est_i$. Then we use the following recursion:

$$est^*_{\Theta_l \cup \Theta_r} = \begin{cases} est^*_{\Theta_r} \text{ if } ect_{\Theta_r} \geq ect_{\Theta_l} + p_{\Theta_r} \\ est^*_{\Theta_l} \text{ otherwise} \end{cases} \tag{15}$$

Then, from a set of tasks Ω we can compute a potentially smaller explanation set: $expl_\Omega = \{j \in \Omega \mid est_j \geq est^*_\Omega\}$. The intuition is that when the earliest computation time of $\Theta_l \cup \Theta_r$ is computed in Eq. 14, if $ect_{\Theta_r} \geq ect_{\Theta_l} + p_{\Theta_r}$ then the tasks in Θ_l are irrelevant and hence can be removed from the explanation. Eq. 15 keeps track of this information. The following theorem essentially says that the set $expl_\Omega$ is a solution of Eq. 8, i.e, sufficient to entail a failure and explainable with Eq. 12.

▶ **Theorem 1.** *For any node of the theta-tree corresponding to the set of tasks Ω, the following equality is true:* $est^*_\Omega + p_{expl_\Omega} = ect_\Omega$

Sketch of proof. The full proof is given in appendix. We use an induction on the size of Ω. There are two cases corresponding to the two sides of the recursion (Eq. 14). The child nodes Θ_l and Θ_r are strictly smaller than $\Theta_l \cup \Theta_r$ and hence they satisfy the induction hypothesis. Moreover the composition is straightforward when $ect_{\Theta_l \cup \Theta_r} = ect_{\Theta_r}$ and only slightly more involved in the second case. ◀

6 Edge Reasoning

All the edges, from the definition of the problem, from branching decisions, from the temporal network (Bellman-Ford or Eq. 4), and from Edge-Finding are stored in the temporal network. We discuss here the possibility of adding yet more edges from transitivity reasoning, and we show how we use all those edges to strengthen propagation on bound literals.

6.1 Transitivity

We do not use the strongest possible consistency in the difference logic reasoner since computing the full transitive closure can be very costly. However, within the scope of a NoOverlap constraint on $\mathcal{I}$, we can compute the transitive closure on the $|\mathcal{I}|(|\mathcal{I}|-1)/2$ edge variables. The transitivity propagation rule is straightforward, and self-explained:

$$[\![e_i - s_j \leq 0]\!] \wedge [\![e_j - s_k \leq 0]\!] \implies [\![e_i - s_k \leq 0]\!] \tag{16}$$

We use a reversible graph data structure for efficient access to the list of current successors and predecessors of a task. When an edge $[\![e_i - s_j \leq 0]\!]$ becomes true we add the edge $[\![e_i - s_k \leq 0]\!]$ for each successor k of j, and $[\![e_k - s_j \leq 0]\!]$ for each predecessor k of i.

6.2 Bounds adjustment

The graph of precedences between tasks sharing the same resource can be leveraged to infer further propagation. This has been described in the more general context of cumulative resources [27], however, we focus here on the simple case of a disjunctive resources and use an algorithm described in Vilím's Ph.D. thesis [52]. Suppose that we have a task j and a set of tasks Ω all requiring the same resource, and such that for each $i \in \Omega$, $[\![e_j - s_i \leq 0]\!]$ is true. Then, although neither the precedences nor the NoOverlap relation can conclude, the tasks in Ω cannot overlap because of the disjunctive resource, and hence we have:

$$e_j \leq lct_\Omega - p_\Omega \tag{17}$$

There is as many non-dominated sets Ω to consider than successors of j. Indeed, for any Ω let $i \in \Omega$ be the task with highest lct_i. The set Ω' containing every successor of j whose latest completion time is lower than or equal than lct_i is such that ${p_\Omega}' \geq p_\Omega$ and $lct_{\Omega'} = lct_\Omega$.

Alg. 1 is not identical, but essentially equivalent to Vilím's. It first sorts the tasks by increasing latest completion time. Now, consider a literal created for task j at Line 1. Task i is a successor of j and each successor whose latest completion time is less than lct_i has been explored earlier (because of the ordering). Therefore, $\Delta[j]$ contains the sum of their durations. Alg. 1 runs in time linear in the number of true edges. It is sound because it applies the rule in Eq. 17, which is sound, and it is complete because we established that it goes through all the potentially non-dominated application of the rule.

Algorithm 1 Disjunctive Temporal Graph Reasoning.

Algorithm: `PrecedenceGraphFiltering` $(\mathcal{A}, \mathcal{I}, G)$

`sort` $\mathcal{I}$ by increasing latest completion time
foreach $i \in \mathcal{I}$ **do**
  foreach *predecessor* j *of* i *in* G **do**
    $\Delta[j] \leftarrow \Delta[j] + p_i$
1   `add literal` $[\![e_j \leq lct_i - \Delta[j]]\!]$

7 Clause Minimization

Clause minimization is an important part of CDCL. It consists in detecting *redundant* literals in the conflict clause (see Section 2.1) and removing them to obtain a stronger clause.

Given a conflict set c for the formula φ, the literal $l \in c$ is said redundant if $\varphi \wedge (c \setminus l) \models \bot$. A sufficient condition is that l is not a decision and its reason $reason(l)$ is a subset of the conflict set c, or that every literal in the difference $reason(l) \setminus c$ is itself redundant. This recursion defines the *implication graph*, which, in the standard clause minimization algorithm, is explored up until encountering a decision or when a given depth is reached. In both of these cases, the literal is considered relevant (non-redundant). Otherwise, the exploration ended because all the ancestors of l belong to c and l is redundant.

However, this procedure is not as simple in the context of lazy bound literal generation. Indeed, the definition of membership of a bound literal $[\![\pm x_i \leq k]\!]$ to a clause c may have more than one definition. A trivial one is that the clause c contains the literal $[\![\pm x_i \leq k]\!]$. However, another possible definition is that the clause c contains a literal $[\![\pm x_i \leq k']\!]$ such that $k' \leq k$. In this case the literal $[\![\pm x_i \leq k]\!]$ is a direct consequence of the clause, and hence this weaker definition of membership yields shorter clauses after minimization. However, one needs to be careful. Assume that Bellman-Ford infers the bound literal $[\![x_i \leq k]\!]$. Then it is explained by a 3-clause of the form $[\![x_i - x_j \leq -p_i]\!] \wedge [\![x_j \leq k + p_i]\!] \implies [\![x_i \leq k]\!]$ (see Eq. 3). Moreover, suppose that the edge $[\![x_i - x_j \leq -p_i]\!]$ was inferred by Eq. 4. Then it has as explanation another 3-clause of the form $[\![x_i \leq k + \epsilon]\!] \wedge [\![-x_j \leq -k - p_i - \epsilon]\!] \implies [\![x_i - x_j \leq -p_i]\!]$ with $\epsilon > 0$ since the literal $[\![x_i \leq k + \epsilon]\!]$ is "older" than $[\![x_i \leq k]\!]$ (i.e., $rk([\![x_i \leq k + \epsilon]\!]) < rk([\![x_i \leq k]\!])$). The *implication graph* for this example is shown in Fig. 2, where each vertex is a literal, and the set of direct predecessors (solid edges) of a literal corresponds to its explanation. The dashed edge is the "implicit" implication via the semantic of the bounds. Notice that it introduces a cycle in the implication graph.

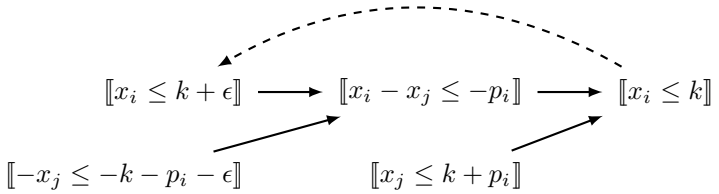


Figure 2 Illustration of the part the implication graph rooted at the literal $[\![x_i \leq k]\!]$.

Suppose that the conflict set (negation of the first UIP clause) contains the literals $[\![x_i \leq k]\!]$, $[\![x_j \leq k']\!]$ and $[\![-x_j \leq -k'']\!]$, with $k' < k + p_i$ and $k'' < k - p_i - \epsilon$. If we query the procedure sketched above for the redundancy of the literal $[\![x_i \leq k]\!]$, it would explore the ancestors of $[\![x_i \leq k]\!]$ and finds they are all implied by the conflict set. As a result, it would wrongfully classify the literal $[\![x_i \leq k]\!]$ as redundant and remove it from the conflict set.

Therefore, one cannot use the implication $[\![\pm x_i \leq k]\!] \implies [\![\pm x_i \leq k + \epsilon]\!]$ when checking the redundancy of literal $[\![\pm x_i \leq k]\!]$. However, we can apply the following rule:

▶ **Proposition 2.** *Let $[\![\pm x_i \leq k]\!] \wedge c$ be a conflict set and let $r^*([\![\pm x_i \leq k]\!])$ be an arbitrary conflict set obtained by conflict-driven clause resolution from $[\![\pm x_i \leq k]\!]$.*

If $r^*([\![\pm x_i \leq k]\!]) \setminus c = [\![\pm x_i \leq k + \epsilon]\!]$ then, $\varphi \wedge c \wedge [\![\pm x_i \leq k + \epsilon]\!] \models [\![\pm x_i \leq k]\!]$

Proof. The proof is direct from the fact that $\varphi \wedge r^*([\![\pm x_i \leq k]\!]) \models \bot$ (because conflict-driven clause resolution is correct), and because, $c \wedge [\![\pm x_i \leq k']\!] = r^*([\![\pm x_i \leq k]\!])$. ◀

Therefore, when Proposition 2 applies, $[\![\pm x_i \leq k]\!]$ is not redundant, but we can weaken it to $[\![\pm x_i \leq k + \epsilon]\!]$ in the conflict set, and hence strengthen the corresponding literal in the conflict clause to $\neg[\![\pm x_i \leq k + \epsilon]\!]$.

8 Experimental Results

We used three types of problems to benchmark our solver: job-shop scheduling, open-shop scheduling and job-shop with time lags. In all three problems there are m machines and n jobs each containing m tasks. All machines are disjunctive resources. In job-shop scheduling, the sequence of the tasks in each job is fixed and each task requires exactly one of the m machines. Job-shop with time lags are job-shop instances with the extra constraint that the interval between the end of any task and the start of the next task on the same job should be less than some value (in the data set it is set to either 0, 0.5, 1, 2, 3 or 10 times the average length of a task). Finally in Open-shop instances, the sequence in each job is a decision variable, or in other words, a job is a disjunctive resource in the same way machines are. We used well known data sets, whose detailed description is given in Table 4 in Appendix.

All experiments ran on a cluster of machines with 3 types of processors: Intel E5-2695 v3 2.3Ghz; Intel E5-2695 v4 2.1Ghz and AMD EPYC 7453 3.4Ghz. All the runs involving job-shop instances ran on the first type, open-shop instances on the second and job-shop with time-lags instances on the third, in order to ensure a fair comparison. We compared against the solvers Aries, CP Optimizer (version 22.11), OptalCP and OR-Tools. We did not run comparison with other solvers, although we quickly assessed the models for job-shop scheduling provided in the sources of Gecode and Chuffed and the results did not seem competitive. We used the following parameter setting for Tempo: the variable selection heuristic is VSIDS; the strategy to select clauses to forget uses literal activities from VSIDS; the value selection heuristic follows the choices that lead to the current best solution [5]; a straightforward insertion heuristic using a randomized mix of earliest start time and minimum slack is used to construct an initial upper bound; and no initial lower bound is implemented, besides the Edge-Finding rule. Some further numeric parameter setting were obtained by running Tempo on 8 instances (4 job-shop, 2 open-shop and 2 job-shop with time lags) and selecting the best out a few reasonable choices. We found that these parameters did not have a very significant impact: the clause minimization depth is set to 10, the decay value for VSIDS is set to 0.99, the ratio of forgotten clauses per restart is 0.7 and a geometric restart policy is used, with a base limit of 128 failures and a factor of 1.05.

For every solver, we used models of job-shop provided in their distribution, and hence we assume that they are reasonably well tuned for these standard instances. Aries, CP Optimizer and OptalCP also provide a default model for open-shop which we used in the same way. We implemented a model for job-shop with time lags by minimally changing the model for job-shop in all four solvers, and did the same for open-shop in OR-Tools.

Table 1 Comparison with the state of the art: number of optimality proofs (#Opt.) and average objective value (Objective) after 30 minutes on "hard" instances. Best results are in bold font.

instances	OR-Tools		Aries		CP Optimizer		OptalCP$_{FDS}$		Tempo	
	#Opt.	Objective	#Opt.	Objective	#Opt.	Objective	#Opt.	Objective	#Opt.	Objective
Jobshop Scheduling Problems										
abz(30)	0	674.0	0	**671.5**	**8**	675.7	0	679.0	2	671.6
cscma(400)	**0**	5690.7	**0**	5472.1	**0**	5364.9	**0**	6405.5	**0**	**5287.1**
ft(10)	**10**	**1165.0**	0	**1165.0**	**10**	**1165.0**	**10**	**1165.0**	**10**	**1165.0**
la(20)	10	1196.3	4	1199.3	**19**	**1193.5**	10	1200.0	13	1193.6
rcmax(390)	10	4354.3	90	4193.0	**181**	4190.2	40	4547.3	170	**4180.7**
swv(150)	10	2139.9	9	2132.2	34	2088.0	20	2293.8	**39**	**2053.6**
ta(710)	44	2652.4	134	2542.6	**326**	**2528.6**	54	2830.9	242	2553.0
yn(40)	**0**	928.0	**0**	921.5	**0**	931.0	**0**	938.5	**0**	**921.4**
all(1750)	84	3583.8	237	3452.5	**578**	3418.1	134	3876.1	476	**3404.9**
Openshop Scheduling Problems										
bru7(10)	0	1048.6	**10**	**1048.0**	**10**	**1048.0**	**10**	**1048.0**	0	**1048.0**
bru8(40)	30	1014.9	**40**	**1014.5**	11	1018.1	**40**	**1014.5**	30	**1014.5**
all(50)	30	1021.7	**50**	**1021.2**	21	1024.1	**50**	**1021.2**	30	**1021.2**
Jobshop Scheduling Problems with Time Lags										
ft_1(10)	0	inf	**10**	**961.0**	**10**	**961.0**	**10**	**961.0**	**10**	**961.0**
la_0(280)	10	**2333.0**	**144**	2344.3	40	2385.8	70	2439.9	119	2364.0
la_0.5(300)	0	1680.2	**104**	**1570.4**	50	1615.1	100	1613.6	79	1589.7
la_1(290)	9	1504.0	109	**1355.1**	139	1385.7	**170**	1374.3	97	1370.4
la_2(30)	11	**895.4**	29	904.7	**30**	904.7	**30**	898.3	**30**	904.7
la_3(120)	32	1552.8	73	1427.7	80	1427.2	70	1433.8	**98**	**1418.2**
la_10(20)	9	1196.5	5	1201.5	**20**	1194.0	10	**1193.5**	10	1194.6
all(1050)	71	1778.7	**474**	**1669.2**	369	1701.2	460	1712.7	443	1682.9

OptalCP, however, uses two main strategies. The default one focuses on finding high quality solutions quickly and relies mainly on LNS, while the other focuses on optimality proofs and relies on FDS. Although the default strategy was remarkably efficient on job-shop, even proving optimality in many cases (see Table 3), it was unable to prove optimality in most other cases, and was much less efficient on job-shop with time-lags. Therefore, we report the results of OptalCP$_{FDS}$ in Table 1 and OptalCP$_{LNS}$ only in Table 3.

8.1 Comparison with the sate of the art

We ran every method on every instance with 10 distinct random seeds. Every run was stopped after 30 minutes of user time. We then grouped the instances into two categories "Easy" and "Hard", where the former corresponds to instances solved to optimality by all five solvers in every run, and the latter contains all the remaining instances.

8.1.1 Hard instances

Results for hard instances are reported in Table 1. For each data set, the total number of runs is shown next to the data set name, then we report the number of proofs (#Opt.) and the average objective value (Objective). No solver is clearly the best on every data set.

On some instances of job-shop with time lags, OR-Tools does not find feasible solutions. The reason is simple: the cycles in the temporal graph make feasibility hard on this problem, even without upper bound. However, without upper bound, constraint propagation is

ineffective and solvers are blind. Other solvers use some form of greedy initialization able to find a reasonable upper bounds to bootstrap search, which is crucial here. We believe that either OR-Tools simply does not run such greedy initialization, or that it sometimes fails to provide an initial feasible solution and hence this observation is not as noteworthy as one might think. However, the overall results are not as good as other solvers in general on these instances. It should also be noted that OR-Tools has been designed to be robust on a much wider range of problems (whereas other solvers are more focused on scheduling) and also to take full advantage of parallelism and is not as efficient in single-thread mode. We consider it beyond the scope of the paper to experiment using multi-threading.

The solver Aries, despite not using any dedicated propagator for the NoOverlap constraint ob-shop with time lags and tied for best on open-shop. In the case of Open-shop, there are only 5 hard instances in the whole data set, all from [9]. Methods that branch on edges and explore the search tree extremely fast have been shown to be efficient on those instances [21]. Moreover, the parameter that is probably the most important here is the variable selection heuristic. We ran Tempo without the NoOverlap propagators (Edge-Finding and Precedence-Disjunctive) since it is then very similar to Aries, with the main difference being the variable selection policy. The results of Tempo are still slightly worse on those instance than with them. We therefore believe that the explanation is related to the variable-selection policy. It is not as simple as simply the difference between learning-rate and VSIDS since we tried both in Tempo, and we did not notice much difference. However, there are many other decisions and parameters in the implementation of these heuristics (which set of literals should get their activity bumped on conflict, which value of decay should be used, etc.). On job-shop with time lags instances, we believe that the reason is mainly that the time lags constraints make the resource constraints comparatively less important to propagate than on classic job-shop or in open-shop. Indeed, Tempo without the NoOverlap propagators has results similar to those of Aries. However, Aries is clearly dominated by CP Optimizer and Tempo on job-shop, where stronger propagation is important.

CP Optimizer is extremely robust. Overall it produces more proofs than any other solver, and is clearly dominated only on open-shop. This is not surprising as CP Optimizer implements a large portfolios of techniques for scheduling problems and is finely tuned on these instances.

OptalCP is extremely fast. However, as discussed previously, it does not have the same adaptive search strategy as CP Optimizer, and hence it can be seen as two solvers. $\text{OptalCP}_{\text{FDS}}$ is the best on open-shop but is clearly dominated by all other solvers on job-shop. On the contrary, $\text{OptalCP}_{\text{LNS}}$ is the best on job-shop, but rarely proves optimality and is overall worse on open-shop and job-shop with time lags (see Table 3).

Tempo seems at least as robust as CP Optimizer, although it does not use any complex strategy to adapt the heuristics and parameters during search. If we include $\text{OptalCP}_{\text{LNS}}$, Tempo is second-best on all three problem types. Moreover, it is consistently within the best solvers in every data set, to the exception of the 10 largest Taillard's job-shop instances, with 20 machines and, crucially, 100 jobs. In other words each of the 20 NoOverlap constraints involve 100 tasks and 4950 edge variables. In Tempo, the Edge-Finding propagator is used unconditionally and without any incremental aspect. As a result, the exploration speed drops dramatically on these instances. Whereas CP Optimizer finds the optimal solution with a proof on all 10 runs for each of these 10 instances, the best solution found by Tempo is very far from optimal. If we ignore those ten instances, the number of proofs is very close for both solvers (478 for CP Optimizer and 476 for Tempo), moreover the gap in average objective value is significantly larger (3300.1 for CP Optimizer and 3273.2 for Tempo).

Table 2 Comparison with state-of-the-art solvers: average CPU Time (CPU Time) and average number of branches (#Branches) after 30 minutes on "easy" instances. Best results are in bold font.

instances	OR-Tools		Aries		CP Optimizer		OptalCP$_{FDS}$		Tempo	
	CPU	#Bran.	CPU	#Bran.	CPU	#Bran.	CPU	#Bran.	CPU	#Bran.
Jobshop Scheduling Problems										
abz(20)	1.11	9236	0.29	4449	1.12	42660	0.33	27681	**0.19**	**2487**
ft(20)	2.97	20036	0.88	8900	2.24	69257	0.46	32846	**0.46**	**3900**
la(380)	20.61	78717	11.56	61186	6.62	190685	8.40	329783	**3.64**	**15764**
orb(100)	11.88	56344	2.72	23874	3.91	134998	1.23	87903	**0.92**	**7197**
rcmax(10)	192.82	577669	5.86	154845	**0.21**	**10522**	850.77	15743276	157.34	376972
swv(50)	46.23	205626	1.79	57660	**0.39**	**15104**	254.06	7267279	36.36	115931
ta(90)	220.20	605827	37.99	144237	90.42	2072631	27.17	745279	**23.01**	**47046**
all(670)	49.49	159276	12.32	64653	16.62	411337	40.28	1079388	**10.37**	**30803**
Openshop Scheduling Problems										
gp(800)	0.51	8316	**0.13**	12687	1.31	53780	0.26	25409	0.19	**5129**
bru(470)	1.77	12197	0.88	8154	3.87	165362	0.39	29342	**0.19**	**2091**
ta(600)	4.17	46528	**0.26**	9497	0.52	15576	3.79	228003	0.91	**6525**
all(1870)	2.00	21552	**0.36**	10524	1.70	69567	1.43	91401	0.42	**4813**
Jobshop Scheduling Problems with Time Lags										
car(320)	1.51	25667	0.30	5192	1.30	53948	0.25	25923	**0.14**	**2007**
ft(150)	12.89	81471	0.63	7446	6.11	192551	1.20	72274	**0.60**	**4510**
la(1140)	50.23	237217	11.86	71476	9.14	323321	4.26	261323	**4.20**	**20275**
all(1610)	37.07	180659	8.52	52336	7.30	257598	3.18	196922	**3.06**	**15175**

8.1.2 Easy instances

Results for easy instances are reported in Table 2. For each data set, the total number of runs is shown next to the data set name, then we report the average CPU time (CPU) and the average number of branching choicepoints explored (#Bran.).

The CPU time results are not easy to interpret. However, the number of explored choicepoints is interesting. Tempo very consistently explores the smallest subtree. Aries does not use costly propagators, and we believe that CP Optimizer and OR-Tools use them more parsimoniously. There are two data sets where Tempo explores more branches: "rcmax" and "swv". However, the number of failures on these instance is still 4 times lower than for CP Optimizer. In fact, Tempo finds very poor initial solutions, and goes through many more improving solutions than CP Optimizer before reaching the optimal. Each one is found after very few failures, but many choicepoints are explored. We believe that LNS is key on these instances as well as on the large Taillard instances discussed earlier.

8.2 Ablation study

The second part of Table 3 gives the result on an ablation study on some aspects of Tempo:

- Tempo$\setminus_{\text{prec}}$ stands for Tempo without the precedence reasoning described in Section 6.
- Tempo$\setminus_{\text{edge-finding}}$ stands for Tempo without the EDGE-FINDING rule described in Section 5.
- Tempo$\setminus_{\text{weakening}}$ stands for Tempo without the numeric literal weakening rule described in Section 7, but with standard clause minimization instead.

The first observation is that the transitivity and temporal graph reasoning does not pay off. It helps on open-shop and with respect to the best objective on job-shop, but actually degrades the performances on job-shop with time lags and decreases the number of optimality

Table 3 Summary of the comparison and ablation study: number of proofs (#Opt.) and average objective value (Objective) after 30 minutes on "hard" instances. Best results are in bold font.

solver	job-shop		open-shop		jsp with time lags	
	#Opt.	Objective	#Opt.	Objective	#Opt.	Objective
OR-Tools	84	3583.8	30	1021.7	71	1778.7
Aries	237	3452.5	**50**	**1021.2**	**474**	**1669.2**
CP Optimizer	**578**	3418.1	21	1024.1	369	1701.2
OptalCP$_{\text{LNS}}$	480	**3391.4**	0	1024.2	70	1724.0
OptalCP$_{\text{FDS}}$	134	3876.1	**50**	**1021.2**	460	1712.7
Tempo	476	3404.9	30	**1021.2**	443	1682.9
Tempo$\setminus_{\text{prec}}$	485	3406.5	30	1021.4	466	1677.6
Tempo$\setminus_{\text{edge-finding}}$	117	3492.9	30	1021.6	460	1678.8
Tempo$\setminus_{\text{weakening}}$	460	3407.1	30	1021.3	446	1683.8

proofs on job-shop. The Edge-Finding rule, however, is clearly useful on job-shop and open-shop, although it also degrades the performance on job-shop with time lags. The bound literal weakening rule has a small but consistently positive impact throughout, however.

Those results are consistent with the comparison between Tempo and Aries: resource constraint propagation has a significant positive impact on standard job-shop, but a negative impact on job-shop with time lags. We believe that in the latter, the temporal graph part is more important and comparatively less propagation comes from resource constraints. This is supported by the results on individual data sets shown in Table 1: Aries is better on la_0, la_0,5 and la_1 where the lags constraints are the strongest, while Tempo is better on la_3 and la_10 which are much closer to standard job-shop.

9 Conclusion

We have introduced Tempo, a hybrid algorithm implementing conflict-driven clause generation on problems defined as constraint programs on Boolean and numeric (bound) variables. Moreover, this solver uses at its core a difference logic engine, useful on temporal problems in particular. The integration of these techniques with resource-based constraint propagation yields a very efficient method for disjunctive scheduling. We believe that the same principles can be applied to cumulative scheduling, to problems considering both allocation and scheduling, to time transition constraints and to a range of other complex problems.

References

1 Joseph Adams, Egon Balas, and Daniel Zawack. The Shifting Bottleneck Procedure for Job Shop Scheduling. *Management Science*, 34(3):391–401, 1988.

2 David Applegate, Rovert Bixby, Vašek Chvátal, and William Cook. Finding Cuts in the TSP (A preliminary report). Technical report, Center for Discrete Mathematics & Theoretical Computer Science, 1995.

3 David Applegate and William Cook. A Computational Study of the Job-Shop Scheduling Problem. *ORSA Journal on Computing*, 3(2):149–156, 1991. doi:10.1287/IJOC.3.2.149.

4 Mokhtar S. Bazaraa and R. W. Langley. A Dual Shortest Path Algorithm. *SIAM Journal on Applied Mathematics*, 26(3):496–501, 1974.

5 J. Christopher Beck. Multi-point Constructive Search. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 737–741, 2005. doi:10.1007/11564751_55.

6 Arthur Bit-Monnot. Aries Git Repository. https://github.com/plaans/aries, 2019.

7 Arthur Bit-Monnot. Enhancing Hybrid CP-SAT Search for Disjunctive Scheduling. In *Proceedings of the 26th European Conference on Artificial Intelligence (ECAI)*, pages 255–262, 2023. doi:10.3233/FAIA230278.

8 Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting Systematic Search by Weighting Constraints. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence (ECAI)*, pages 146–150, 2004.

9 Peter Brucker, Johann Hurink, Bernd Jurisch, and Birgit Wöstmann. A branch & bound algorithm for the open-shop problem. *Discrete Applied Mathematics*, 76(1):43–59, 1997. doi:10.1016/S0166-218X(96)00116-3.

10 Jacques Carlier and Eric Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78(2):146–161, 1994. Project Management and Scheduling.

11 Anthony Caumond, Philippe Lacomme, and Nikolay Tchernev. A memetic algorithm for the job-shop with time-lags. *Computers & Operations Research*, 35(7):2331–2356, 2008. doi:10.1016/J.COR.2006.11.007.

12 Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed Git Repository. https://github.com/chuffed/chuffed, 2016.

13 Scott Cotton and Oded Maler. Fast and Flexible Difference Constraint Propagation for DPLL(T). In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 170–183, 2006. doi:10.1007/11814948_19.

14 Toby O. Davies, Frédéric Didier, and Laurent Perron. ViolationLS: Constraint-Based Local Search in CP-SAT. In *Proceedings of the 21st International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, pages 243–258, 2024. doi:10.1007/978-3-031-60597-0_16.

15 Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. Benchmarks for shop scheduling problems. *European Journal of Operational Research*, 109(1):137–141, 1998. doi:10.1016/S0377-2217(97)00019-2.

16 Thibaut Feydy, Andreas Schutt, and Peter J. Stuckey. Global difference constraint propagation for finite domain solvers. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 226–235, 2008. doi:10.1145/1389449.1389478.

17 Thibaut Feydy and Peter J. Stuckey. Lazy Clause Generation Reengineered. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 352–366, 2009. doi:10.1007/978-3-642-04244-7_29.

18 H. Fisher and G.L. Thompson. Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules. In J.F. Muth and G.L. Thompson, editors, *Industrial Scheduling*, pages 225–251. Prentice-Hall, Englewood Cliffs, 1963.

19 Daniel Frost and Rina Dechter. Dead-End Driven Learning. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*, pages 294–300, 1994. URL: http://www.aaai.org/Library/AAAI/1994/aaai94-045.php.

20 Diarmuid Grimes and Emmanuel Hebrard. Solving Variants of the Job Shop Scheduling Problem Through Conflict-Directed Search. *INFORMS J. Comput.*, 27(2):268–284, 2015. doi:10.1287/IJOC.2014.0625.

21 Diarmuid Grimes, Emmanuel Hebrard, and Arnaud Malapert. Closing the Open Shop: Contradicting Conventional Wisdom. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 400–408, 2009. doi:10.1007/978-3-642-04244-7_33.

22 Christelle Guéret and Christian Prins. A new lower bound for the open-shop problem. *Annals of Operations Research*, 92:165–183, 1999. doi:10.1023/A\%3A1018930613891.

23 Emmanuel Hebrard. Tempo. Software, swhId: swh:1:dir:bbab6415fb8f44fef04878de35e7b50e797596bd (visited on 2025-07-23). URL: https://gitlab.laas.fr/roc/emmanuel-hebrard/tempo, doi:10.4230/artifacts.24099.

24 IBM. IBM ILOG CP Optimizer website. https://www.ibm.com/fr-fr/products/ilog-cplex-optimization-studio/cplex-cp-optimizer.

25 George Katsirelos and Fahiem Bacchus. Unrestricted Nogood Recording in CSP Search. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 873–877, 2003. doi:10.1007/978-3-540-45193-8_70.

26 George Katsirelos and Fahiem Bacchus. Generalized nogoods in csps. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI)*, pages 390–396, 2005. URL: http://www.aaai.org/Library/AAAI/2005/aaai05-062.php.

27 Philippe Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artif. Intell.*, 143(2):151–188, 2003. doi:10.1016/S0004-3702(02)00362-4.

28 Philippe Laborie and Danièle Godard. Self-Adapting Large Neighborhood Search: Application to Single-Mode Scheduling Problems. In *Proceedings of MISTA*, 2007.

29 Philippe Laborie, Jerome Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling - 20+ years of scheduling with constraints at IBM/ILOG. *Constraints An Int. J.*, 23(2):210–250, 2018. doi:10.1007/S10601-018-9281-X.

30 S. Lawrence. *An Experimental Investigation of Heuristic Scheduling Techniques (Supplement)*. PhD thesis, Carnegie Mellon University, 1984.

31 Christophe Lecoutre. ACE Git Repository. https://github.com/xcsp3team/ACE, 2024.

32 Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Nogood recording from restarts. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 131–136, 2007. URL: http://ijcai.org/Proceedings/07/Papers/019.pdf.

33 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for sat solvers. In *Proceedings of the 19th International conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 123–140, 2016. doi:10.1007/978-3-319-40970-2_9.

34 J.P. Marques-Silva and K.A. Sakallah. GRASP – A new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 220–227, 1996.

35 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)*, pages 530–535, 2001. doi:10.1145/378239.379017.

36 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = Lazy Clause Generation. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 544–558, 2007. doi:10.1007/978-3-540-74970-7_39.

37 Laurent Perron, Frédéric Didier, and Steven Gay. OR-Tools Git Repository. https://github.com/google/or-tools, 2016.

38 Laurent Perron, Frédéric Didier, and Steven Gay. The CP-SAT-LP Solver. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 3:1–3:2, 2023.

39 Diego Olivier Fernandez Pons and Petr Vilím. OptalCP website. https://optalcp.com, 2023.

40 Charles Prud'homme and Jean-Guillaume Fages. Choco-solver: A Java library for constraint programming. *Journal of Open Source Software*, 7(78):4708, 2022. doi:10.21105/JOSS.04708.

41 Charles Prud'homme and Jean-Guillaume Fages. Choco-solver website. https://choco-solver.org, 2022.

42 Philippe Refalo. Impact-Based Search Strategies for Constraint Programming. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 557–571, 2004. doi:10.1007/978-3-540-30201-8_41.

43 Thomas Schiex and Gérard Verfaillie. Nogood Recording for static and dynamic constraint satisfaction problems. In *Proceedings of 1993 IEEE Conference on Tools with AI (ICTAI)*, pages 48–55, 1993. doi:10.1109/TAI.1993.633935.

44 Christian Schulte, Guido Tack, and Mikael Zayenz Lagerkvist. Gecode website. https://www.gecode.org/index.html, 2002.

45 Andreas Schutt, Thibaut Feydy, and Peter J. Stuckey. Scheduling Optional Tasks with Explanation. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 628–644, 2013. doi:10.1007/978-3-642-40627-0_47.

46 Andreas Schutt and Peter J. Stuckey. Explaining Producer/Consumer Constraints. In *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP)*, pages 438–454, 2016. doi:10.1007/978-3-319-44953-1_28.

47 Paul Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 417–431, 1998. doi:10.1007/3-540-49481-2_30.

48 Mohamed Siala, Christian Artigues, and Emmanuel Hebrard. Two Clause Learning Approaches for Disjunctive Scheduling. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP)*, pages 393–402, 2015. doi:10.1007/978-3-319-23219-5_28.

49 Robert H. Storer, S. David Wu, and Renzo Vaccari. New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling. *Management Science*, 38(10):1495–1509, 1992.

50 E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.

51 Petr Vilím. O(n log n) Filtering Algorithms for Unary Resource Constraint. In *Proceedings of the 1st International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, pages 335–347, 2004. doi:10.1007/978-3-540-24664-0_23.

52 Petr Vilím. *Global Constraints in Scheduling*. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, KTIML MFF, Universita Karlova, Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic, August 2007. URL: http://vilim.eu/petr/disertace.pdf.

53 Petr Vilím. CP solver design for maximum CPU utilization (invited talk). In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 5:1–5:1, 2023. doi:10.4230/LIPICS.CP.2023.5.

54 Petr Vilím, Philippe Laborie, and Paul Shaw. Failure-Directed Search for Constraint-Based Scheduling. In *Proceedings of the 12th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, pages 437–453, 2015. doi:10.1007/978-3-319-18008-3_30.

55 Takeshi Yamada and Ryohei Nakano. A Genetic Algorithm Applicable to Large-Scale Job-Shop Problems. In *Parallel Problem Solving from Nature*, 1992.

A Appendix

A.1 Proof of Theorem 1

Proof. We prove by induction on the size of Ω, for length 1 the node is a leaf and we have $est^*_{\{i\}} = est_i$ and by definition $ect_i = est_i + p_i$.

Now suppose that the equality is true for sets of size $n-1$ and consider a node labeled with the set Ω such that $|\Omega| = n+1$. Moreover, let Θ_l and Θ_r be its left and right children, respectively.

Suppose first that $ect_{\Theta_r} \geq ect_{\Theta_l} + p_{\Theta_r}$. Then we have

- $ect_\Omega = ect_{\Theta_r}$ and

- $est^*_\Omega = est^*_{\Theta_r}$ and
- $p_{expl_\Omega} = p_{expl_{\Theta_r}}$ (since every task in Θ_l has an earliest start time strictly smaller than $est^*_{\Theta_r}$)

However, since $|\Theta_r| \leq n$ we have $est^*_{\Theta_r} + p_{expl_{\Theta_r}} = ect_{\Theta_r}$ and therefore $est^*_\Omega + p_{expl_\Omega} = ect_\Omega$.

Otherwise, if $ect_{\Theta_r} < ect_{\Theta_l} + p_{\Theta_r}$. Then we have

- $ect_\Omega = ect_{\Theta_l} + p_{\Theta_r}$ and
- $est^*_\Omega = est^*_{\Theta_l}$ and
- $p_{expl_\Omega} = p_{expl_{\Theta_l}} + p_{\Theta_r}$

However, since $|\Theta_l| \leq n$ we have $est^*_{\Theta_l} + p_{expl_{\Theta_l}} = ect_{\Theta_l}$, and hence $est^*_{\Theta_l} + p_{expl_{\Theta_l}} + p_{\Theta_r} = ect_{\Theta_l} + p_{\Theta_r}$ and therefore $est^*_\Omega + p_{expl_\Omega} = ect_\Omega$. ◀

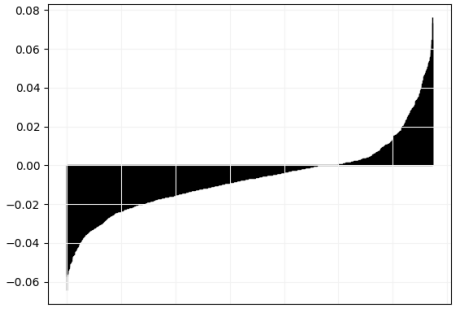
A.2 Description of the data sets

Table 4 List of data sets, "Size" is the number of instances in the data set, the column "#jobs", "#machines" and "Avg. duration" give the minimum and maximum number of jobs, number of machines, and average duration of task, respectively.

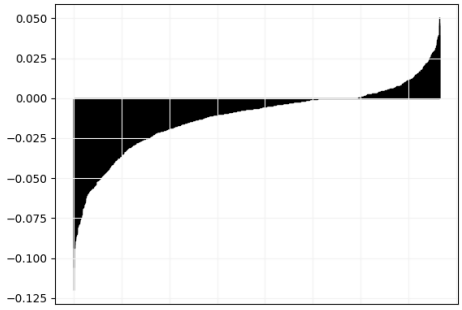
Dataset	Type	Size	#jobs	#machines	Avg. duration	Reference
abz	Job-shop	5	10-20	10-15	24-77	[1]
cscmax	Job-shop	40	20-50	15-20	95-104	[15]
rcmax	Job-shop	40	20-50	15-20	92-105	[15]
ft	Job-shop	3	6-20	5-10	5-51	[18]
la	Job-shop	40	10-30	5-15	45-56	[30]
orb	Job-shop	10	10-10	10-10	24-56	[3]
swv	Job-shop	20	20-50	10-15	48-52	[49]
ta	Job-shop	80	15-100	15-20	47-52	[50]
yn	Job-shop	4	20-20	20-20	29-30	[55]
bru	Open-shop	42	4-8	4-8	110-333	[9]
gp	Open-shop	80	3-10	3-10	100-333	[22]
ta	Open-shop	60	4-20	4-20	41-65	[50]
car	Job-shop w. Lags	36	7-10	6-9	483-498	[11]
ft	Job-shop w. Lags	18	6-10	6-10	5-51	[11]
la	Job-shop w. Lags	233	10-30	5-15	45-56	[11]

A.3 Pairwise comparisons

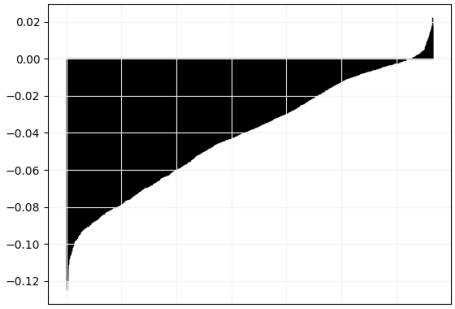
We plot in Fig. 3 and 4 pairwise comparisons of relative difference in objective value between Tempo and all other solvers. For every instance i in the "hard" category, let $obj_{\mathsf{Tempo}}(i)$ be the objective value found by Tempo and $obj_{\mathsf{other}}(i)$ the objective value found by the other solver, we plot the values of $(obj_{\mathsf{Tempo}}(i) - obj_{\mathsf{other}}(i))/\max(obj_{\mathsf{Tempo}}(i), obj_{\mathsf{other}}(i))$ in non-decreasing order.



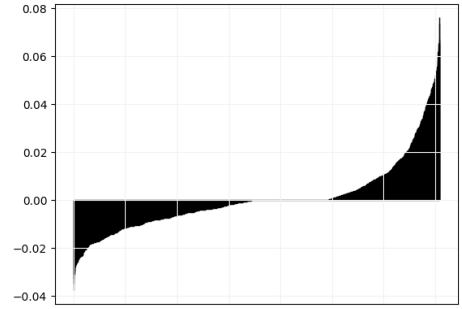
(a) (Tempo- CP Optimizer)/max.



(b) (Tempo- Aries)/max.

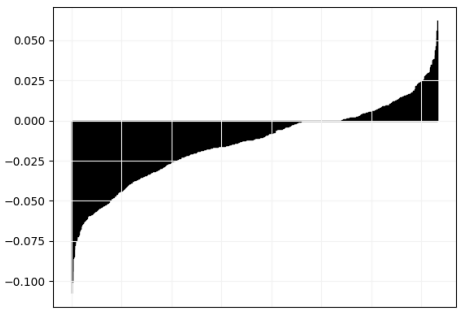


(c) (Tempo- OR-Tools)/max.

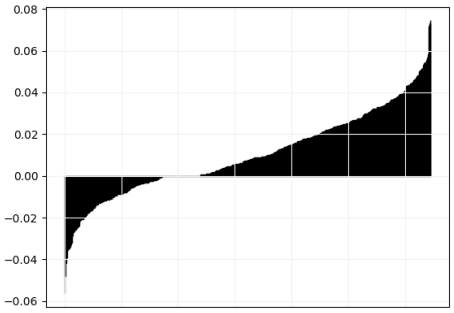


(d) (Tempo- OptalCP$_{\text{LNS}}$)/max.

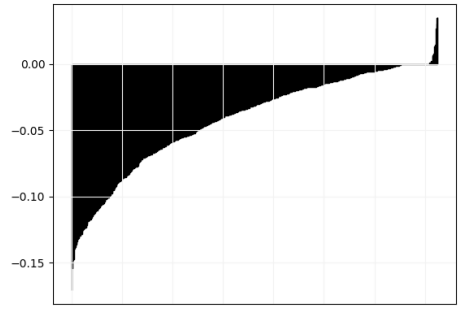
Figure 3 Pairwise comparison of objective value on Job-shop scheduling.



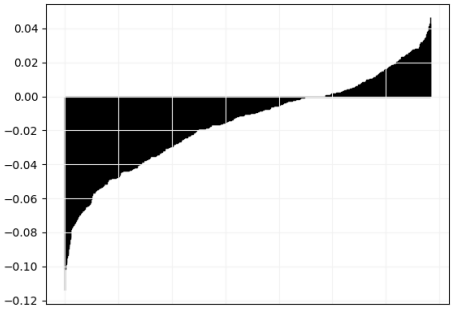
(a) (Tempo- CP Optimizer)/max.



(b) (Tempo- Aries)/max.



(c) (Tempo- OR-Tools)/max.



(d) (Tempo- OptalCP$_{\mathsf{FDS}}$)/max.

Figure 4 Pairwise comparison of objective value on Job-shop scheduling with time lags.

Guess and Prove: A Hybrid Approach to Linear Polynomial Recovery in Circuit Verification

Clemens Hofstadler
Johannes Kepler University, Linz, Austria

Daniela Kaufmann
TU Wien, Austria

Abstract

Formal verification of arithmetic circuits using computer algebra has been shown to be highly successful. The circuit is encoded as a system of polynomials, which automatically generates a lexicographic Gröbner basis. Correctness is then verified by computing the polynomial remainder of the specification. To optimize the remainder computation, prior work extracts linear polynomials. However, this required recomputing a Gröbner basis with respect to a degree-compatible order.

In this paper, we show that this computationally expensive step is unnecessary and propose a novel hybrid verification approach that combines an FGLM-style linearization technique with a guess-and-prove method using SAT solving to derive the linear relations directly from lexicographic Gröbner bases. We enhance our approach using caching techniques and propagating vanishing monomials. Our experimental results demonstrate that our method significantly outperforms previous linearization techniques.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Computer Algebra, FGLM, And-Inverter Graphs, Hardware Verification

Digital Object Identifier 10.4230/LIPIcs.CP.2025.14

Supplementary Material *Software (Source Code)*: `https://github.com/d-kfmnn/talisman/tree/be8187d` [9], archived at `swh:1:dir:8af022a61661d2b7fb281abbdc2f18d51f221c20`

Funding *Clemens Hofstadler*: LIT AI Lab funded by the state of Upper Austria.
Daniela Kaufmann: Austrian Science Fund (FWF) [10.55776/ESP666].

Acknowledgements We thank Jérémy Berthomieu and the reviewers for insightful comments.

1 Introduction

Computer algebra-based formal verification techniques have proven to be highly effective in verifying gate-level arithmetic circuits. As digital systems become more complex, ensuring the accuracy of arithmetic circuits is essential, particularly in safety-critical fields like cryptography and signal processing, where even minor errors can lead to significant consequences. Formal verification approaches on the word-level based on theorem provers [25] or computer algebra [13, 15, 17, 18], particularly those leveraging Gröbner bases [4], have demonstrated significant advancements in recent years.

These methods encode circuits given as and-inverter graphs (AIGs) [19] as polynomial systems where the coefficients are integers and the variables represent Boolean values. The terms are sorted according to a reverse topological term order [20]. This ensures that the output variable of a gate precedes its input variables, which results in an encoding where all the leading terms are disjoint. Hence, the set of gate polynomials automatically generates a Gröbner basis with respect to a lexicographic term order. This fact admits verification to be performed by simply computing the remainder of the specification polynomial modulo the Gröbner basis. The circuit is correct if and only if the final remainder is zero [16].

© Clemens Hofstadler and Daniela Kaufmann;
licensed under Creative Commons License CC-BY 4.0

However, a major bottleneck of using a lexicographic term order is the significant growth of intermediate reduction results. This phenomenon, known as *monomial blow-up*, occurs because the tails of the polynomials in the lexicographic Gröbner basis have higher degrees than their leading terms, and can easily lead to intermediate reduction results with millions of monomials [21]. To address this challenge, various preprocessing and rewriting algorithms have been developed, which syntactically or semantically analyze the input circuit to remove redundant information and choose a suitable reduction order.

Several advanced reduction engines that use a lexicographic order have been developed, including DynPhaseOrderOpt [17], DyPoSub [22], and AMulet2 [14, 15], along with its variant TeluMA [12]. The work in [14, 15] employs SAT solving to rewrite parts of the multiplier before applying an incremental column-wise verification algorithm. A subsequent refinement in [12] eliminated the external SAT solver by using a sophisticated algebraic encoding that includes polarities of literals. In [22], a dynamic rewriting approach is used that determines the reduction order on-the-fly and backtracks if the size of the intermediate reductions exceeds a predefined threshold. Building on this, [17] introduced an improved method that incorporates mixed signals into the encoding.

Recent work [13] diverges from the lexicographic order and opts instead for a degree-compatible order to ensure that the degree of intermediate reduction results cannot increase during reduction. By first linearizing the specification polynomial, the entire reduction is restricted to linear polynomials. However, this approach requires extracting linear polynomials from the circuit, which is not straightforward. The authors proposed an on-the-fly technique that identifies subcircuits and computes degree-compatible Gröbner bases only for them. The required linear polynomials for reduction are supposed to be contained in the individually computed Gröbner bases.

Despite its innovation, the approach of [13] has notable limitations. First, it disregards that the polynomials already form a lexicographic Gröbner basis, thereby missing important structural advantages. Second, the method relies on an off-the-shelf computer algebra system (CAS), which requires explicitly encoding that all variables are Boolean, missing out on optimizations provided by modified polynomial operations. Third, whenever the initial subcircuit selection is inadequate, the subcircuit must be iteratively expanded, leading to repeated incremental Gröbner basis computations. Since CASs cannot recognize that parts of the input already generate a partial Gröbner basis, expensive computations must be repeated. Our work aims to address these shortcomings.

In this work, we introduce a novel method for extracting linear polynomials from subcircuits. Our hybrid approach combines an *FGLM-based linear extraction* algorithm with a *guess-and-prove-style linearization* technique.

In the *FGLM-based algorithm*, we initially compute the normal forms of single variables with respect to the lexicographic Gröbner basis. The linear polynomials are then recovered by determining the kernel of the corresponding coefficient matrix, which ensures that the linear extractions are *proven* to be correct. This algorithm can be considered as the first steps of the change-of-order FGLM-algorithm [8] for converting a Gröbner basis w.r.t. one term order into one for a different order. The *guess-and-prove linearization* approach involves sampling input values for the subcircuit. By solving a linear system of equations generated from these samples, we can identify potentially valid linear polynomials. We *prove* the correctness of the *guessed* polynomials using satisfiability (SAT) solving. To repair incorrect guesses, we exploit information obtained from the SAT solving process.

Despite taking advantage of properties that are characteristic for circuit verification, we highlight that our methods are not restricted to this application and can be used in any context that meets these characteristics. Therefore, we introduce the FGLM and guess-and-prove algorithms in a general setting in Section 3, before discussing how to specialize them for circuit verification in Section 4.

We further enhance our method by reusing computations for isomorphic subcircuits, and by generating and propagating vanishing monomials, see Section 4.1. We implement our techniques in a new tool, called TalisMan. Our experimental evaluation (Section 5) shows that we significantly outperform existing linearization techniques.

2 Preliminaries

In Section 2.1, we recall basic concepts about polynomials and Gröbner bases that will be needed for the rest of this work. For further information, we refer to the great introductory books [5, 6], which also served as basis for our presentation. Section 2.2 introduces AIGs and how they can be encoded into polynomials.

2.1 Polynomials and Gröbner Bases

Fix a finite set $X = \{x_1, \dots, x_n\}$ of indeterminates and let $\mathbb{K}$ be a field.

▶ **Definition 1** (Monomials). *A* monomial *(in X) is a product of the form $x_1^{\alpha_1} \dots x_n^{\alpha_n}$ with exponents $\alpha_1, \dots, \alpha_n \in \mathbb{N}$. The set of all monomials in X is denoted by $[X]$. The* (total) degree $\deg(m)$ *of a monomial $m = x_1^{\alpha_1} \dots x_n^{\alpha_n} \in [X]$ is the sum of exponents, i.e.,* $\deg(m) = \alpha_1 + \dots + \alpha_n$.

▶ **Definition 2** (Polynomials). *A* polynomial *f (over $\mathbb{K}$) is a finite $\mathbb{K}$-linear combination of monomials, that is, $f = c_1 m_1 + \dots + c_s m_s$, with* coefficients *$c_1, \dots, c_s \in \mathbb{K}$ and $m_1, \dots, m_s \in [X]$. The set of all polynomials is denoted by $\mathbb{K}[X] = \mathbb{K}[x_1, \dots, x_n]$. A polynomial f is* linear *if it is of the form $f = c_0 + c_1 x_1 + \dots + c_n x_n$ with $c_0, \dots, c_n \in \mathbb{K}$.*

For a polynomial $f = c_1 m_1 + \dots + c_s m_s \in \mathbb{K}[X]$, we call the product $c_i m_i$ a *term* of f, for $i = 1, \dots s$. In this work, we consider all polynomials to be in canonical form, which means that terms with equal monomials are merged by adding their coefficients and terms with coefficient zero are omitted. Furthermore, as we will only consider polynomial equations with right hand side zero, we will also write "f" instead of "$f = 0$".

Many algebraic operations (such as division) require the terms within a polynomial to be sorted. This is achieved by a *monomial order*.

▶ **Definition 3** (Monomial Order). *A total order $\prec$ on $[X]$ is a* monomial order *if it satisfies:*

1. *$m_1 \prec m_2$ implies $m \cdot m_1 \prec m \cdot m_2$ for all $m, m_1, m_2 \in [X]$;*
2. *every nonempty subset of $[X]$ has a smallest element;*

▶ **Example 4.** We list two classical examples of monomial orders. Let $m_1 = x_1^{\alpha_1} \dots x_n^{\alpha_n}$ and $m_2 = x_1^{\beta_1} \dots x_n^{\beta_n}$ be monomials.

1. *Lexicographic order*: We say that $m_1 \prec_{\text{lex}} m_2$ if $\alpha_i < \beta_i$ for the smallest index $i \in \{1, \dots, n\}$ where $\alpha_i \neq \beta_i$.
2. *Degree lexicographic order*: We say that $m_1 \prec_{\text{dlex}} m_2$ if $\deg(m_1) < \deg(m_2)$ or if $\deg(m_1) = \deg(m_2)$ and $m_1 \prec_{\text{lex}} m_2$.

The degree lexicographic order is an instance of a *degree-compatible* (or *graded*) order. A monomial order $\prec$ is *degree-compatible* if $\deg(m_1) < \deg(m_2)$ implies $m_1 \prec m_2$ for all monomials $m_1, m_2 \in [X]$.

With respect to a fixed monomial order $\prec$, we can identify the unique maximal monomial in each nonzero polynomial $f \in \mathbb{K}[X]$. This monomial is called the *leading monomial* of f and denoted by $\operatorname{lm}(f)$. The coefficient of $\operatorname{lm}(f)$ is called the *leading coefficient* of f, denoted by $\operatorname{lc}(f)$, and the term $\operatorname{lc}(f)\operatorname{lm}(f)$ is called the *leading term* of f, denoted by $\operatorname{lt}(f)$. The *tail* of f is $\operatorname{tail}(f) = f - \operatorname{lt}(f)$.

▶ **Definition 5** (Ideal). *A nonempty subset $I \subseteq \mathbb{K}[X]$ is an* ideal *if it is closed under addition, that is, $f + g \in I$ for all $f, g \in I$, and if it is closed under multiplication by arbitrary polynomials, that is, $hf \in I$ for all $f \in I$ and $h \in \mathbb{K}[X]$.*

A set of polynomials $F = \{f_1, \ldots, f_r\} \subseteq \mathbb{K}[X]$ can be considered as a system of equations $f_1 = \cdots = f_r = 0$. The set of all polynomials h for which the equation $h = 0$ can be derived algebraically from this system (by adding equations and by multiplying an equation by a polynomial) forms an ideal, which is denoted by $\langle F \rangle$. It is given explicitly by $\langle F \rangle = \{h_1 f_1 + \cdots + h_r f_r \mid h_1, \ldots, h_r \in \mathbb{K}[X]\}$.

A central result in (computational) algebra, known as *Hilbert's Basis Theorem* [5, Thm. 2.5.4], says that every ideal $I \subseteq \mathbb{K}[X]$ can be written as $I = \langle F \rangle$ for some finite set $F \subseteq \mathbb{K}[X]$. The set F is called a *basis* of I and we say that I *is generated by* F.

In general, an ideal has many bases. Given an arbitrary basis F of an ideal, it is a difficult question to determine whether an equation $h = 0$ can be derived from the system induced by F, or in algebraic terms, to determine whether $h \in \langle F \rangle$. The latter problem is known as the *ideal membership problem*. In principle, it can be solved by repeatedly dividing h by the elements in the basis F until a remainder is obtained that cannot be divided further. See [5, Thm. 2.3.3] for a precise description of this division process. However, this remainder is typically not unique but it depends on the order of divisions. Only for certain particular bases of an ideal a unique remainder can be obtained. These bases are called *Gröbner bases* and they can be used to solve the ideal membership problem. The following definition depends on a monomial order; we assume that some order $\prec$ has been fixed.

▶ **Definition 6** (Gröbner Basis). *A basis G of an ideal $I \subseteq \mathbb{K}[X]$ is a* Gröbner basis *of I (w.r.t. $\prec$) if every polynomial $f \in \mathbb{K}[X]$ has a unique remainder under division by G. This unique remainder is denoted by $\operatorname{NF}_G(f)$ and is also called the* normal form *of f.*

An equivalent characterisation of a Gröbner basis G of I is that $f \in I$ if and only if $\operatorname{NF}_G(f) = 0$, see [5, Cor. 3.6.2]. Moreover, it can be shown that every ideal has a finite Gröbner basis. A Gröbner basis of an ideal I can be computed starting from any finite basis of I by a completion procedure that can be considered as an algebraic analogue of the Knuth–Bendix completion algorithm. This completion procedure is known as *Buchberger's algorithm* [4] and, in contrast to Knuth–Bendix, is guaranteed to terminate for any input.

A polynomial $f \in \mathbb{K}[X] = \mathbb{K}[x_1, \ldots, x_n]$ *vanishes* on a set of points $P \subseteq \mathbb{K}^n$ if $f(a_1, \ldots, a_n) = 0$ for all $(a_1, \ldots, a_n) \in P$. For an ideal $I \subseteq \mathbb{K}[X]$, the set of all points on which all polynomials in I vanish is called the *(algebraic) variety* of I, denoted by $V(I)$, i.e., $V(I) = \{(a_1, \ldots, a_n) \in \mathbb{K}^n \mid f(a_1, \ldots, a_n) = 0 \text{ for all } f \in I\}$. If $V(I)$ is finite, then I is called *zero-dimensional*.

Gate Polynomials $G(C)$	Logical constraint
$s_3 - g_{24}$	$s_3 = g_{24}$
$s_2 - g_{28}$	$s_2 = g_{28}$
$g_{28} - g_{26}g_{24} + g_{26} + g_{24} - 1$	$g_{28} = \neg g_{26} \wedge \neg g_{24}$
$g_{26} - g_{22}g_{16} + g_{22} + g_{16} - 1$	$g_{26} = \neg g_{22} \wedge \neg g_{16}$
$g_{24} - g_{22}g_{16}$	$g_{24} = g_{22} \wedge g_{16}$
$g_{22} - b_1a_1$	$g_{22} = b_1 \wedge a_1$
$s_1 - g_{20}$	$s_1 = g_{20}$
$g_{20} - g_{18}g_{16} + g_{18} + g_{16} - 1$	$g_{20} = \neg g_{18} \wedge \neg g_{16}$
$g_{18} - g_{14}g_{12} + g_{14} + g_{12} - 1$	$g_{18} = \neg g_{14} \wedge \neg g_{12}$
$g_{16} - g_{14}g_{12}$	$g_{16} = g_{14} \wedge g_{12}$
$g_{14} - b_1a_0$	$g_{14} = b_1 \wedge a_0$
$g_{12} - b_0a_1$	$g_{12} = b_0 \wedge a_1$
$s_0 - g_{10}$	$s_0 = g_{10}$
$g_{10} - b_0a_0$	$g_{10} = b_0 \wedge a_0$

$B(C)$: $a_1^2 - a_1, a_0^2 - a_0, b_1^2 - b_1, b_0^2 - b_0$

Spec $\mathcal{S}$: $8s_3 + 4s_2 + 2s_1 + s_0 - 4a_1b_1 - 2a_1b_0 - 2a_0b_1 - a_0b_0$

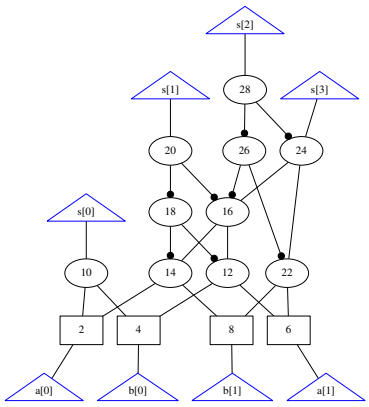


Figure 1 AIG and polynomial encoding of a 2-bit multiplier in the ring $\mathbb{Q}[X]$.

2.2 Polynomial Encodings of And-Inverter Graphs

An AIG [19] is a directed acyclic graph that can be used to represent Boolean functions and logic circuits in a compact way. The nodes (also called *gates*) in an AIG correspond to logical conjunction. The edges represent signal propagation, with optional markers indicating negation. The *primary inputs* of the AIG represent Boolean variables.

▶ **Definition 7** (Specification). *The specification of an AIG is a polynomial $\mathcal{S} \in \mathbb{K}[X]$ that relates the outputs of an AIG to its primary inputs.*

While the AIG performs logical operations on Boolean variables, the specification is not confined to the Boolean ring $\mathbb{B}[X]$. Instead, it can extend to alternative coefficient domains, such as $\mathbb{Q}[X]$, depending on the underlying AIG.

▶ **Definition 8** (Gate Polynomials $G(C)$). *We map $\top \mapsto 1$ and $\bot \mapsto 0$ to derive the relations $a \wedge b \cong ab$ and $\neg a \cong 1 - a$ between logic and algebra. Let g be an AIG node with inputs a, b:*

Logical constraint		*Gate polynomial*
$g = a \wedge b$	$\Rightarrow$	$g - ab$
$g = \neg a \wedge b$	$\Rightarrow$	$g - (1 - a)b = g + ab - b$
$g = a \wedge \neg b$	$\Rightarrow$	$g - a(1 - b) = g + ab - a$
$g = \neg a \wedge \neg b$	$\Rightarrow$	$g - (1 - a)(1 - b) = g - ab + b + a - 1$

For a given AIG C, we collect all its gate polynomials in the set $G(C)$.

Since the specification is not restricted to the Boolean ring, we need to add further polynomials that restrict the variables to the Boolean domain.

▶ **Definition 9** (Boolean Value Polynomials $B(C)$). *For every primary input a_i of the AIG we define a* Boolean value polynomial $a_i(a_i - 1) = a_i^2 - a_i = 0$ *that encodes that the variable can only take the values 0 and 1. Let $B(C)$ denote the set of Boolean value polynomials.*

It suffices to define the Boolean value polynomials only for the primary inputs, as they propagate, i.e., $\forall x_i \in X : x_i^2 - x_i \in \langle G(C) \cup B(C) \rangle$, with X being the set of all output, gate, and input variables [16].

▶ **Example 10.** Figure 1 shows an AIG representing a 2-bit multiplier and its corresponding polynomial encoding. We denote the primary inputs by a_0, a_1, b_0, b_1 and outputs by s_0, s_1, s_2, s_3. The internal nodes are denoted by g_i, where i corresponds to the respective AIG node. The specification $\mathcal{S}$ expresses that the weighted sum of output bits (= output bit-vector) is equal to the product of the weighted sum of input bits (= input bit-vectors).

The correctness of the AIG can be shown by deriving that $\mathcal{S}$ is implied by the polynomial encoding, which algebraically means that we want to show that $\mathcal{S} \in \langle G(C) \cup B(C) \rangle$ [16]. Since the terms are sorted lexicographically according to a reverse topological variable order, all leading terms are disjoint. Hence the polynomials automatically generate a Gröbner basis. Thus, verification can be performed by computing the remainder of $\mathcal{S}$ modulo $G(C) \cup B(C)$. The circuit is correct if and only if the final remainder is zero [16].

3 Finding Linear Polynomials in Ideals

A key step in our approach to circuit verification is identifying linear polynomials in a given ideal. In this section, we propose two methods that recover linear polynomials by exploiting essential properties of the ideals in our application.

We highlight that, while these techniques are designed to take advantage of these properties, they are not specific to our application and can be used in any context that meets our assumptions about the ideals involved. Formally, we consider the following problem over a (computable) field $\mathbb{K}$:

> **Given:** an ideal $I \subseteq \mathbb{K}[X]$
> **Compute:** the set $\mathbb{L}(I)$ of all linear polynomials in I

Note that $\mathbb{L}(I)$ is non-empty because it always contains the zero polynomial. Moreover, if $p, q \in \mathbb{L}(I)$, then so are $p + q$ and cp for every $c \in \mathbb{K}$. Thus, we obtain the following result.

▶ **Lemma 11.** *The set $\mathbb{L}(I)$ of all linear polynomials of an ideal $I \subseteq \mathbb{K}[X]$ forms a finite-dimensional vector space over $\mathbb{K}$ of dimension at most $|X| + 1$.*

Proof. We have already seen that $\mathbb{L}(I)$ is a vector space. The dimension result follows from the fact that $\mathbb{L}(I)$ is a subspace of the $(|X| + 1)$-dimensional vector space $\mathbb{L}(\mathbb{K}[X])$, which has the basis $X \cup \{1\}$. ◀

By "computing" the set $\mathbb{L}(I)$, we mean determining a $\mathbb{K}$-vector space basis. In the general case, when the ideal I is given by an arbitrary basis without additional structural properties, the only approach we are aware of to obtain this basis for $\mathbb{L}(I)$ is by computing a Gröbner basis of I w.r.t. a degree-compatible monomial order. By [13, Thm. 1], such a Gröbner basis contains the desired basis as a subset. This was also the approach used in [13]. However, in the application of circuit verification, the given basis has a lot more structure:

1. It forms a Gröbner basis w.r.t. a lexicographic monomial order.
2. The lexicographic Gröbner basis enables easy computation (of a subset) of $V(I)$.

A naïve computation of a Gröbner basis w.r.t. a degree-compatible order using a black-box computation completely ignores this existing structure.

3.1 FGLM-style Linearization

While the lexicographic order is not degree-compatible, we can still exploit the fact that we already have a Gröbner basis w.r.t. this order when computing a basis of $\mathbb{L}(I)$. More generally, the method presented in this section works with arbitrary Gröbner bases. Thus, we consider the following refined problem:

Algorithm 1 FGLM-style-Linearization.

Input : Gröbner basis G of an ideal $I \subseteq \mathbb{K}[X] = \mathbb{K}[x_1, \ldots, x_n]$
Output : a $\mathbb{K}$-vector space basis L of $\mathbb{L}(I)$
Compute the normal forms $\mathrm{NF}_G(1), \mathrm{NF}_G(x_1), \ldots, \mathrm{NF}_G(x_n)$;
Set up the coefficient matrix

$$A \leftarrow \begin{pmatrix} | & | & & | \\ \mathrm{NF}_G(1) & \mathrm{NF}_G(x_1) & \cdots & \mathrm{NF}_G(x_n) \\ | & | & & | \end{pmatrix}$$

$b_1, \ldots, b_k \leftarrow$ a basis of $\ker(A)$;
return $L \leftarrow \{b_{i,0} + b_{i,1}x_1 + \cdots + b_{i,n}x_n \mid b_i = (b_{i,0}, b_{i,1}, \ldots, b_{i,n}), i = 1, \ldots, k\}$;

Given: an ideal $I \subseteq \mathbb{K}[X]$ and a Gröbner basis G of I
Compute: the set $\mathbb{L}(I)$ of all linear polynomials in I

Our method is based on the classical fact that if G is a Gröbner basis of an ideal (w.r.t. any monomial order), then the normal form map $f \mapsto \mathrm{NF}_G(f)$ is linear, i.e., $\mathrm{NF}_G(cf + dg) = c\,\mathrm{NF}_G(f) + d\,\mathrm{NF}_G(g)$, for all $c, d \in \mathbb{K}$ and $f, g \in \mathbb{K}[X]$, cf. [5, Ex. 2.6.13]. We use this linearity to compute a basis of $\mathbb{L}(I)$.

▶ **Lemma 12.** *Let G be a Gröbner basis of an ideal $I \subseteq \mathbb{K}[X] = \mathbb{K}[x_1, \ldots, x_n]$. A linear polynomial $f = c_0 + c_1x_1 + \cdots + c_nx_n$ with $c_0, \ldots, c_n \in \mathbb{K}$ is contained in I if and only if*

$$c_0\,\mathrm{NF}_G(1) + c_1\,\mathrm{NF}_G(x_1) + \cdots + c_n\,\mathrm{NF}_G(x_n) = 0.$$

Proof. Follows from the linearity of the map $f \mapsto \mathrm{NF}_G(f)$ and the fact that $f \in I$ if and only if $\mathrm{NF}_G(f) = 0$. ◀

Lemma 12 outlines a procedure for computing a basis of $\mathbb{L}(I)$: First, compute the normal forms $\mathrm{NF}_G(1), \mathrm{NF}_G(x_1), \ldots, \mathrm{NF}_G(x_n)$ using the Gröbner basis G. Then find all $\mathbb{K}$-linear relations between these normal forms, which can be achieved as follows: Form the coefficient matrix A whose columns contain the coefficients of $\mathrm{NF}_G(1), \mathrm{NF}_G(x_1), \ldots, \mathrm{NF}_G(x_n)$. More precisely, we assign to each row A_i of A a monomial m_i that appears in the normal forms. The entry $A_{i,j}$ is then given by the coefficient of m_i in $\mathrm{NF}_G(x_j)$ (with $x_0 = 1$). With this, a basis of the kernel $\ker(A)$ of A yields a basis of $\mathbb{L}(I)$ by translating any basis vector $c = (c_0, c_1, \ldots, c_n) \in \ker(A)$ into the linear polynomial $c_0 + c_1x_1 + \cdots + c_nx_n \in \mathbb{L}(I)$. Recall that a basis of $\ker(A)$ can be computed by Gaussian elimination, which has a cubic computational complexity in the matrix size; see, e.g., [24, Sec. 7.1]. These steps are summarized in Algorithm 1.

▶ **Proposition 13.** *Algorithm 1 terminates and is correct.*

Proof. Termination is clear. For correctness, note that Lemma 12 implies that there is a one-to-one correspondence between linear polynomials $c_0 + c_1x_1 + \cdots + c_nx_n \in \mathbb{L}(I)$ and vectors $(c_0, c_1, \ldots, c_n) \in \ker(A)$. Thus, any basis of the latter can be translated into a basis of the former (and vice versa). ◀

▶ **Example 14.** Consider the ideal $I \subseteq \mathbb{Q}[a, b, g_1, g_2, g_3, g_4]$ generated by the four polynomials

$$g_1 - ab, \qquad g_2 - (1-a)(1-b), \qquad g_3 - a(1-b), \qquad g_4 - (1-g_1)(1-g_2),$$

as well as by the two Boolean value polynomials $a^2 - a$ and $b^2 - b$. These six polynomials form a Gröbner basis of I w.r.t. the lexicographic order where $a \prec b \prec g_1 \prec g_2 \prec g_3 \prec g_4$.

We use Algorithm 1 to compute a $\mathbb{Q}$-vector space basis of $\mathbb{L}(I)$. To this end, we compute the normal forms of all six variables and of 1 w.r.t. the given Gröbner basis and set up the corresponding coefficient matrix A. This yields

$$A = \begin{array}{c} \begin{matrix} \mathrm{NF}_G(1) & \mathrm{NF}_G(a) & \mathrm{NF}_G(b) & \mathrm{NF}_G(g_1) & \mathrm{NF}_G(g_2) & \mathrm{NF}_G(g_3) & \mathrm{NF}_G(g_4) \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & -1 & -2 \end{pmatrix} \end{array} \begin{matrix} 1 \\ a \\ b \\ ab \end{matrix}$$

The kernel of A is generated by the three vectors $(0,-1,-1,2,0,0,1)$, $(0,-1,0,1,0,1,0)$, and $(-1,1,1,-1,1,0,0)$. Thus, a basis of $\mathbb{L}(I)$ is given by the three linear polynomials

$$g_4 + 2g_1 - a - b, \qquad g_3 + g_1 - a, \qquad g_2 - g_1 + a + b - 1.$$

We call Algorithm 1 "FGLM-style linearization" because it can be considered as the first steps of the change-of-order *FGLM-algorithm* [8] for converting a Gröbner basis G w.r.t. one monomial order into another one G' for a different order.

Typically, the FGLM-algorithm is used to convert a degree-compatible Gröbner basis into one for a lexicographic order, because the former are easier to compute while the latter are better suited for solving polynomial systems. Nevertheless, the FGLM-algorithm can be used to transition between any monomial orders as long as the underlying ideal is zero-dimensional. Algorithm 1 can be considered as the first steps of an FGLM-computation for converting a Gröbner basis into a Gröbner basis w.r.t. a degree-compatible order. We note that, while FGLM is only applicable to zero-dimensional ideals, Algorithm 1 works for arbitrary ideals.

The complexity of Algorithm 1 is essentially cubic in the size of the computed normal forms $\mathrm{NF}_G(1), \mathrm{NF}_G(x_1), \ldots, \mathrm{NF}_G(x_n)$, which could be exponential in the worst case, cf. [11, Ex. 5.1]. In our application, we found that Algorithm 1 works well for ideals generated by several dozen polynomials in a few dozen variables. However, for dealing with more involved (sub)circuits, we have to handle ideals spanned by several thousand polynomials in several thousand variables. For such ideals, the normal forms consist of tens of millions of terms and are too large to compute. Therefore, in these cases, we switch to a different method, exploiting the fact that we can cheaply compute many points in the variety $V(I)$ of the considered ideals.

3.2 Guess-and-Prove Linearization

In cases where we do not have access to a Gröbner basis or when computing the normal forms required in Algorithm 1 is prohibitively expensive, we can employ a different strategy: When (sufficiently many) points in the variety $V(I)$ can be computed, we can guess candidates for linear polynomials by interpolation and then verify their correctness *a posteriori*. Our approach is based on the following basic result.

▶ **Lemma 15.** *Let $I \subseteq \mathbb{K}[X]$ be an ideal and let $P \subseteq V(I)$. If $S \subseteq \mathbb{K}[X]$ is the set of all linear polynomials that vanish on P, then $\mathbb{L}(I) \subseteq S$.*

Proof. Follows from the fact that all polynomials in I vanish on all points in $V(I)$. ◀

Lemma 15 provides a necessary condition for determining whether an ideal I contains any nonzero linear polynomials. Given a set of points $P \subseteq V(I)$, we can compute a $\mathbb{K}$-vector space basis for the set S by solving a system of linear equations (see lines 3 – 6 in Algorithm 2). If

Algorithm 2 Guess-And-Prove-style Linearization.

Input : a zero-dimensional radical ideal $I \subseteq \mathbb{K}[X]$ over an algebraically closed field $\mathbb{K}$, methods `sample` and `verify` as specified in Sec. 3.2, and a positive integer $N \in \mathbb{N}_{>0}$

Output : a $\mathbb{K}$-vector space basis L of $\mathbb{L}(I)$

$A \leftarrow$ empty matrix;
while *true* **do**
  $p_1, \ldots, p_N \leftarrow$ use `sample`(I) to compute N points in $V(I)$;
  Prepend the coordinate 1 to each p_i and add these vectors as new rows to A, i.e.,

$$A \leftarrow \begin{pmatrix} & A & \\ 1 & — \, p_1 \, — & \\ \vdots & \vdots & \\ 1 & — \, p_N \, — & \end{pmatrix}$$

  $b_1, \ldots, b_k \leftarrow$ a basis of $\ker(A)$;
  $L \leftarrow \{b_{i,0} + b_{i,1}x_1 + \cdots + b_{i,n}x_n \mid b_i = (b_{i,0}, b_{i,1}, \ldots, b_{i,n}), i = 1, \ldots, k\}$;
  if $\forall f \in L :$ `verify`(f) **then**
    return L;

this basis is empty, it follows that $\mathbb{L}(I) = \{0\}$. More precisely, if the computed basis does not include a linear polynomial involving a given variable $x \in X$, then we can conclude that neither does $\mathbb{L}(I)$.

Obviously, the set S can contain wrong guesses and be (a lot) larger than $\mathbb{L}(I)$. To turn Lemma 15 into an actual algorithm for computing a basis of $\mathbb{L}(I)$, we have to make some additional assumptions:

1. The ideal I is zero-dimensional, i.e., $V(I)$ is finite.
2. The ideal I is radical, which means that $f^r \in I$ implies $f \in I$, for all $f \in \mathbb{K}[X]$, $r \in \mathbb{N}$.
3. The coefficient field $\mathbb{K}$ is algebraically closed, which means that every univariate polynomial in $\mathbb{K}[x]$ has a root in $\mathbb{K}$. For example, $\mathbb{C}$ is algebraically closed, while $\mathbb{Q}$ and $\mathbb{R}$ are not.

In the worst case, we may need to process all points in $V(I)$. The zero-dimensionality of I ensures that we can do this in finite time. The other two conditions exclude pathological cases where polynomials have roots with higher multiplicity or roots in an extension of $\mathbb{K}$.

We furthermore assume that we have access to a function `sample`(I), which returns a point of $V(I)$ and, when called sufficiently often, enumerates all of $V(I)$. Additionally, a function `verify`(f) is assumed, which returns "true" if and only if $f \in I$ and "false" otherwise. We deliberately keep the functions `sample` and `verify` abstract in Algorithm 2 to allow for a high-level presentation. Their instantiation depends on the concrete application. We discuss their instantiation for our application of circuit verification in the following section.

Provided that an ideal I meets all the above assumptions, we can compute a basis of $\mathbb{L}(I)$ through an iterative process, summarized in Algorithm 2. First, we use `sample`(I) to enumerate a subset of points $P \subseteq V(I)$. We then compute a basis L of linear polynomials that vanish on P using linear algebra. Note that, to also recover linear polynomials with a constant term, we have to prepend an additional coordinate 1 to all points in P. If $L \subseteq I$, we have found a basis of $\mathbb{L}(I)$. Otherwise, we sample additional points from $V(I)$ and repeat the process. The assumptions on I ensure that this procedure will eventually terminate.

▶ **Proposition 16.** *Algorithm 2 terminates and is correct.*

Algorithm 3 Verification using linear extractions.

Input : Circuit C in AIG format, Specification polynomial $\mathcal{S}$
Output : Determine whether C fulfills the specification

1 $G_{\text{init}} \leftarrow$ Polynomial-Encoding(C);
2 $\mathcal{S}_{\text{lin}}, G_{\text{ext}} \leftarrow$ Linearize-Spec-wrt-AIG($\mathcal{S}, G_{\text{init}}$);
3 $G \leftarrow$ Preprocessing(G_{ext});
4 **while** $\text{lm}(\mathcal{S}_{\text{lin}}) \in \{\text{lm}(g) | g \in G\}$ **do**
5 $\quad p \leftarrow g \in G$ such that $\text{lm}(g) = \text{lm}(\mathcal{S}_{\text{lin}})$; $G_{\text{sub}} \leftarrow \emptyset$;
6 $\quad$ **while** $\nexists p_{\text{lin}}$ **and** Can-Increase(G_{sub}) **do**
7 $\quad\quad G_{\text{sub}} \leftarrow$ Subcircuit($\text{lm}(g)$, G, G_{sub});
8 $\quad\quad$ **if** $|G_{\text{sub}}|$ *is moderate* **then** $L \leftarrow$ FGLM-style-Linerization(G_{sub});
9 $\quad\quad$ **else** $L \leftarrow$ Guess-and-Prove(G_{sub}, `AdBinSample`, `SAT`, $\min(10 \cdot |G_{\text{sub}}|, 10^4)$);
10 $\quad\quad p_{\text{lin}} \leftarrow p \in L$ such that $\text{lm}(p) = \text{lm}(\mathcal{S}_{\text{lin}})$;
11 $\quad$ **if** $\nexists\, p_{\text{lin}}$ **then return** $\perp$;
12 $\quad \mathcal{S}_{\text{lin}} \leftarrow$ Linear-Reduce($\mathcal{S}_{\text{lin}}, p_{\text{lin}}$);
13 **return** $\mathcal{S}_{\text{lin}} = 0$;

Proof. For termination, note that the assumptions on the method `sample` ensure that the matrix A will eventually contain all of $V(I)$ as its rows. Then, by construction, all elements in L vanish on all of $V(I)$. With this, Hilbert's Nullstellensatz [5, Thm. 4.1.2] together with the assumption that I is radical ensure that $L \subseteq I$. Thus, the test in line 7 will succeed and the algorithm will terminate and return L. For correctness, the returned set L consists of linear polynomials that all lie in I by the test in line 7. Thus $L \subseteq \mathbb{L}(I)$, and Lemma 15 ensures that L is actually a basis of $\mathbb{L}(I)$. ◀

In the worst case, Algorithm 2 has to consider all points in $V(I)$ before finding a correct basis of $\mathbb{L}(I)$. However, in practice, a much smaller subset of points is often sufficient, especially if the sampling strategy is well-designed. In particular, Algorithm 2 can be optimized by including a *repair step*, which prevents wrong guesses from reoccurring in subsequent iterations. If `verify`(f) in line 7 fails for a polynomial $f \in L$, i.e., $f \notin I$, then there is at least one point $p \in V(I)$ such that $f(p) \neq 0$. This point p cannot have been sampled in line 3. If p can be computed, adding it to A in the next iteration ensures that the incorrect guess f cannot reoccur, thereby improving the algorithm's convergence rate.

4 Recovering Linear Polynomials for Circuit Verification

We now discuss how we use the FGLM-style linearization and guess-and-prove-style linearization algorithms of the previous section in the context of multiplier verification.

Multipliers typically consist of three components:

1. *partial product generation (PPG)*, where the individual bitwise products are computed,
2. *partial product accumulation (PPA)*, where these products are combined using compression techniques such as Wallace trees or Dadda trees, and
3. *the final-stage adder (FSA)*, which produces the final multiplication result. The FSA is particularly critical in determining circuit efficiency, and may use advanced addition techniques like carry-lookahead techniques to speed up computation.

The main loop of our approach to circuit verification is outlined in Algorithm 3. The structure of this loop follows the idea from [13] of linearizing only subcircuits instead of attempting to extract linear polynomials from the entire circuit. We discuss optimizations of Algorithm 3 in Section 4.1.

Line 1 – 2: We encode the circuit using the translation presented in Definitions 8 and 9, and linearize the specification polynomial $\mathcal{S} \in \mathbb{K}[X]$ by replacing every non-linear monomial m_i by a new variable y_i and adding the set $\Gamma = \{y_i - m_i \mid y_i \notin X \wedge m_i \in \mathcal{S} \wedge \deg(m_i) > 1\}$ to the set of gate polynomials. The correctness proof for this step can be found in [13, Lem. 3].

Line 3: We apply the same basic preprocessing techniques as in [13]. That is, we eliminate all gates that only occur positively and detect vanishing monomials. Vanishing monomials are pairs of nodes that have the same children, but with different polarities. Thus, the product of the parent nodes is equal to zero. We additionally enhance the preprocessing by propagating vanishing monomials, see Section 4.1.

Furthermore, we identify and mark parts of the circuit that require dedicated reasoning. In the case of multiplier circuits, these correspond to the FSA. Whenever the FSA employs carry-lookahead techniques, we must treat the entire FSA as a single subcircuit to derive the required linear polynomials. This necessity arises because carry-lookahead techniques compute carries in parallel, unlike sequential approaches such as ripple-carry adders. For identifying these subcircuits, we utilize the cut identification method implemented in AMulet2 [15].

Line 4 – 6: The algorithm then iterates over the linearized specification as long as we have a gate polynomial in the encoding that has the same leading monomial as the linearized specification. In each iteration, we generate a subcircuit with root node g for which we perform the linearization. If no matching linearized polynomial p_{lin} is found, we increase the subcircuit size until completion. This is the case when the subcircuit contains all gates that are topologically smaller than g.

Line 7: We identify the subcircuit as follows. If the root node g contains a marking, i.e., if it belongs to a pattern identified in line 3, all nodes with the same marking are collected and returned. For unmarked nodes, the algorithm initially constructs a subcircuit G_{sub} by collecting all nodes up to a given depth d with a fanout size below the specified limit f_{out}. Our initial values are $d = 2$ and $f_{\text{out}} = 4$ as this has empirically shown to be the most efficient. Additionally, we include all nodes whose children are already part of G_{sub}. We also promote all input nodes of G_{sub} with a fanout of one to be included in G_{sub}.

Rather than immediately increasing the depth d in following iterations, we attempt to expand the subcircuit by adding individual input nodes of G_{sub}. The expansion prioritizes the largest input node, based on our predefined variable ordering that still has a fanout size at most f_{out}. However, every 15 iterations, the algorithm restarts with incremented depth and fanout limits. The threshold of 15 was empirically selected to keep the alternations between the depth-first and breath-first selection approaches balanced.

In that sense, our subcircuit selection algorithm is more sophisticated than the one in [13], which did not consider expanding the subcircuit by individual nodes, neither was there a limit on the fanouts nor were special markings considered.

Line 8: If G_{sub} is of moderate size, that is, the number of polynomials is less than 100, and the depth is at most 5 (which in our setting corresponds to all subcircuits except the marked FSA), we use the FGLM-based extraction method.

There is one exception: we also use the FGLM-based extraction method for FSA where, after preprocessing, we detect polynomials with a degree greater than the input bitwidth of the circuit, but consisting only of two terms. We detected that the guess-and-prove approach will mostly produce incorrect guesses for such polynomials due to the high-degree monomials evaluating to zero exponentially often and evaluating to 1 only for one particular input.

Line 9: For larger subcircuits with polynomials of moderate degree, we use the guess-and-prove approach to avoid excessive growth in normal forms, which would cause the computation to stall in the FGLM-style algorithm.

First of all, let us note that the ideals arising from the subcircuits satisfy all assumptions made in Section 3.2. They are zero-dimensional and radical because they contain, by definition, the Boolean value polynomials for all variables. The presence of these polynomials also takes care of the third assumption: Even if a considered ideal I is formally defined over a field $\mathbb{K}$ that is not algebraically closed, we can implicitly treat it as defined over the algebraic closure $\overline{\mathbb{K}}$[1] of $\mathbb{K}$. The Boolean value polynomials ensure that the underlying variety $V(I)$ remains unchanged, which guarantees that our algorithm will return the correct result.

We instantiate the `sample` and `verify` plugins required in Algorithm 2 as follows. For sampling, we apply an adaptive strategy. In the first iteration, we randomly draw values 0 and 1 for all the inputs of the subcircuit and propagate the signals according to the gate structure of the subcircuit. We draw at least 10 times the size of gates in G_{sub} samples, but at most 10 000. This ensures that we have enough samples, while keeping the matrix A in Algorithm 2 at a computationally manageable size. To ensure that every input appears with both polarities, we also include the trivial samples by setting all inputs to 0 and to 1, respectively. If more than one iteration is required in Algorithm 2, we switch from random sampling to the repair strategy discussed at the end of Section 3.2. Using our verification plugin, we can compute witnesses for wrong guesses in the form of points in the variety and only append those witnesses to the matrix in subsequent iterations. We call our strategy `adaptive-binary-sampling` (`AdBinSample`).

As a verification technique, we use SAT solving (`SAT`). Verifying that a guessed polynomial f lies in the ideal induced by G_{sub} corresponds to showing that f equals zero for all signals that can pass through the circuit. To show this, we assume $f \neq 0$ and show unsatisfiability. We first encode each AIG gate that belongs to G_{sub} into conjunctive normal form (CNF). To translate $f \neq 0$, we split it into two pseudo-Boolean (PB) constraints $f < 0$ and $f > 0$, both of which we translate into CNF using PBLIB [23]. PBLIB does not offer a direct encoding of $\neq$. We make two SAT calls using the SAT solver KISSAT [3]: one for the constraint $f < 0$ and another for $f > 0$. If $f = 0$, both calls have to return "unsatisfiable".

We acknowledge that we could also use a PB solver, such as ROUNDINGSAT [7], directly without making the detour of translating everything to CNF. We have tried this, and there was no computational difference. All instances that could be solved using KISSAT could also be solved by ROUNDINGSAT in the same time. However, we decided to go with KISSAT because it was easier to integrate directly into our C++ implementation.

If one instance $f < 0$ or $f > 0$ is satisfiable, we collect the satisfying assignment provided by KISSAT as a witness for the wrong guess and add it as a sample in the next iteration of the guess-and-prove algorithm. This ensures that the incorrect guess f cannot reappear in future iterations, leading to a faster convergence of the guessing procedure.

Moreover, by using Lemma 15, we can also quickly identify situations when the considered subcircuit was chosen too small. We are only interested in finding a linear polynomial in G_{sub} that involves the variable $\text{lm}(g)$. If, at some point in Algorithm 2, the candidate set L does not contain any polynomial involving this variable, then Lemma 15 implies that no such polynomial can exist in G_{sub} (even if the candidate set L still contains incorrect guesses). Thus, in such a situation, we can immediately abort Algorithm 2 and extend G_{sub}.

Line 10– 13: In case our linearization was successful, we extract p_{lin} from L to reduce the specification. If no polynomial p_{lin} was found after choosing G_{sub} as large as possible, we return *false*. We reduce the linearized specification S_{lin} by p_{lin}. After the main loop terminates, that is, when S_{lin} cannot be reduced any further, we return whether the final result is equal to zero.

[1] The algebraic closure $\overline{\mathbb{K}}$ of a field $\mathbb{K}$ is the smallest algebraically closed field containing $\mathbb{K}$. Every field has an algebraic closure. For instance, $\overline{\mathbb{R}} = \mathbb{C}$.

4.1 Optimizations

Caching. To minimize algebraic computations, we cache the computed linear basis L for subcircuits. Whenever we encounter an isomorphic subcircuit, i.e., one that is structurally equivalent up to variable renaming, we reuse the stored linear basis L. To compare subcircuits for this kind of equivalence, we map the variables of both circuits to a set of "standardized" variables. Then we can compare the polynomial encodings of the circuits for syntactic equality. In our application, we have observed cases where more than 99% of the utilized subcircuits are cached.

Extract all linear polynomials. In line 10 of Algorithm 3 we actually do not only extract p_{lin}, but all linear polynomials of L. Since p_{lin} is the root of the subcircuit, all other linear polynomials in L are topologically smaller and may be required to reduce $\mathcal{S}_{\text{lin}}$ at later stages.

Propagating Vanishing Constraints. In [13], only pairs of nodes with equal input nodes are marked as vanishin monomials. We extend this by further propagating the vanishing monomials along the positive parent nodes. For example, consider a detected vanishing monomial $g_1 g_2 = 0$, and assume that there are gates with $g_3 - g_1(1 - g_4)$, $g_5 - g_2 g_6$, where g_1 and g_2 are positive inputs, then we also have $g_2 g_3 = g_1 g_5 = g_3 g_5 = 0$.

5 Experiments

We have implemented Algorithm 3 in a new C++ tool, called TalisMan. We highlight that TalisMan is not restricted to circuit verification, but it can handle any AIG.

We now evaluate TalisMan and compare it to related work [12–14, 17] using a total of 207 integer multiplier benchmarks. All benchmarks represent correct multipliers, meaning the circuits satisfy their specifications. We consider two types of benchmarks: *structured circuits*, where the components of a multiplier are clearly recognizable, and *synthesized circuits*, where gates are merged and rewritten to optimize the circuit, blurring component boundaries and complicating direct verification. We consider the following two sets:

1. *Structured aoki-multipliers* [10]: This set of benchmarks is generated by combining different architectures for PPG, PPA, and FSA[2], yielding 192 structured multiplier architectures. All of these circuits have an input bit-width of 64 and consist of 38 000 to 52 000 nodes.
2. *Synthesized ABC-multipliers* [1]: We generate a multiplier using ABC, consisting of a simple PPG, an array PPA, and a ripple-carry FSA for bitwidths 32, 64, and 128. We optimize the multipliers using four standard synthesis scripts (resyn, resyn2, resyn3, dc2) and a complex script that combines multiple techniques[3]. These 15 optimized benchmarks showcase the robustness of our approach. The node size ranges between 8000 and 130 000.

Our experiments are conducted on a cluster of dual-socket AMD EPYC 7313 @ 3.7GHz machines running Ubuntu 24.04. The time is listed in rounded seconds (wall-clock time). We set the time limit to 300 s and the memory limit to 25 000 MB.

5.1 Results

In the following, we discuss the results of the experimental evaluation, splitting it into a comparison of TalisMan with related work and an ablation study on TalisMan.

[2] PPG: simple (sp), Booth encoding (bp); PPA: Array (ar), Wallace tree (wt), Balanced delay tree (bd), Overturned-stairs tree (os), Dadda tree (dt), (4;2) compressor tree (ct), (7,3) counter tree (cn), Red. binary addition tree (ba); FSA: Ripple-carry (rc), Carry look-ahead (cl), Ripple-block carry look-ahead (rb), Block carry look-ahead (bc), Ladner-Fischer (lf), Kogge-Stone (ks), Brent-Kung (bk), Han-Carlson (hc), Conditional sum (cn), Carry select (cs), Carry-skip fix size (csf), Carry-skip var. size (csv)

[3] `-c "logic; mfs2 -W 20; ps; mfs; st; ps; dc2 -l; ps; resub -l -K 16 -N 3 -w 100; ps; logic; mfs2 -W 20; ps; mfs; st; ps; iresyn -l; ps; resyn; ps; resyn2; ps; resyn3; ps; dc2 -l; ps;"`

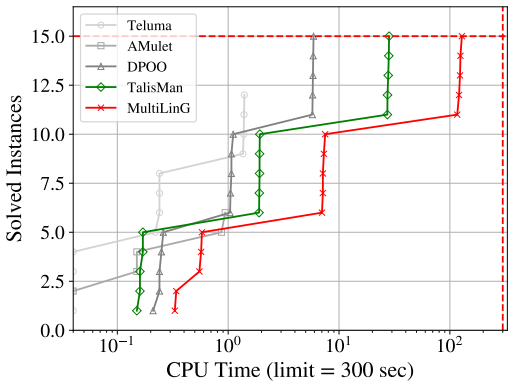


Figure 2 Performance on ABC benchmarks.

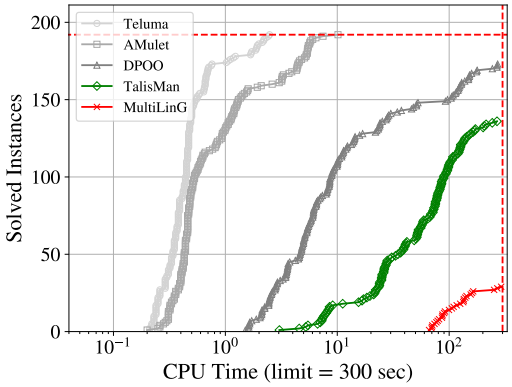


Figure 3 Performance on aoki benchmarks.

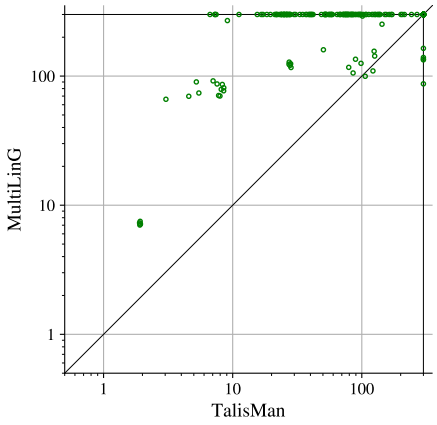


Figure 4 TalisMan vs. MultiLinG.

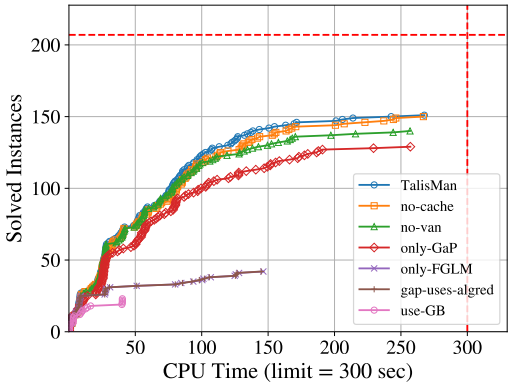


Figure 5 Ablation study of TalisMan.

We compare our tool TalisMan to MultiLinG [13], which also uses a linearization approach, as well as to DynPhaseOrderOpt (DPOO) [17], AMulet2 [14], and TeluMA [12], all of which rely on lexicographic monomial orders. The results for the ABC benchmarks are shown in Figure 2 and for the aoki benchmarks in Figure 3. Tools using linearization approaches (MultiLinG and our tool TalisMan) are depicted in color. All tools using non-linear approaches are depicted in grayscale. We highlight the following key observations:

1. While AMulet2 and TeluMA can solve all of the aoki benchmarks, their approaches are not robust under logic synthesis. AMulet2 solves only 4 ABC benchmarks, whereas TeluMA solves 13. In contrast, DPOO, MultiLinG, and TalisMan solve all of them.
2. DPOO solves 173 aoki benchmarks, while TalisMan solves 136 benchmarks. However, among those 136 benchmarks are 11 that are not solved by DPOO.
3. Among the 44 aoki benchmarks solved by MultiLinG, 5 cannot be solved by TalisMan[4]. We examined these cases but could not identify any shared structural characteristics that might explain MultiLinG's advantage. We believe that the performance difference may be due to variations in the heuristics used for subcircuit selection.

[4] Those instances are: bp-ba-cl, bp-os-cl, sp-cn-cl, bp-cn-rc, and sp-cn-rc. All of them time out in the inner loop of Algorithm 3.

Table 1 Statistics of selected benchmarks. TO: > 300 sec, MO: > 25000 MB, EE: error.

Name	Related Work				TalisMan									
					Total	Subcircuits		FGLM		Guess and Prove				
	[14]	[12]	[17]	[13]	Time	#	%Ch	#	Time	#	Time	Guess	Eval	%Co	m.It.
abc64-cmp	TO	EE	1.1	7.3	1.9	4033	99.7	12	0.0	0	0	0	0	0	0
abc64-rsn2	TO	0.2	1.1	7.2	1.9	4033	99.7	12	0.0	0	0	0	0	0	0
abc128-cmp	TO	EE	5.8	123.5	27.3	16257	99.9	12	0.0	0	0	0	0	0	0
abc128-rsn2	TO	1.4	5.9	124.6	28.1	16257	99.9	12	0.0	0	0	0	0	0	0
bp-ar-ks	0.6	0.5	52.7	TO	205.9	6040	98.7	74	0.6	3	197.6	15087	3855	28.5	5
bp-ba-hc	0.8	0.6	TO	TO	113.2	6147	97.6	144	0.6	1	104.2	4432	2754	48.0	3
bp-bd-rb	0.5	0.5	24.1	TO	75.6	5561	98.2	100	0.6	2	67.9	4452	1364	80.3	3
bp-cn-cl	10.2	2.5	TO	155.8	124.4	2684	82.3	474	32.3	0	0	0	0	0	0
bp-ct-csv	0.9	0.3	10.2	TO	68.3	5204	88.2	614	40.4	1	17.3	864	864	100	1
bp-dt-lf	0.5	0.5	4.8	TO	94.0	5605	98.3	96	0.7	1	85.8	5091	2285	52.1	4
bp-os-rc	0.3	0.2	3.6	86.2	8.3	5697	98.1	108	0.7	0	0	0	0	0	0
bp-wt-csf	1.1	0.5	8.9	TO	24.0	6007	98.0	117	0.7	1	14.8	800	800	100	1
sp-ar-rc	0.2	0.2	1.5	66.1	3.0	6509	99.7	18	0.0	0	0	0	0	0	0
sp-ba-csf	1.2	0.5	6.1	TO	26.3	8012	98.9	87	0.0	1	18.0	883	883	100	1
sp-bd-lf	0.5	0.4	2.8	TO	81.5	6403	99.4	40	0.0	1	76.3	5733	2077	52.4	5
sp-ct-cl	4.5	1.7	TO	291.6	101.1	7782	95.7	335	25.4	0	0	0	0	0	0
sp-ct-hc	0.5	0.3	47.9	TO	98.2	7756	97.7	174	0.1	1	89.0	3675	2154	52.5	3
sp-dt-cs	0.4	0.3	2.5	TO	127.2	6958	99.2	57	0.0	2	120.5	5906	1735	83.7	3
sp-os-bc	0.4	0.4	9.5	MO	57.9	6420	99.2	50	0.0	2	52.7	6794	1137	84.9	4
sp-wt-ks	0.9	0.5	114.0	TO	137.0	7639	99.3	49	0.0	1	129.0	4701	2861	48.1	3

While we are still unable to fully outperform the *non-linear* lexicographic reasoning approaches, TalisMan solves around three times more benchmarks and is an order of magnitude faster than the *linearization* approach of MultiLinG. A direct comparison of the two linearization approaches is shown in Figure 4.

We present statistics on selected benchmarks in Table 1. The first block shows the timings of related work, while the second block provides additional insights into our results. We list the *total time* in rounded seconds. The *subcircuits* block lists the number of generated subcircuits ("#") and the percentage of circuits found in the cache ("%Ch"). In *FGLM*, we list the number of calls ("#") and the total time ("Time") spent in the FGLM algorithm. In the *Guess and Prove* block, we also list the number of calls ("#") along with the total time ("Time"). We further state the number of guessed polynomials ("Guess") and the actual number evaluated ("Eval"), since we do not re-evaluate polynomials already found in previous iterations. We show the percentage of correctly guessed polynomials ("%Co"), and we also list the maximum number of iterations needed ("m.It."). A complete table, showing statistics for all 207 benchmarks, is included in Appendix A.

Table 1 reveals some interesting facts: we can cache more than 80% of the circuits, even for synthesized ones. When there are fewer than 100 FGLM calls, computation time remains very low. In some cases, the guess-and-prove method immediately returns the correct polynomials. However, at most 5 iterations are needed to repair the incorrect guesses.

The time difference between "Total Time" and the "FGLM-time" and "Guess-and-Prove-Time" is mostly spent on the reduction of the specification. Only a negligible amount of computation time is spent on parsing and preprocessing. Around 30–50% of the time listed for "Guess and Prove" is spent in the SAT solver.

We also run an ablation study on TalisMan. The results are summarized in Figure 5. We compare the default setting of TalisMan to the following configurations: We only use FGLM-style linearization (*only-FGLM*) or only guess-and-prove-style linearization (*only-*

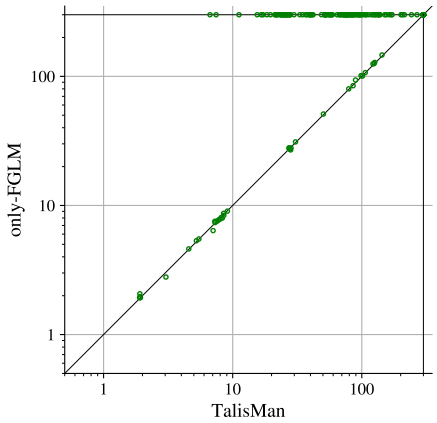


Figure 6 TalisMan vs. only-FGLM.

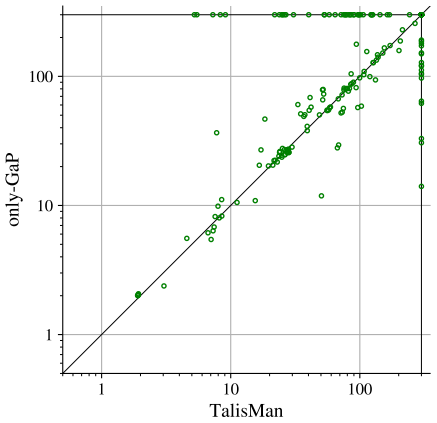


Figure 7 TalisMan vs. only-GaP.

GaP) to compute linear polynomials. We turn off caching and always repeat all algebraic computations (*no-cache*) or turn off identification and propagation of vanishing monomials (*no-van*). We use repeated polynomial division (= ideal membership test) for verifying the guessed polynomials, instead of using Kissat (*gap-uses-algred*). And finally, since we also use a different subcircuit heuristic, employ propagation of vanishing monomials, and use caching, a direct comparison with MultiLinG does not fully reflect the strengths of our hybrid approach in comparison to using a degree-compatible Gröbner basis. Hence, we have also added the option to extract linear polynomials by computing a degree-compatible Gröbner basis using msolve [2] in TalisMan (*use-GB*).

It can be seen in Figure 5 that the default settings of TalisMan and *no-cache* solve the most benchmarks. Surprisingly, turning off caching does not have a huge negative impact on the computation time. This indicates that caching a subcircuit is as expensive as simply conducting the algebraic operations in our case. Turning off vanishing monomials leads to a loss of 11 benchmarks. When forcing TalisMan to use only one of our two proposed approaches, we always lose benchmarks. Using only FGLM-style linearization leads to only 42 solved instances, while using only guess-and-prove-style linearization leads to 129 solved benchmarks. We have included scatter plots in Figure 6 and Figure 7. While it can be seen in Figure 6 that using only FGLM always worsens the situation, Figure 7 shows that there are some benchmarks that benefit from using only the guess-and-prove approach. Using only the guess-and-prove approach solved 18 benchmarks that are not solved in the default settings, 17 of which use a (7,3)-counter tree as PPA. Verifying guessed polynomials using algebra, or using the Gröbner basis approach instead, lead to a tremendous loss of benchmarks.

6 Conclusion

We have presented a hybrid approach for extracting linear polynomials from a set of polynomials. Our approach combines an FGLM-style linearization method with a guess-and-prove-style approach. We first present these algorithms at a general level before instantiating them for circuit verification. Our experimental evaluation shows that we outperform existing linearization techniques based on Gröbner bases. In the future, we aim to further improve our tool with more sophisticated algorithms for subcircuit detection. We also envision that our methods have applications beyond circuit verification, such as equivalence checking.

References

1 Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/, 2019. Bitbucket Version 1.01.

2 Jérémy Berthomieu, Christian Eder, and Mohab Safey El Din. msolve: A library for solving polynomial systems. In *Intl. Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 51–58. ACM, 2021. doi:10.1145/3452143.3465545.

3 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024. In *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.

4 Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* PhD thesis, University of Innsbruck, 1965.

5 David A. Cox, John Little, and Donal O'Shea. *Ideals, varieties, and algorithms - an introduction to computational algebraic geometry and commutative algebra (2. ed.)*. Undergraduate texts in mathematics. Springer, 1997.

6 David A. Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry.* Springer, 2005.

7 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 *Conf. on Artificial Intelligence AAAI*, pages 3750–3758. AAAI Press, 2021. doi:10.1609/aaai.v35i5.16492.

8 Jean-Charles Faugère, Patrizia M. Gianni, Daniel Lazard, and Teo Mora. Efficient computation of zero-dimensional gröbner bases by change of ordering. *J. Symb. Comput.*, 16(4):329–344, 1993. doi:10.1006/jsco.1993.1051.

9 Clemens Hofstadler and Daniela Kaufmann. TalisMan. Software, version 1.0., swhId: swh:1:dir:8af022a61661d2b7fb281abbdc2f18d51f221c20 (visited on 2025-07-16). URL: https://github.com/d-kfmnn/talisman/tree/be8187d, doi:10.4230/artifacts.23376.

10 Naofumi Homma, Yuki Watanabe, Takafumi Aoki, and Tatsuo Higuchi. Formal design of arithmetic circuits based on arithmetic description language. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 89-A(12):3500–3509, 2006. doi:10.1093/ietfec/e89-a.12.3500.

11 Daniela Kaufmann. *Formal Verification of Multiplier Circuits using Computer Algebra.* PhD thesis, Computer Science, Johannes Kepler University Linz, 2020.

12 Daniela Kaufmann, Paul Beame, Armin Biere, and Jakob Nordström. Adding dual variables to algebraic reasoning for gate-level multiplier verification. In *2Design, Automation & Test in Europe Conf. & Exhibition, DATE*, pages 1431–1436. IEEE, 2022. doi:10.23919/DATE54114.2022.9774587.

13 Daniela Kaufmann and Jérémy Berthomieu. Extracting linear relations from gröbner bases for formal verification of and-inverter graphs. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS*, volume 15696 of *LNCS*, pages 355–374. Springer, Springer, 2025. doi:10.1007/978-3-031-90643-5_19.

14 Daniela Kaufmann and Armin Biere. Amulet 2.0 for verifying multiplier circuits. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS*, volume 12652 of *LNCS*, pages 357–364. Springer, 2021. doi:10.1007/978-3-030-72013-1_19.

15 Daniela Kaufmann, Armin Biere, and Manuel Kauers. Verifying large multipliers by combining SAT and computer algebra. In *Formal Methods in Computer Aided Design, FMCAD*, pages 28–36. IEEE, 2019. doi:10.23919/FMCAD.2019.8894250.

16 Daniela Kaufmann, Armin Biere, and Manuel Kauers. Incremental column-wise verification of arithmetic circuits using computer algebra. *Formal Methods Syst. Des.*, 56(1):22–54, 2020. doi:10.1007/s10703-018-00329-2.

17 Alexander Konrad and Christoph Scholl. Symbolic computer algebra for multipliers revisited - it's all about orders and phases. In *Formal Methods in Computer-Aided Design FMCAD*, pages 261–271. IEEE, 2024. doi:10.34727/2024/isbn.978-3-85448-065-5_32.

18 Alexander Konrad, Christoph Scholl, Alireza Mahzoon, Daniel Große, and Rolf Drechsler. Divider verification using symbolic computer algebra and delayed don't care optimization. In *Formal Methods in Computer-Aided Design FMCAD*, pages 1–10. IEEE, 2022. doi:10.34727/2022/isbn.978-3-85448-053-2_17.

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

20 Jinpeng Lv, Priyank Kalla, and Florian Enescu. Efficient gröbner basis reductions for formal verification of galois field arithmetic circuits. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 32(9):1409–1420, 2013. doi:10.1109/TCAD.2013.2259540.

21 Alireza Mahzoon, Daniel Große, and Rolf Drechsler. Polycleaner: clean your polynomials before backward rewriting to verify million-gate multipliers. In *Intl. Conf. on Computer-Aided Design ICCAD*, page 129. ACM, 2018. doi:10.1145/3240765.3240837.

22 Alireza Mahzoon, Daniel Große, Christoph Scholl, and Rolf Drechsler. Towards formal verification of optimized and industrial multipliers. In *Design, Automation & Test in Europe Conf. & Exhibition, DATE*, pages 544–549. IEEE, 2020. doi:10.23919/DATE48585.2020.9116485.

23 Tobias Philipp and Peter Steinke. Pblib - A library for encoding pseudo-boolean constraints into CNF. In *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th Intl. Conf., Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *LNCS*, pages 9–16. Springer, 2015. doi:10.1007/978-3-319-24318-4_2.

24 William A. Stein. *Modular Forms: A Computational Approach*, volume 79. American Mathematical Soc., 2007.

25 Mertcan Temel. Vescmul: Verified implementation of s-c-rewriting for multiplier verification. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS*, volume 14570 of *LNCS*, pages 340–349. Springer, 2024. doi:10.1007/978-3-031-57246-3_19.

A Statistics on Benchmarks

We present statistics for the full benchmark set used in our experimental evaluation.

Table 2 Statistics of TalisMan and related work. TO: > 300 sec, MO: > 25000 MB, EE: error.

Name	Related Work				TalisMan										
					Total	Subcircuits		FGLM		Guess and Prove					
	[14]	[12]	[17]	[13]	Time	#	%Ch	#	Time	#	Time	Guess	Eval	%Co	m.It.
abc32-cmp	TO	EE	0.2	0.6	0.2	993	98.8	12	0.0	0	0	0	0	0	0
abc32-dc2	0.0	0.0	0.2	0.3	0.2	992	99.1	9	0.0	0	0	0	0	0	0
abc32-rsn	TO	0.0	0.2	0.6	0.2	993	98.8	12	0.0	0	0	0	0	0	0
abc32-rsn2	TO	0.0	0.3	0.6	0.2	993	98.8	12	0.0	0	0	0	0	0	0
abc32-rsn3	0.0	0.0	0.2	0.3	0.2	992	99.1	9	0.0	0	0	0	0	0	0
abc64-cmp	TO	EE	1.1	7.3	1.9	4033	99.7	12	0.0	0	0	0	0	0	0
abc64-dc2	0.1	0.2	1.1	7.0	1.9	4032	99.8	9	0.0	0	0	0	0	0	0
abc64-rsn	TO	0.2	1.0	7.5	1.9	4033	99.7	12	0.0	0	0	0	0	0	0
abc64-rsn2	TO	0.2	1.1	7.2	1.9	4033	99.7	12	0.0	0	0	0	0	0	0
abc64-rsn3	0.1	0.2	1.1	7.2	1.9	4032	99.8	9	0.0	0	0	0	0	0	0
abc128-cmp	TO	EE	5.8	123.5	27.3	16257	99.9	12	0.0	0	0	0	0	0	0
abc128-dc2	0.9	1.4	5.8	121.0	27.8	16256	99.9	9	0.0	0	0	0	0	0	0
abc128-rsn	TO	1.4	5.8	116.5	28.3	16257	99.9	12	0.0	0	0	0	0	0	0
abc128-rsn2	TO	1.4	5.9	124.6	28.1	16257	99.9	12	0.0	0	0	0	0	0	0
abc128-rsn3	0.9	1.4	5.8	128.3	27.4	16256	99.9	9	0.0	0	0	0	0	0	0
bp-ar-bc	0.4	0.5	11.9	TO	33.1	6039	98.7	75	0.6	2	25.2	2253	848	87.3	2
bp-ar-bk	0.4	0.5	4.2	TO	41.1	6042	98.7	74	0.6	3	33.2	4327	852	86.2	3
bp-ar-cl	2.6	1.1	243.2	TO	30.6	2425	92.4	184	8.6	0	0	0	0	0	0
bp-ar-cn	0.5	0.4	6.1	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-ar-cs	0.4	0.3	5.1	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-ar-csf	0.4	0.5	9	TO	17.2	6045	98.7	78	0.6	1	9.3	628	628	100	1
bp-ar-csv	0.5	0.5	6.2	TO	18.4	6038	98.7	75	0.6	1	10.5	673	673	100	1
bp-ar-hc	0.5	0.5	11.0	TO	51.9	6040	98.8	74	0.6	1	44.1	2807	1557	55.9	3
bp-ar-ks	0.6	0.5	52.7	TO	205.9	6040	98.7	74	0.6	3	197.6	15087	3855	28.5	5
bp-ar-lf	0.4	0.5	4.6	TO	51.4	6040	98.8	74	0.6	1	43.4	3589	1438	59.5	4
bp-ar-rb	0.4	0.5	10.9	TO	52.5	6040	98.7	75	0.6	3	44.6	3813	1011	84.3	2
bp-ar-rc	0.3	0.2	3.5	70.7	7.8	6132	98.7	80	0.6	0	0	0	0	0	0
bp-ba-bc	0.5	0.5	12.2	TO	58.3	6148	97.6	144	0.7	2	49.8	4042	1188	83.9	3
bp-ba-bk	0.5	0.5	14.3	TO	83.1	6148	97.6	144	0.7	2	74.6	5890	1987	56.4	3
bp-ba-cl	5.7	2.4	TO	135.3	TO	-	-	-	-	-	-	-	-	-	-
bp-ba-cn	0.5	0.4	46.1	MO	TO	-	-	-	-	-	-	-	-	-	-
bp-ba-cs	0.4	0.3	5.6	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-ba-csf	1.3	0.5	8.2	TO	27.8	6147	97.6	144	0.7	1	18.2	894	894	100	1
bp-ba-csv	1.2	0.5	8.5	TO	28.0	6147	97.6	144	0.7	1	18.6	898	898	99.8	1
bp-ba-hc	0.8	0.6	TO	TO	113.2	6147	97.6	144	0.6	1	104.2	4432	2754	48.0	3
bp-ba-ks	1.2	0.6	151.6	TO	213.8	6147	97.6	144	0.7	1	204.4	7493	3969	41.5	4
bp-ba-lf	0.6	0.5	5.2	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-ba-rb	0.5	0.5	13.7	MO	85.6	6148	97.6	144	0.7	2	77.1	4728	1480	78.7	3
bp-ba-rc	0.3	0.3	3.6	81.2	8.5	6275	97.7	147	0.7	0	0	0	0	0	0
bp-bd-bc	0.4	0.5	10.0	TO	41.9	5560	98.2	100	0.6	1	34.0	3831	1057	89.8	4
bp-bd-bk	0.5	0.5	5.0	TO	51.6	5563	98.2	99	0.6	2	43.6	3801	1110	85.0	3
bp-bd-cl	4.8	2.0	TO	125.5	98.5	2305	87.8	280	27.8	0	0	0	0	0	0
bp-bd-cn	0.5	0.4	6.0	MO	TO	-	-	-	-	-	-	-	-	-	-
bp-bd-cs	0.4	0.3	5.7	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-bd-csf	1.3	0.5	8.5	TO	23.9	5567	98.2	102	0.6	1	15.0	814	814	100	1
bp-bd-csv	0.9	0.5	8.6	TO	26.0	5560	98.2	100	0.7	1	17.0	858	858	99.9	1
bp-bd-hc	0.6	0.5	28.9	TO	85.0	5562	98.2	99	0.6	1	76.9	3666	2060	54.9	3
bp-bd-ks	0.8	0.6	128.6	TO	150.6	5560	98.2	99	0.6	1	142.2	4909	2841	50.6	3

Name	Related Work				TalisMan										
					Total	Subcircuits		FGLM		Guess and Prove					
	[14]	[12]	[17]	[13]	Time	#	%Ch	#	Time	#	Time	Guess	Eval	%Co	m.It.
bp-bd-lf	0.5	0.5	4.9	EE	138.5	5564	98.2	99	0.6	3	130.6	12083	2725	41.1	4
bp-bd-rb	0.5	0.5	24.1	TO	75.6	5561	98.2	100	0.6	2	67.9	4452	1364	80.3	3
bp-bd-rc	0.3	0.2	3.6	70.1	7.9	5681	98.2	101	0.7	0	0	0	0	0	0
bp-cn-bc	5.4	0.7	267.8	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-cn-bk	5.5	0.7	148.8	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-cn-cl	10.2	2.5	TO	155.8	124.4	2684	82.3	474	32.3	0	0	0	0	0	0
bp-cn-cn	1.3	1.1	51.1	MO	TO	-	-	-	-	-	-	-	-	-	-
bp-cn-cs	1.3	1.0	123.0	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-cn-csf	5.8	0.6	TO	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-cn-csv	6.5	0.7	268.0	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-cn-hc	5.7	0.7	TO	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-cn-ks	6.0	0.8	TO	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-cn-lf	5.5	0.7	152.3	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-cn-rb	5.4	0.7	271.3	MO	TO	-	-	-	-	-	-	-	-	-	-
bp-cn-rc	5.3	0.7	TO	87.1	TO	-	-	-	-	-	-	-	-	-	-
bp-ct-bc	0.4	0.3	37.4	TO	102.6	5205	88.2	614	40.4	2	52.3	6732	1141	83.8	4
bp-ct-bk	0.4	0.3	4.8	TO	96.2	5207	88.2	613	40.7	2	45.6	3837	1152	82.5	3
bp-ct-cl	4.4	1.9	TO	252.2	143.1	4137	81.7	757	65.8	0	0	0	0	0	0
bp-ct-cn	0.4	0.3	6.1	MO	TO	-	-	-	-	-	-	-	-	-	-
bp-ct-cs	0.4	0.3	5.4	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-ct-csf	1.4	0.3	8.6	TO	66.9	5211	88.2	616	40.3	1	15.5	821	821	100	1
bp-ct-csv	0.9	0.3	10.2	TO	68.3	5204	88.2	614	40.4	1	17.3	864	864	100	1
bp-ct-hc	0.6	0.3	21.9	TO	132.0	5206	88.2	613	40.6	1	81.5	3703	2091	54.7	3
bp-ct-ks	1.0	0.4	129.4	TO	200.9	5204	88.2	613	40.2	1	150.2	4978	2982	48.7	3
bp-ct-lf	0.5	0.3	5	TO	126.7	5206	88.2	613	40.3	1	76.4	4831	2111	53.6	4
bp-ct-rb	0.5	0.3	16.9	TO	119.4	5205	88.2	614	40.6	2	68.4	4479	1332	82.8	3
bp-ct-rc	0.3	0.2	3.5	159.9	50.3	5325	88.4	616	40.4	0	0	0	0	0	0
bp-dt-bc	0.5	0.5	14.4	TO	74.6	5606	98.2	97	0.7	3	66.5	6975	1239	80.0	3
bp-dt-bk	0.4	0.5	4.7	TO	72.8	5607	98.2	96	0.7	3	64.8	7317	1253	78.9	4
bp-dt-cl	5.1	2.2	TO	142.9	125.7	2229	90.2	218	31.6	0	0	0	0	0	0
bp-dt-cn	0.5	0.4	6.5	MO	TO	-	-	-	-	-	-	-	-	-	-
bp-dt-cs	0.4	0.3	5.4	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-dt-csf	0.8	0.5	8.8	TO	24.8	5611	98.2	101	0.7	1	16.4	847	847	100	1
bp-dt-csv	1.4	0.5	9.2	TO	27.4	5604	98.2	97	0.7	1	18.4	899	899	99.8	1
bp-dt-hc	0.6	0.5	92.7	TO	107.4	5605	98.3	96	0.7	1	99.2	5068	2200	54.1	4
bp-dt-ks	1.0	0.6	175.2	TO	267.6	5605	98.2	96	0.7	2	259.0	10564	3872	39.1	3
bp-dt-lf	0.5	0.5	4.8	TO	94.0	5605	98.3	96	0.7	1	85.8	5091	2285	52.1	4
bp-dt-rb	0.5	0.5	14.2	TO	59.2	5605	98.2	97	0.7	2	51.2	5002	1178	84.1	4
bp-dt-rc	0.3	0.2	3.5	78.9	8.1	5730	98.2	100	0.7	0	0	0	0	0	0
bp-os-bc	0.4	0.4	10.1	MO	40.2	5575	98.1	107	0.7	1	32.1	2897	1052	90.9	3
bp-os-bk	0.4	0.4	4.6	TO	53.3	5578	98.1	105	0.7	2	45.2	3838	1154	82.4	3
bp-os-cl	5.4	1.9	TO	133.6	TO	-	-	-	-	-	-	-	-	-	-
bp-os-cn	0.5	0.4	6.1	MO	TO	-	-	-	-	-	-	-	-	-	-
bp-os-cs	0.5	0.4	5.5	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-os-csf	1.4	0.4	8.6	TO	24.9	5582	98.0	110	0.7	1	15.6	821	821	100	1
bp-os-csv	1.0	0.4	8.3	TO	26.0	5575	98.1	107	0.7	1	17.2	867	867	99.7	1
bp-os-hc	0.6	0.4	23.5	TO	96.5	5577	98.1	105	0.7	1	88.2	3725	2175	52.5	3
bp-os-ks	1.0	0.5	115.8	TO	169.5	5575	98.1	105	0.7	1	161.0	4978	2964	49.0	3
bp-os-lf	0.5	0.4	5	TO	83.8	5577	98.1	105	0.7	1	75.8	4809	2075	54.5	4
bp-os-rb	0.5	0.4	23.3	TO	77.0	5576	98.0	107	0.7	2	69.0	4471	1325	83.2	3
bp-os-rc	0.3	0.2	3.6	86.2	8.3	5697	98.1	108	0.7	0	0	0	0	0	0
bp-wt-bc	0.4	0.5	11.3	TO	39.1	6000	98.1	115	0.7	1	30.9	2823	1035	90.0	3
bp-wt-bk	0.5	0.5	4.9	TO	40.5	6002	98.1	113	0.7	1	32.1	2796	1038	89.2	3
bp-wt-cl	3.9	1.9	TO	135.1	89.3	2276	85.2	336	22.6	0	0	0	0	0	0

Name	Related Work				TalisMan									
					Total	Subcircuits		FGLM		Guess and Prove				
	[14]	[12]	[17]	[13]	Time	#	%Ch	#	Time	#	Time	Guess	Eval	%Co	m.It.
bp-wt-cn	0.5	0.4	5.9	MO	TO	-	-	-	-	-	-	-	-	-	-
bp-wt-cs	0.4	0.3	5.3	TO	TO	-	-	-	-	-	-	-	-	-	-
bp-wt-csf	1.1	0.5	8.9	TO	24.0	6007	98.0	117	0.7	1	14.8	800	800	100	1
bp-wt-csv	0.8	0.5	8.6	TO	25.1	6000	98.1	115	0.7	1	16.3	847	847	99.5	1
bp-wt-hc	0.6	0.5	121.6	TO	86.7	6002	98.1	113	0.7	1	77.9	3602	2087	53.2	3
bp-wt-ks	0.9	0.6	119.9	TO	154.8	6000	98.1	113	0.7	1	145.8	4798	2779	50.8	3
bp-wt-lf	0.5	0.5	4.7	TO	78.5	6002	98.1	113	0.7	1	70.0	4655	1950	56.4	4
bp-wt-rb	0.5	0.5	22.5	TO	75.3	6001	98.0	115	0.7	2	66.7	4379	1337	80.5	3
bp-wt-rc	0.3	0.2	3.5	76.9	8.5	6118	98.1	116	0.7	0	0	0	0	0	0
sp-ar-bc	0.3	0.2	2.6	TO	15.5	6448	99.8	14	0.0	2	12.6	3451	600	83.2	4
sp-ar-bk	0.3	0.2	1.9	TO	11.2	6448	99.8	15	0.0	1	8.3	1483	540	90.9	3
sp-ar-cl	0.8	0.3	27.8	74.0	5.4	3999	98.1	75	1.2	0	0	0	0	0	0
sp-ar-cn	0.4	0.3	2.6	TO	55.9	6459	99.4	37	0.2	2	52.5	4077	1197	83.1	4
sp-ar-cs	0.4	0.3	2.1	TO	29.8	6460	99.4	37	0.2	3	26.8	1441	732	98.8	1
sp-ar-csf	0.3	0.2	2.2	TO	6.7	6451	99.7	16	0.0	1	3.8	425	425	100	1
sp-ar-csv	0.3	0.2	2.5	EE	7.4	6447	99.8	14	0.0	1	4.5	455	455	100	1
sp-ar-hc	0.3	0.2	4.8	TO	22.9	6448	99.8	15	0.0	1	19.0	1781	967	58.2	3
sp-ar-ks	0.3	0.2	15.8	TO	36.8	6447	99.8	14	0.0	1	32.5	2303	1282	54.5	3
sp-ar-lf	0.3	0.2	2.2	TO	21.2	6448	99.8	15	0.0	1	17.6	2344	879	64.0	4
sp-ar-rb	0.3	0.2	2.4	TO	16.6	6447	99.8	14	0.0	1	13.5	1737	639	89.7	3
sp-ar-rc	0.2	0.2	1.5	66.1	3.0	6509	99.7	18	0.0	0	0	0	0	0	0
sp-ba-bc	0.5	0.5	11.4	TO	55.7	8013	98.9	87	0.0	2	48.5	3998	1172	84.1	3
sp-ba-bk	0.6	0.5	4.5	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-ba-cl	5.2	2.2	TO	EE	TO	-	-	-	-	-	-	-	-	-	-
sp-ba-cn	0.4	0.4	3.3	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-ba-cs	0.4	0.3	2.6	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-ba-csf	1.2	0.5	6.1	TO	26.3	8012	98.9	87	0.0	1	18.0	883	883	100	1
sp-ba-csv	1.1	0.5	8.1	TO	26.3	8012	98.9	88	0.0	1	18.1	887	887	99.8	1
sp-ba-hc	0.8	0.5	158.6	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-ba-ks	1.3	0.6	137.9	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-ba-lf	0.7	0.5	3.1	TO	7.3	8249	98.3	142	0.1	0	0	0	0	0	0
sp-ba-rb	0.5	0.5	10.1	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-ba-rc	0.4	0.2	1.7	EE	7.3	8140	98.9	93	0.0	0	0	0	0	0	0
sp-bd-bc	0.5	0.4	7.0	TO	34.8	6402	99.4	40	0.0	1	30.4	2804	1010	91.8	3
sp-bd-bk	0.4	0.4	2.0	TO	48.7	6404	99.3	40	0.0	2	44.2	5898	1093	84.1	4
sp-bd-cl	4.4	1.7	TO	105.7	85.5	4112	95.6	182	22.5	0	0	0	0	0	0
sp-bd-cn	0.5	0.4	3.5	MO	TO	-	-	-	-	-	-	-	-	-	-
sp-bd-cs	0.4	0.3	2.5	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-bd-csf	1.1	0.4	5.0	TO	19.6	6408	99.3	43	0.0	1	14.4	792	792	100	1
sp-bd-csv	1.1	0.4	6.5	TO	21.6	6402	99.4	40	0.0	1	16.2	838	838	99.9	1
sp-bd-hc	0.6	0.4	25.3	TO	79.9	6403	99.4	40	0.0	1	74.3	3596	2027	54.4	3
sp-bd-ks	1.2	0.5	96.7	TO	134.4	6402	99.4	39	0.0	1	128.9	4782	2724	51.4	3
sp-bd-lf	0.5	0.4	2.8	TO	81.5	6403	99.4	40	0.0	1	76.3	5733	2077	52.4	5
sp-bd-rb	0.5	0.4	8.0	TO	73.0	6403	99.3	40	0.0	2	68.5	5435	1349	79.3	4
sp-bd-rc	0.3	0.2	1.6	69.8	4.6	6520	99.4	42	0.0	0	0	0	0	0	0
sp-cn-bc	3.2	0.7	51.0	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-cn-bk	3.2	0.7	28.2	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-cn-cl	7.5	2.3	TO	164.0	TO	-	-	-	-	-	-	-	-	-	-
sp-cn-cn	1.6	1.2	11.2	MO	TO	-	-	-	-	-	-	-	-	-	-
sp-cn-cs	1.5	1.2	10.4	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-cn-csf	4.6	0.7	13.4	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-cn-csv	4.3	0.7	14	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-cn-hc	3.3	0.7	104.2	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-cn-ks	3.7	0.8	124.9	TO	TO	-	-	-	-	-	-	-	-	-	-

Name	Related Work				TalisMan									
					Total	Subcircuits		FGLM		Guess and Prove				
	[14]	[12]	[17]	[13]	Time	#	%Ch	#	Time	#	Time	Guess	Eval	%Co	m.It.
sp-cn-lf	3.2	0.7	24.1	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-cn-rb	3.2	0.7	23.9	MO	TO	-	-	-	-	-	-	-	-	-	-
sp-cn-rc	3.0	0.7	177.8	139.3	TO	-	-	-	-	-	-	-	-	-	-
sp-ct-bc	0.3	0.3	10.0	TO	71.5	7757	97.7	174	0.1	3	62.6	8277	1175	80.9	4
sp-ct-bk	0.3	0.3	2.1	TO	53.0	7757	97.7	174	0.1	2	44.1	3800	1133	83.2	3
sp-ct-cl	4.5	1.7	TO	291.6	101.1	7782	95.7	335	25.4	0	0	0	0	0	0
sp-ct-cn	0.3	0.3	2.8	MO	TO	-	-	-	-	-	-	-	-	-	-
sp-ct-cs	0.3	0.2	2.5	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-ct-csf	1.2	0.2	6.2	TO	25.2	7761	97.7	176	0.1	1	15.0	814	814	100	1
sp-ct-csv	1.2	0.3	7.4	TO	26.7	7755	97.7	174	0.1	1	16.7	860	860	99.8	1
sp-ct-hc	0.5	0.3	47.9	TO	98.2	7756	97.7	174	0.1	1	89.0	3675	2154	52.5	3
sp-ct-ks	0.7	0.3	108.7	TO	163.2	7755	97.8	173	0.1	1	153.6	4928	2915	49.4	3
sp-ct-lf	0.4	0.3	2.9	TO	90.3	7756	97.7	174	0.1	1	81.2	5942	2211	50.7	5
sp-ct-rb	0.4	0.3	9.3	TO	77.7	7756	97.7	174	0.1	2	68.3	4464	1362	80.6	3
sp-ct-rc	0.2	0.2	1.6	269.1	9.1	7875	97.7	180	0.1	0	0	0	0	0	0
sp-dt-bc	0.5	0.5	5.4	TO	71.5	6959	99.1	57	0.0	3	64.7	6717	1296	76.8	3
sp-dt-bk	0.5	0.4	2.1	TO	68.1	6960	99.1	58	0.0	3	61.3	7084	1258	78.1	4
sp-dt-cl	5.5	2.1	TO	109.6	121.9	3971	95.6	175	30.3	0	0	0	0	0	0
sp-dt-cn	0.4	0.4	3.8	MO	TO	-	-	-	-	-	-	-	-	-	-
sp-dt-cs	0.4	0.3	2.5	TO	127.2	6958	99.2	57	0.0	2	120.5	5906	1735	83.7	3
sp-dt-csf	1.9	0.4	6.7	TO	24.9	6963	99.1	61	0.0	1	16.4	850	850	100	1
sp-dt-csv	2.1	0.4	6.5	EE	27.2	6957	99.2	57	0.0	1	18.6	898	898	99.7	1
sp-dt-hc	0.6	0.5	27.3	TO	99.7	6958	99.2	58	0.0	1	92.7	3855	2213	53.5	3
sp-dt-ks	1.1	0.5	149.1	TO	171.4	6957	99.2	57	0.0	1	164.1	5187	3039	49.5	3
sp-dt-lf	0.5	0.4	2.8	TO	89.2	6958	99.2	58	0.0	1	82.2	5037	2246	52.5	4
sp-dt-rb	0.5	0.4	12.3	TO	108.0	6959	99.1	57	0.0	3	101.1	9581	1531	75.2	4
sp-dt-rc	0.4	0.2	1.6	91.8	7.0	7082	99.2	58	0.0	0	0	0	0	0	0
sp-os-bc	0.4	0.4	9.5	MO	57.9	6420	99.2	50	0.0	2	52.7	6794	1137	84.9	4
sp-os-bk	0.4	0.3	2.1	TO	64.4	6422	99.2	50	0.0	3	59.2	6945	1243	77.1	4
sp-os-cl	4.5	1.9	TO	99.6	106.2	4102	95.0	206	27.3	0	0	0	0	0	0
sp-os-cn	0.5	0.4	3.6	MO	TO	-	-	-	-	-	-	-	-	-	-
sp-os-cs	0.4	0.3	2.5	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-os-csf	1.6	0.3	5.8	TO	21.9	6425	99.2	53	0.0	1	15.4	828	828	100	1
sp-os-csv	1.6	0.3	6.4	TO	23.8	6419	99.2	50	0.0	1	17.3	874	874	99.9	1
sp-os-hc	0.6	0.4	34.4	TO	122.7	6421	99.2	50	0.0	2	116.3	7582	2587	44.5	3
sp-os-ks	0.9	0.4	105.7	TO	242.6	6420	99.2	49	0.0	2	236.1	11547	3766	38.9	4
sp-os-lf	0.5	0.4	2.8	TO	82.6	6420	99.2	50	0.0	1	77.2	4879	2156	53.0	4
sp-os-rb	0.5	0.3	8.9	TO	75.0	6420	99.2	50	0.0	2	69.8	4520	1345	82.9	3
sp-os-rc	0.3	0.2	1.6	90.1	5.2	6542	99.2	52	0.0	0	0	0	0	0	0
sp-wt-bc	0.5	0.4	7.1	TO	39.2	7639	99.3	51	0.0	1	31.6	3682	1011	90.3	4
sp-wt-bk	0.5	0.4	2.1	TO	37.5	7640	99.3	50	0.0	1	30.0	2733	1029	87.8	3
sp-wt-cl	4.4	1.7	TO	116.8	79.1	4064	93.4	269	20.1	0	0	0	0	0	0
sp-wt-cn	0.5	0.3	3.6	MO	TO	-	-	-	-	-	-	-	-	-	-
sp-wt-cs	0.4	0.3	2.5	TO	TO	-	-	-	-	-	-	-	-	-	-
sp-wt-csf	1.1	0.4	5.1	TO	22.1	7645	99.3	53	0.0	1	13.8	778	778	100	1
sp-wt-csv	1	0.4	5.8	TO	23.6	7639	99.3	51	0.0	1	15.4	823	823	99.9	1
sp-wt-hc	0.7	0.5	30.1	TO	93.6	7640	99.3	50	0.0	1	85.8	3514	2038	53.1	3
sp-wt-ks	0.9	0.5	114.0	TO	137.0	7639	99.3	49	0.0	1	129.0	4701	2861	48.1	3
sp-wt-lf	0.6	0.4	2.9	TO	78.0	7640	99.3	50	0.0	1	70.5	5596	1962	54.4	5
sp-wt-rb	0.5	0.4	8.1	TO	57.3	7639	99.3	51	0.0	1	49.7	4266	1207	87.4	4
sp-wt-rc	0.4	0.3	1.6	86.9	7.6	7753	99.3	54	0.0	0	0	0	0	0	0

Symmetric Core Learning for Pseudo-Boolean Optimization by Implicit Hitting Sets

Hannes Ihalainen
University of Helsinki, Finland

Jeremias Berg
University of Helsinki, Finland

Matti Järvisalo
University of Helsinki, Finland

Bart Bogaerts
KU Leuven, Belgium
Vrije Universiteit Brussel, Belgium

Abstract

We propose symmetric core learning (SCL) as a novel approach to making the implicit hitting set approach (IHS) to constraint optimization more symmetry-aware. SCL has the potential of significantly reducing the number of iterations and, in particular, the number of calls to an NP decision solver for extracting individual unsatisfiable cores. As the technique is focused on generating symmetric cores to the hitting set component of IHS, SCL is generally applicable in IHS-style search for essentially any constraint optimization paradigm. In this work, we focus in particular on integrating SCL to IHS for pseudo-Boolean optimization (PBO), as earlier proposed static symmetry breaking through lex-leader constraints generated before search turns out to often degrade the performance of the IHS approach to PBO. In contrast, we show that SCL can improve the runtime performance of a state-of-the-art IHS approach to PBO and generally does not impose significant overhead in terms of runtime performance.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Theory of computation → Constraint and logic programming

Keywords and phrases Implicit hitting sets, symmetries, unsatisfiable cores, pseudo-Boolean optimization

Digital Object Identifier 10.4230/LIPIcs.CP.2025.15

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.15630156`

Funding *Hannes Ihalainen*: Partially funded by Research Council of Finland under grant 356046.
Jeremias Berg: Research Council of Finland under grant 362987.
Matti Järvisalo: Partially funded by Research Council of Finland under grant 356046.
Bart Bogaerts: Partially funded by Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G064925N) and by the European Union (ERC, CertiFOX, 101122653). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

1 Introduction

Symmetries are intrinsically present in various types of computationally hard decision and optimization problems, ranging from, e.g., scheduling and timetabling through the resolution of mathematical conjectures by automated reasoning to the analysis of systems broadly construed. As automated reasoning and constraint optimization solvers constitute the de facto approach in such settings, different ways of making solvers more symmetry-aware

© Hannes Ihalainen, Jeremias Berg, Matti Järvisalo, and Bart Bogaerts;
licensed under Creative Commons License CC-BY 4.0

have been developed: techniques that break symmetries in a preprocessing phase [2, 19, 3], techniques that learn such breaking constraints on the fly [26, 24, 41], techniques that guarantee asymmetric branching inside a solver [32], and techniques that extend the conflict learning mechanism applied during search within a solver [6, 18]. All these techniques aim to avoid unnecessary exploration of symmetric parts of search spaces underlying declarative models of real-world problems.

In this work, we propose symmetric core learning (SCL) as a novel approach to making the implicit hitting set approach (IHS) [30, 31, 27, 35] to constraint optimization more symmetry-aware. Our motivation for focusing on IHS is two-fold. Firstly, IHS offers a generic framework for developing constraint optimization solvers; IHS has been shown to yield efficient solvers for various constraint optimization languages, including maximum satisfiability (MaxSAT) [12, 13, 14, 33], MaxSAT modulo theories [11, 21], quantified MaxSAT [23, 28] answer set programming [34], finite-domain constraint optimization [16, 15, 25], and, among the most recent ones, pseudo-Boolean optimization (PBO) [37, 38]. Secondly, what sets IHS apart from other typical solving approaches is that IHS iteratively combines two types of constraint solvers. The first solver is a decision solver for the constraint language at hand, used for identifying so-called unsatisfiable cores expressing inconsistencies in the input formula. The second solver is a hitting set optimizer, typically implemented through an integer programming (IP) solver, for identifying ways of ruling out the so-far identified cores from consideration towards an optimal solution of the original problem. As we will explain, the particular way in which IHS search is structured offers a novel way of integrating symmetry-awareness to IHS-based approaches to constraint optimization, referred to as symmetric core learning, or SCL for short.

Symmetric core learning employs knowledge of instance-level symmetries extracted before search, and can be considered a new type of a dynamic symmetry exploitation technique in the style of symmetric explanation learning [18], with the key insight that learning symmetric images of cores extracted within IHS iterations can benefit the search process. A key difference of SCL compared to lex-leader symmetry breaking (which is in many domains the current state-of-the-art symmetry handling technique) is that, whereas lex-leader symmetry breaking adds symmetry breaking constraints to an input instance before invoking a solver in hope that some of the constraints might prove useful during search, SCL does not bloat the input instance with *potentially* useful symmetry breaking constraints. Instead, SCL employs symmetry information on the fly during IHS search to generate likely beneficial symmetric cores. As a high number of (symmetric) cores may need to be extracted in the worst case for termination, SCL has the potential of speeding up IHS by avoiding a high number of potentially time-consuming calls to the core-extracting decision solver, and instead can produce symmetric cores with polynomial delay from the cores extracted. Specifically, as the subsequently-computed hitting sets over the cores are required to also relax away the sources of inconsistencies expressed by the symmetric cores generated with SCL, SCL has potential for significantly lowering the number of IHS iterations needed for termination.

We focus on pseudo-Boolean optimization, also known as 0–1 IP solving, for which IHS is a competitive solving approach [38]. As we will empirically show, earlier-proposed static lex-leader symmetry breaking applied before search generally degrades the performance of IHS for PBO. This also holds for the recently-proposed extension of symmetry breaking to so-called weak symmetries broken by enforcing dominance constraints that ensure the preservation of optimal solutions [40]. This motivates the study of alternative ways of integrating symmetries into IHS search for PBO. Symmetric core learning provides one such alternative approach to making use of symmetries in IHS. At the same time, SCL makes use of weak symmetries

which discount the impact of the objective function at hand without needing dominance constraints. Beyond explicitly generating symmetric cores, we also investigate a compact representation of symmetric cores that has the potential of representing an exponential set of symmetric cores as a single abstract core [7] over a set of extension variables by making use of linear constraints native to the IP solver. In contrast to lex-leader symmetry breaking, we show that SCL can improve the runtime performance of a state-of-the-art IHS approach to PBO, and generally does not impose significant runtime overheads. Complementing the empirical results, we also point out fundamental differences in the effects of lex-leader symmetry breaking and SCL on IHS.

2 Preliminaries

2.1 Pseudo-Boolean Optimization

A *literal* ℓ is a Boolean variable x or its negation $\overline{x} = 1 - x$, where variables take values 0 (false) or 1 (true). A *pseudo-Boolean (PB) constraint* C is a 0–1 linear inequality $\sum_i a_i \ell_i \geq A$, where a_i and A are integers. Without loss of generality, we often assume PB constraints to be in *normal form*, meaning that the literals ℓ_i are over distinct variables, each *coefficient* a_i is *positive* and the *degree (of falsity)* A is non-negative. A *pseudo-Boolean formula* F is a conjunction $\bigwedge_j C_j$ of PB constraints. When convenient, we view F as a *set* of constraints. For a constraint C, $\text{LIT}(C)$ is the set of literals that appear in C. For a pseudo-Boolean formula F, $\text{LIT}(F) = \bigcup_{C \in F} \text{LIT}(C)$. An *objective* O is an expression $\sum_i w_i \ell_i + lb$ where the coefficients w_i and the constant lb are integers (lb stands for "lower bound": when the w_i are all possible, this constant term is indeed a lower bound on the cost).

A *substitution* (sometimes also called a *witness*) ω is a mapping of variables to 0, 1 and literals. An *assignment* α is a substitution that maps each variable in its domain into $\{0, 1\}$. An assignment is *complete* for F if it assigns a value to each variable of interest (where the set of variables should be clear from the context). We write $\omega(C)$ for the constraint obtained from C by replacing each variable x in the domain of ω by $\omega(x)$ (and implicitly normalizing); $\omega(O)$, $\omega(\ell)$, $\omega(F)$ are defined analogously.

A (normalized) PB constraint $C = \sum_i a_i \ell_i \geq A$ is *trivial* if $A \leq 0$ and *contradictory* if $\sum_i a_i < A$. The constraint C is *satisfied* by α (denoted $\alpha \models C$) if $\alpha(C)$ is trivial. A PB formula is satisfied if all of its constraints are satisfied. An instance of the *pseudo-Boolean optimization problem* is a tuple (F, O) with F a PB formula and O an objective to minimize. A complete assignment α is a *solution* of (F, O) if $\alpha \models F$ and is optimal if also $\alpha(O) \leq \beta(O)$ for each solution β of (F, O). The optimal cost of (F, O) is $\alpha(O)$ for an optimal solution of (F, O).

Following [37], a constraint C is an *unsatisfiable core* of (F, O) if the following conditions hold: (i) the literals in C are objective literals or their negations and (ii) all assignments α that satisfy F also satisfy C (denoted by $F \models C$). In other words, an unsatisfiable core is an implied constraint that only mentions objective literals.

▶ Remark 1. This definition of unsatisfiable core might sound somewhat unconventional; its naming has grown historically as follows. Originally an "unsatisfiable core" was defined a subset of the original formula that is unsatisfiable. In the context of assumption-based SAT solving (also employed for SAT-based optimization) fresh variables are used to represent constraints in the input formula and such a core became "a subset of the assumptions that cannot jointly be set to a specific value" (or alternatively, a clause over assumption variables learned by the solver). In the PBO setting, this was then further generalized to be any PB constraint over assumptions (objective variables) learned by the PB solver.

The following proposition forms the basis for how the IHS approach to PBO uses cores to compute optimal solutions.

▶ **Proposition 2** (e.g. [37]). *Let (F, O) be a PBO instance, α_{best} an optimal solution to (F, O), $\mathcal{K}$ a set of cores of (F, O), and γ an optimal solution to $(\mathcal{K}, O)$. Then $\gamma(O) \leq \alpha_{best}(O)$.*

In words, Proposition 2 states that the cost of the optimal (minimum-cost) solutions to any set of cores is a lower bound on the optimal cost of the instance.

2.2 Symmetries in Pseudo-Boolean Optimization

A substitution σ is called a (syntactic) *weak symmetry of* (F, O) if $\sigma(F) = F$. If $\sigma(O) = O$ further holds, σ is a (strong) symmetry. For strong symmetries, α is an optimal solution of (F, O) if and only if $\alpha \circ \sigma$ is an optimal solution of (F, O). We also consider a subset of weak symmetries that we call *core-preserving symmetries*. A weak symmetry σ is core-preserving if $\sigma(\ell)$ is an objective literal if and only if ℓ is. Similarly to weak symmetries, and in contrast to strong ones, a core-preserving symmetry σ is not guaranteed to preserve the cost of solutions, i.e., $\alpha(O)$ need not equal $\alpha \circ \sigma(O)$. In contrast to weak symmetries, however, core-preserving symmetries are guaranteed to map cores to cores, i.e., $\sigma(C)$ is a core of (F, O) if and only if C is. If Σ is a set of symmetries, we write $\langle\Sigma\rangle$ for the group generated by Σ. If Σ is a set of symmetries acting on a set S (e.g., acting on the set of cores as just defined), the *orbit* of $s \in S$ under Σ, denoted $\mathcal{O}(s, \Sigma)$, is the set $\{\sigma(s) \mid \sigma \in \langle\Sigma\rangle\}$.

When symmetries are present, they can slow down search of a PBO solver, causing the solver to explore many symmetric subareas of the search space of solutions. One way to deal with symmetric search spaces is to add symmetry-breaking constraints. Specifically, given a (strong) symmetry σ, a common approach is to add a (set of) constraint(s) that is satisfied by an assignment α if and only if α is lexicographically smaller than (or equal to) $\alpha \circ \sigma$ [1]. This can be achieved with a single pseudo-Boolean constraint LL_σ (called a *lex-leader* constraint) of the form

$$\sum_i 2^i \cdot x_i \leq \sum_i 2^i \cdot \sigma(x_i)$$

or by expressing the constraint as a set of constraints with smaller coefficients using auxiliary variables (see for instance [19]). Since weak symmetries do not preserve the costs of solutions, lex-leader constraints over a weak symmetry might change the optimal cost of a PBO instance. Van Caudenberg and Bogaerts [40] generalized the lex-leader constraint to weak symmetries, which then becomes a *dominance constraint*

$$2^{n+1} \cdot O + \sum_{i=1}^{n} 2^i \cdot x_i \leq 2^{n+1} \cdot \sigma(O) + \sum_{i=1}^{n} 2^i \cdot \sigma(x_i),$$

which states that the objective value must be smaller than or equal to the objective of the symmetric assignment, and, furthermore, *if* the objective values are equal, the assignment must be lexicographically smaller than its symmetric counterpart.

An alternative to defining symmetries as *substitutions* is to define symmetries as *permutations of literals* respecting negation, i.e., via a permutation π of the literals such that $\pi(\overline{x}) = \overline{\pi}(x)$ for each x. We will use *disjoint cycle* notation and write, for instance, $(x\overline{y}z)$ (or $(x\overline{y}z)(\overline{x}y\overline{z})$ if we also want to spell out the redundant information) for the symmetry that maps x to $\overline{y}$, $\overline{y}$ to z and z to x and all other variables to themselves.

Algorithm 1 PBO-IHS. Text highlighted in yellow marks the symmetric core learning extension.

```
1  PBO-IHS(F, O)
       Input: A PBO instance (F, O)
       Output: An optimal solution α_best
2      (α_best, sat?) ← PB-Solve(F);
3      if not sat? then
4          return "no feasible solutions";
5      ub ← α_best(O); lb ← −∞; K ← ∅;
6      Σ ← Compute-Symmetries(F) ;
7      while TRUE do
8          γ ← Min-Sol(K, O);
9          lb ← γ(O);
10         if ub = lb then break;
11         (C, α′, ub′) ← Extract-Core(γ, F, O);
12         if ub′ < ub then ub ← ub′; α_best ← α;
13         if ub = lb then break;
14         K ← SCL(C, K, Σ) ;
15         K ← K ∪ {C} ∪ K ;
16     return α_best;
```

Min-Sol$(\mathcal{K}, O)$:
minimize: O
subject to:

$$C \qquad \forall C \in \mathcal{K}$$
$$\ell \in \{0,1\} \qquad \forall \ell \in \text{LIT}(\mathcal{K})$$

return:
$\{\ell \mid \ell = 1 \text{ in opt. soln}\} \cup$
$\{\overline{\ell} \mid \ell = 0 \text{ in opt. soln}\}$

Figure 1 Hitting set IP.

▶ **Example 3.** Let $F = \{x + \overline{y} \geq 1, 2\overline{x} + y + z \geq 2, 2y + \overline{x} + z \geq 1\}$ and $O = x + y$. Then $\sigma = (x\overline{y})$ is a weak symmetry of (F, O) since applying σ to F swaps the last two constraints; σ is neither a core-preserving symmetry since σ maps the objective literal x to the non-objective literal $\overline{y}$, nor a strong symmetry since $\sigma(O) = \overline{y} + \overline{x}$ which is not $x + y$.

In practice, there can be exponentially many symmetries and hence breaking all of them, e.g., via lex-leader constraints, is in general infeasible. The most common way of dealing with this is to simply break a set of *generators* of the symmetry group, without formal guarantees on completeness, i.e., on that all symmetries would be broken. A set of generators is typically detected via a *graph* representing a simplification of the syntax tree of the formula and computing automorphisms of this graph [2]. Some tools go further then detecting an arbitrary set of generators, aiming to identify *structure* in the symmetry group [19, 4, 3]. As an example, these tools aim to detect *row interchangeability* [17], representing a symmetry as a matrix M of literals such that each two rows of the matrix are interchangeable in the sense that σ_{ij} maps every literal M_{ik} to M_{jk}, each M_{jk} to M_{ik}, and other literals to themselves. If one uses the "right" set of generators, and the "right" order of the variables at hand, the entire group generated by such a matrix can be broken with a polynomially-sized set of symmetry breaking constraints.

2.3 The Implicit Hitting Set Approach to PBO

Algorithm 1 details PBO-IHS, the implicit hitting set algorithm for pseudo-Boolean optimization [37, 38]. The highlighted lines correspond to the symmetric core learning extension we detail in Section 3 and should be ignored in this section.

Given a PBO instance (F, O), `PBO-IHS` begins by checking the existence of solutions of (F, O) by invoking PB decision procedure `PB-Solve` on F (Line 2). The call returns an indicator *sat?* for satisfiability and a solution α_{best} in the positive case. Given α_{best}, an upper bound ub and a lower bound lb on the optimal cost of the instance are initialized to $\alpha_{best}(O)$ and $-\infty$, respectively, on line 5. During the search, α_{best} will always contain the currently best-known solution for which $\alpha_{best}(O) = ub$. A set $\mathcal{K}$ of cores of the instance is initialized to $\emptyset$ and the main search loop (lines 7-15) entered. The main search loop iterates until $lb = ub$, at which point α_{best} is known to be optimal. Each iteration begins with the computation of an optimal solution γ of the instance $(\mathcal{K}, O)$ consisting of the cores found so far and the objective (Line 9). In practice, γ is computed by solving the hitting set IP in Figure 1 with an integer programming (IP) solver. By Proposition 2, $\gamma(O)$ will be a lower bound on the optimal cost of the instance. Since no cores are ever removed from $\mathcal{K}$, the values of $\gamma(O)$ will be non-decreasing over the iterations, so lb can always be updated on Line 9. If the algorithm does not terminate on Line 10, the procedure `Extract-Core` next computes a new core C of (F, O) not satisfied by γ. This is achieved by invoking a decision procedure for PB constraints of F under the assumptions γ. The result of this call will either be a solution that extends γ, or a constraint C (learned using standard conflict analysis) falsified by γ but for which $F \models C$.

Each extracted core in IHS witnesses that the current γ cannot be extended to an optimal solution of the original instance. As a byproduct, `Extract-Core` also often obtains a solution α' of cost $\alpha'(O) = ub'$, which it also returns. The cost of α' can (but need not) be lower than the current upper bound; hence, termination is checked on Line 13. If the algorithm does not terminate, the new core is added to $\mathcal{K}$, and IHS reiterates.

3 Symmetric Core Learning for IHS

We detail *symmetric core learning* (SCL) for IHS-based PBO solving as our main contribution.

3.1 The Basic Idea

SCL is based on the observation that given a PBO instance (F, O), a core C, and a core-preserving symmetry σ, the constraint $\sigma(C)$ obtained by applying σ to C is also a core of (F, O). Symmetric core learning uses core-preserving symmetries of a PBO instance to generate new cores *dynamically during search*, with the key aims of decreasing the number of calls to the `Extract-Core` and `Min-Sol` procedures of `PBO-IHS`, while at the same time avoiding the generation of (a potentially large number) of lex-leader constraints before search.

▶ **Example 4.** Consider the PBO instance (F, O) with $F = \{x_i + y_j \geq 1 \mid i = 1, \ldots, n, j = 1, \ldots, m\}$, $O = \sum_{i=1}^{n} x_i + \sum_{j=1}^{m} y_j$ and $n \leq m$. Examples of core-preserving symmetries of (F, O) are $\sigma_1 = (x_1 x_2 \ldots x_n)$ and $\sigma_2 = (y_1 y_2 \ldots y_n)$. The core $C = x_1 + y_1 \geq 1$ and the symmetry σ_1 together imply the cores $\sigma_1(C) = x_2 + y_1 \geq 1$, $\sigma_1^2(C) = \sigma_1(\sigma_1(C)) = x_3 + y_1 \geq 1$, $\ldots$, $\sigma_1^{n-1}(C) = x_n + y_1 \geq 1$.

The IHS approach to PBO extended with symmetric core learning is obtained by including the highlighted lines of Algorithm 1. During the initialization phase, the algorithm computes a set of core-preserving symmetries of the PBO instance to be solved on Line 6. The symmetries are used during each iteration of the search loop by the `SCL` procedure on Line 14 to generate a set K of cores that are symmetric to the core C computed by `Extract-Core`, with the aim of increasing the number of cores added to $\mathcal{K}$ in each iteration of IHS search.

Algorithm 2 Explicit symmetric core learning.

```
1 Explicit-SCL(C, 𝒦, Σ)
    Input: A core C, a set of all so-far extracted cores 𝒦, and a set of symmetries Σ.
    Output: Set S of cores implied by C and the symmetries in Σ.
2   S ← {C};   coreQueue.init(C);
3   while coreQueue.size()>0 do
4     C ← coreQueue.pop();
5     for σ ∈ Σ do
6       if σ(C) ∉ 𝒦 ∪ S then
7         S ← S ∪ {σ(C)};   coreQueue.push(σ(C));
8         if |S| ≥ constrLim or Σ_{C∈S} |LIT(C)| ≥ litLim then return S;
9   return S;
```

An intuition underlying symmetric core learning is that if we have a symmetry σ and a core C, then computing a symmetric core $\sigma(C)$ is very fast. In contrast, extracting cores with the subroutine `Extract-Core` invokes a decision procedure for an NP-hard constraint language, which could require a significant amount of time. Furthermore, in theory, a single symmetry can be used to generate many different symmetric cores. Thus, we expect symmetric core learning to be efficient for solving instances where the time required to compute core-preserving symmetries and invoking `SCL` is lower than the time needed to extract enough cores to terminate using only `Extract-Core`.

We detail two variants of symmetric core learning, i.e., instantiations of the `SCL` procedure of Algorithm 1. In *explicit symmetric core learning*, detailed in Section 3.2, the generated cores are explicitly added to the accumulated set of cores and thus as individual constraints to the hitting set IP. In *compact symmetric core learning*, detailed in Section 3.3, the generated cores are instead more compactly encoded with a small number of linear constraints that are then added to the hitting set IP instead of the cores themselves.

3.2 Explicit Symmetric Core Learning

In explicit symmetric core learning, the cores generated via symmetries are directly (or explicitly) added to the hitting set IP as individual constraints. Algorithm 2 details our breadth-first style implementation of explicit symmetric core learning. Given a core C computed by `Extract-Core` in the current iteration of IHS search, the set $\mathcal{K}$ of all cores found during search and a set Σ of core-preserving symmetries, `Explicit-SCL` first initialises a set S of cores that are symmetric to C, and a queue *coreQueue* of cores to be processed on Line 2. Then, the algorithm enters the while-loop between Lines 3 and 8, during which the cores in *coreQueue* are processed in breadth-first order as follows. For a core C, the algorithm iterates through each symmetry $\sigma \in \Sigma$. If $\sigma(C)$ is a previously unseen core, $\sigma(C)$ is added to the set S and pushed into *coreQueue* to be processed later. After adding a new core to S the algorithm checks for two termination criteria: whether $|S|$, the number of new symmetric cores that have been generated, exceeds a user-defined parameter *constrLim*, and whether $\sum_{C \in S} |\text{LIT}(C)|$, the sum of the number of literals in the new symmetric cores, exceeds the parameter *litLim*. As detailed in the following, these early termination criteria are employed to escape situations in which Algorithm 2 would otherwise produce an excessive number of symmetric cores to the point that overall search performance could suffer.

A central challenge in realizing explicit symmetric core learning is that naively applying symmetries to cores may result in a very large set of cores that do not help the IHS search terminate faster. More precisely, while a single core with n literals can, in theory, generate a number of symmetric cores exponential in n (e.g., in a degenerate case where any permutation of variables is a symmetry), the following observations demonstrate that the effect of explicit symmetric core learning on IHS search depends on the PBO instance. The first observation demonstrates how explicit symmetric core learning can reduce the number of iterations of the IHS main loop.

▶ **Observation 5.** *Consider the PBO instance (F, O), the symmetries σ_1 and σ_2 from Example 4, and assume for the example that* `Extract-Core` *only returns at-least-one constraints (i.e., propositional clauses) as cores. An optimal solution α to (F, O) assigns exactly the n variables x_i to 1. As every at-least-one core extracted results in at most one new variable being assigned to 1 by solutions returned by* `Min-Sol`*,* `PBO-IHS` *without symmetric core learning (i.e. Algorithm 1 without the highlighted parts) invokes the* `Extract-Core` *and* `Min-Sol` *subroutine at least n times before termination. In contrast, given the symmetries σ_1, σ_2, and a single core of form $x_i + y_j \geq 1$ for some i and j, Algorithm 2 can generate $n \times m$ symmetric cores, allowing* `PBO-IHS` *with symmetric core learning (i.e. Algorithm 1 with the highlighted parts) to terminate after a single invocation of* `Extract-Core` *and two invocations of* `Min-Sol`*.*

The next observation demonstrates that IHS with unrestricted explicit symmetric core learning can lead to the extraction of unnecessarily many cores during IHS search.

▶ **Observation 6.** *Assume that, invoked on the PBO instance (F, O) from Example 4, the IHS algorithm has extracted the n cores $x_i + y_i \geq 1$ for $i = 1, \ldots, n$. In the next call to* `Min-Sol`*, one of the minimum-cost solutions that can be returned assigns all x_i variables to 1 and all y_j variables to 0. Given this solution, the next call to* `Extract-Core` *will not find a core, and hence IHS terminates after extracting n cores. In contrast, as detailed in Observation 5, IHS with unrestricted explicit SCL and the symmetries $\sigma_1 = (x_1 \ldots x_n)$ and $\sigma_2 = (y_1 \ldots y_m)$ is guaranteed to add $n \times m$ cores to $\mathcal{K}$ before termination.*

In contrast to the simple instance in Observation 6, in practical settings adding too many constraints to the hitting set IP can significantly increase the time needed to compute a minimum-cost solution. Identifying which cores allow IHS to make fast progress towards termination is an open research challenge. We limit explicit SCL in attempt to balance between harnessing the potential of symmetric cores to speed up search (Observation 5) and mitigating the risk of having to spent unnecessary effort to extract cores (Observation 6).

3.3 Compact Symmetric Core Learning

Compact symmetric core learning aims to express a large number of symmetric cores with a smaller number of constraints in the hitting set IP, making use of linear constraints native to IP. The following examples provide intuition for the approach.

▶ **Example 7.** Consider again the PBO instance (F, O), the core $C = x_1 + y_1 \geq 1$, and the symmetries σ_1 and σ_2 from Example 4. Using two fresh auxiliary Boolean variables s_x and s_y all of the $n \times m$ cores implied by C and σ_1 and σ_2 can be captured by the constraints $n \cdot s_x \leq x_1 + \cdots + x_n$, $m \cdot s_y \leq y_1 + \cdots + y_m$ (which enforces s_x (s_y) to be smaller than each of the x_i (y_i)), and $s_x + s_y \geq 1$.

As the minimum-cost solutions to the extracted cores are computed with an IP solver, we are not restricted to using binary variables for expressing a set of cores. Indeed, integer variables can be employed for representing sets of symmetric cores more compactly.

▶ **Example 8.** Consider a PBO instance with two symmetries $\sigma_1 = (x_1x_2x_3)(y_1y_2y_3)$ and $\sigma_2 = (z_1z_2z_3)(w_1w_2w_3)$. Given the core $C = x_1+2y_1+z_1+w_1 \geq 3$, the group generated by σ_1 and σ_2 yields the symmetric cores $x_i + 2y_i + z_j + w_j \geq 3$ for $i = 1, \ldots, 3$ and $j = 1, \ldots, 3$. All these cores can be represented compactly with the constraints $s_{xy} \leq x_i + 2y_i$ (for $1 \leq i \leq 3$), $s_{zw} \leq w_i + z_i$ (for $1 \leq i \leq 3$), and $s_{xy} + s_{zw} \geq 3$, where s_{xy} and s_{zw} are (non-binary) *integer* variables.

We continue with a high-level description of compact SCL. Assume that we partition a core $C = \sum_\ell a^\ell \ell \geq B$ as $\sum_{\ell \in S} a^\ell \ell + \sum_{\ell \notin S} a^\ell \ell \geq B$ for a subset S of its literals. Now assume a set Σ of symmetries for which $\sigma(\sum_{\ell \notin S} a^\ell \ell) = \sum_{\ell \notin S} a^\ell \ell$ for each $\sigma \in \Sigma$. Consider the orbit $\mathcal{O}(\sum_{\ell \in S} a^\ell \ell, \Sigma)$ of the partition of the constraint containing the literals in S. Compact symmetric core learning generates a compact representation for a set of cores implied by C and Σ by replacing $\sum_{\ell \in S} a^\ell \ell$ with an integer variable s_S and the definition $s_S \leq \min(\mathcal{O}(\sum_{\ell \in S} a^\ell \ell, \Sigma))$. In the rest of this section, we fill in the details of this high-level description of compact SCL.

As illustrated in Examples 7 and 8, compact SCL can substitute several subsets of a core with new variables and in doing so represent many symmetric cores more compactly. What is happening in these examples is that (certain) subgroups of the group of all symmetries act independently on parts of the core rather than on the whole core. However, as the following observation shows, independent rewriting of parts of the cores is not sound in case symmetries interact too much. In the rest of this section, we will formalize conditions that guarantee the soundness of rewriting parts of the core.

▶ **Observation 9.** *Assume that a PBO instance has the symmetries $\sigma_1 = (x_1x_2x_3)(y_4y_2y_3)$, $\sigma_2 = (y_1y_2y_3)(x_4x_2x_3)$ and the core $C = x_1 + y_1 \geq 1$. The orbit of x_1 under $\{\sigma_1\}$ is $\{x_1, x_2, x_3\}$ and the orbit of y_1 under $\{\sigma_2\}$ is $\{y_1, y_2, y_3\}$. Compact SCL could introduce both $s_x + y_1 \geq 1$, $s_x \leq \min(\{x_1, x_2, x_3\})$ and $x_1 + s_y \geq 1$, $s_y \leq \min(\{y_1, y_2, y_3\})$. Introducing $s_x + s_y \geq 1$ with $s_x \leq \min(\{x_1, x_2, x_3\})$ and $s_y \leq \min(\{y_1, y_2, y_3\})$, however, is not correct since these would also represent the constraint $x_2 + y_2 \geq 1$ that is not a core implied by C, σ_1 and σ_2.*

Observation 9 demonstrates that the choice of symmetries under which to compute orbits can not be done independently for all subsets to be replaced. One way to interpret Observation 9 is that if we wish to introduce a new variable defined by the orbit $\mathcal{O}_2$ into a core that already has another new variable defined by the orbit $\mathcal{O}_1$, we need to ensure that the symmetries used to compute the orbit $\mathcal{O}_2$ "behave well" together with those used to compute $\mathcal{O}_1$. The following definition formalizes this notion of well-behavedness.

▶ **Definition 10.** *Let C be a core and assume that Σ_i is a set of symmetries for each $i \leq k$. We call $\langle \Sigma_i \mid 1 \leq i \leq k \rangle$ a symmetry decomposition of C if C can be written as $L_1 + \cdots + L_k \geq A$ and the following conditions hold:*

1. *For each $\sigma_i \in \Sigma_i$, $\sigma_i(\sum_{j>i} L_j) = \sum_{j>i} L_j$.*
2. *For each $\sigma_i \in \Sigma_i$, each $j < i$, and each $L'_j \in \mathcal{O}(L_j, \Sigma_j)$, it holds that $\sigma_i(L'_j) \in \mathcal{O}(L_j, \Sigma_j)$.*

For intuition on Definition 10, assume that we are computing a compact representation for a set of cores symmetric to C partitioned as $L_1 + \cdots + L_k \geq A$. If we process the partitions in the order of $L_1, L_2, \ldots, L_k$, then a symmetry decomposition of C defines which symmetries we apply to generate the orbit $\mathcal{O}_i$ for each L_i. To be a symmetry decomposition, we should be able to split our core so that each symmetry in the i^{th} set stabilizes the parts of the core beyond i as well as the orbits in all of the earlier parts of the core. These conditions together guarantee that every constraint represented by the compact representation is a core.

▶ **Proposition 11.** *Let $\langle \Sigma_i \mid 1 \leq i \leq k \rangle$ be a symmetry decomposition of $C = L_1 + \cdots + L_k \geq A$. If $L'_i \in \mathcal{O}(L_i, \Sigma_i)$ for all i, then $L'_1 + \cdots + L'_k \geq A$ is a core.*

Proof Sketch. This core can be constructed iteratively from k to 1 as follows. Since $L'_k \in \mathcal{O}(L_k, \Sigma_k)$, there is a symmetry σ_k such that $\sigma_k(L_k) = L'_k$ and hence $\sigma_k(L_1) + \cdots + \sigma_k(L_{k-1}) + L'_k$ is a core. The second item of Definition 10 guarantees that there exists a symmetry σ_{k-1} that stabilizes L_k and such that $\sigma_{k-1}(L_{k-1}) = \sigma_k^{-1}(L'_{k-1})$. As a consequence, $\sigma_k \circ \sigma_{k-1}(C) = \sigma_k \circ \sigma_{k-1}(L_1 + \cdots + L_{k-2}) + L'_{k-1} + L'_k$. Iteratively applying this construction yields the desired core. ◀

In order to use a symmetry decomposition, we will replace each L_i by new variables intuitively to represent all its symmetric images at once. Before giving the formal definition, we give one more example of how to get even more compact encodings.

▶ **Example 12.** Consider a PBO instance where $x_1, \ldots, x_n$ are interchangeable without touching the term $y_1 + 2y_2 + 3y_3$. In other words, assume that the group of stabilizers of $y_1 + 2y_2 + 3y_3$ acts symmetrically on $\{x_1, \ldots, x_n\}$ (meaning that for each i, j, it contains a permutation that swaps x_i and x_j and maps each other x_k to itself). Additionally, assume that $x_1 + 2x_2 + 3x_3 + y_1 + 2y_2 + 3y_3 \geq 5$ is a core. In that case, there are cubically many cores: for each triple of different values of i, j and k, $x_i + 2x_j + 3x_k + y_1 + 2y_2 + 3y_3 \geq 5$ is a core. We can introduce three counter variables: v_1, v_2, and v_3 such that v_i is false if at least i of the x variables are false using $(n - i + 1)v_i \leq x_1 + x_2 \cdots + x_n$. Given these three counting constraints, all the cores are captured by the single core $v_3 + 2v_2 + 3v_1 + y_1 + 2y_2 + 3y_3 \geq 5$.

We are now ready to formally define the compact encoding of cores that we use.

▶ **Definition 13.** *Let $\langle \Sigma_i \mid 1 \leq i \leq k \rangle$ be a symmetry decomposition of $C = L_1 + \cdots + L_k \geq A$. Assume that $L_i = \sum_j w_j \ell_j$ (where the coefficients w_j are wlog assumed to be decreasing). The* compact encoding *of L_i consists of:*

Case 1: if $\langle \Sigma_i \rangle$ acts symmetrically on a set L that includes all ℓ_j. *(1) constraints $(|L_i| - j + 1)c_{i,j} \leq \sum_j \ell_j$ defining fresh counting variables $c_{i,j}$ (for $1 \leq j \leq |L_i|$): and (2) replacing L_i by $\sum_j w_j c_{i,j}$ in C;*

Case 2: otherwise *(1) constraints $V_i \leq L'_i$ introducing a fresh integer variables V_i for each $L'_i \in \mathcal{O}(L_i, \Sigma_i)$, and (2) replacing L_i by V_i in C.*

Case 1 of Definition 13 was shown in Example 7, Case 2 in Example 8. In the worst case, i.e., when the **Case 1** optimization is not used, compact SCL uses $1 + \sum_i |\mathcal{O}(L_i, \Sigma_i)|$ constraints to represent $\prod_i |\mathcal{O}(L_i, \Sigma_i)|$ cores. (In the special case $|L_i| = 1$, we use 1 constraint instead of $|\mathcal{O}(L_i, \Sigma_i)|$ constraints to encode the semantics of V_i.)

Implementing compact SCL requires care and various heuristic choices. As a first way of realize compact SCL, we implemented a procedure which, given a core C and a set Σ of symmetries, heuristically computes different partitions of C and subsets of Σ as candidate symmetry decomposition of C. The procedure prioritizes finding symmetry groups that act symmetrically on a set of literals (to enable the Case 1 optimization of Definition 13). Our procedure for computing a symmetry decomposition of C and the orbits of a partition under it resemble explicit SCL as detailed in Algorithm 2. When orbits containing more than the proposed partition itself are found, C is modified by replacing the partition under consideration with a new variable defined based on the orbit. Technical details on our current implementation of this approach to implementing compact SCL are provided in Appendix A.

In contrast to explicit SCL, compact SCL can decrease the best-case number of constraints needed in the hitting set IP solved by `Min-Sol` during IHS search.

▶ **Observation 14.** *Consider the PBO instance (F, O) and the symmetries σ_1 and σ_2 from Example 4. Observation 6 illustrated that IHS search can terminate after having extracted n cores of form $x_i + y_i \geq 1$ for $i = 1, \ldots, n$. However, any assignment that sets exactly one from each pair x_i, y_i to 1 and all other variables to 0 is a minimum cost solution to the integer program formed over these n cores that* `Min-Sol` *solves. Only one of these 2^n minimum-cost solutions will results in termination of IHS search, namely the one that assigns all x_i to 1 and all y_i to 0. To guarantee termination using only cores of form $x_i + y_j \geq 1$ (i.e., the at-least-one cores of (F, O) that have a minimum number of literals), IHS (with or without explicit SCL) needs to add at least $2n$ of them to $\mathcal{K}$ since with any fewer, there is at least one x_i that appears only in one core $x_i + y_j \geq 1$, meaning that there is a minimum-cost hitting set that assigns x_i to 0 and y_j to 1. On the other hand, the set $\{x_i + y_i \geq 1 \mid 1 \leq i \leq n\} \cup \{x_i + y_{i+1} \geq 1 \mid 1 \leq i \leq n\}$ of $2n$ cores only has one minimum-cost hitting set, assigning all x_i to 1.*

In contrast, as detailed in Example 7, assuming the first core extracted is $x_1 + y_1 \geq 1$, compact SCL will result in the constraints $s_x \leq \min(\{x_i \mid i = 1, \ldots, n\})$, $s_y \leq \min(\{y_i \mid i = 1, \ldots, m\})$ and $s_x + s_y \geq 1$, where s_x and s_y are fresh binary variables. In total, compact SCL will in this case add 3 constraints to $\mathcal{K}$: (1) $ns_x \leq \sum_{i=1}^{n} x_i$, (2) $ms_y \leq \sum_{i=1}^{m} y_i$ and (3) $s_x + s_y \geq 1$. In the next iteration, the only minimum-cost solution that `Min-Sol` *can return sets all x_i and s_x to 1, and the other variables to 0. With this solution, IHS with compact SCL terminates with 3 constraints (that represent $n \times m$ cores) in $\mathcal{K}$.*

Compact SCL and Abstract Cores. Finally, we note that compact SCL has a similar flavor to a technique called *abstract cores* proposed for IHS in the context of maximum satisfiability [7]. An IHS solver using abstract cores will (i) heuristically select a subset S of objective literals, (ii) introduce new count variables o_k with the definition $o_k \leftrightarrow \sum_{\ell \in S} \ell \geq k$, and (iii) use the `Extract-Core` subroutine to extract so-called abstract cores, i.e., core constraints that contain both original objective literals and the new count variables. A central difference between abstract cores and compact SCL is that rather than using `Extract-Core` to prove that the new variables appear in cores, the way in which compact SCL chooses which objective literals should be grouped together is guaranteed to cover a large number of symmetric cores of the PBO instance.

3.4 SCL and Heuristic Optimizations in IHS

Practical implementations of IHS employ various additional heuristic optimizations which are central for runtime efficiency. This is also the case for PBO-IHS. The two heuristics that affect our implementation of SCL are forms of *hardening*, including reduced cost fixing [5] and *weight-aware core extraction* (WCE) [14, 8].

Hardening refers to fixing the value of an objective literal ℓ during search. There are two ways in which that can happen: (i) if the coefficient of ℓ is strictly greater than the current upper-bound ub, then the literal is fixed to 0 since setting it to 1 would lead to a worse solution than the current best known; or (ii) via reduced cost fixing [5], i.e., if the reduced cost of ℓ in the LP relaxation of the hitting set IP detailed in Figure 1 imply that ℓ is set either to 0 or 1 in the optimal solutions. Hardening affects SCL in that symmetries that map non-fixed literals to fixed ones are removed from Σ when symmetric cores are generated.

Weight-aware core extraction aims to delay calls to `Min-Sol` by extracting multiple cores falsified by a solution γ in each iteration. Intuitively this decreases the number of times that the hitting set IP needs to be solved [38, 37]. With SCL, we interleave the calls to the `SCL` procedure between the core-extraction steps: after `Extract-Core` obtains each new core C,

we compute symmetric cores to it and then invoke `Extract-Core` again, asking for a core that is falsified by γ and not any of the cores that have been computed in this iteration. In practice, this is achieved by removing at least one literal from every obtained core from the assumptions posed in the next query to `Extract-Core`. The `Min-Sol` procedure is invoked only when `Extract-Core` does not find more cores.

3.5 SCL and Symmetry Breaking

We end this section with a short observation comparing SCL and symmetry breaking with lex-leader constraints in the context of IHS. In contrast to SCL, symmetry breaking with lex-leader constraints does not result in the IHS search extracting more of the original cores of the PBO instance, but rather introduces new cores that change the search.

▶ **Observation 15.** *Consider the PBO instance (F, O) and the symmetry σ_1 detailed in Example 4 as a strong symmetry of (F, O). Applying lex-leader symmetry breaking on (F, O) adds the constraint $LL_x = \sum_{i=1}^{n} 2^i x_i \leq \sum_{i=1}^{n-1} 2^i x_{i+1} + 2^n x_1$. The constraints $x_1 + \overline{x_i} \geq 1$ for $2 \leq i \leq n$ are examples of cores of $(F \cup \{LL_x\}, O)$ that are not cores of (F, O). In other words, symmetry breaking can (implicitly) introduce new cores to the original instance.*

4 Experiments

To evaluate the impact of SCL, we implemented the PBO-IHS algorithm from scratch in C++, using RoundingSat 2 [20] as the PB core extractor, Gurobi 9.5.2 as the IP solver for hitting set computation, and BreakID [19] to compute symmetries and lex-leader constraints. Our PBO-IHS re-implementation (without symmetry breaking) outperforms the original PBO-IHS implementation [37, 38] on the benchmark set used in [37, 38], solving 948 instances against 920 for the implementation by the original authors. (For a comparison of PBO-IHS with other approaches, see [37, 38].) Our PBO-IHS implementation includes the optimizations described in [37, 38], including assumption shuffling, weight-aware core extraction (WCE) [5], solution-improving search at stagnation [38], reduced cost fixing [5], and hardening. Most relevant in terms of SCL are assumption shuffling and WCE which both concern core extraction: SCL (when used) is applied after assumption shuffling, and within the WCE loop, i.e., after each core extraction step. We evaluate the following variants of PBO-IHS.

- **Baseline**: our PBO-IHS implementation without symmetry techniques.
- **LL**: Baseline extended with lex-leader symmetry breaking.
- **LLS**: LL restricted to only break *strong* symmetries (see [40]).
- **ExplicitSCL**: Baseline extended with explicit SCL.
- **CompactSCL**: ExplicitSCL extended with compact SCL.

We use 100-second time limit for BreakID in the LL and SCL variants for computing symmetries, and 1 h for the overall runtime; reported runtimes include symmetry computation. For LL, we run BreakID with the same parameters as in of the original work on weak symmetries for lex-leader symmetry breaking for PBO [40] (the detection of row interchangeability and weak symmetry breaking are enabled). For LLS, weak symmetries are disabled. Based on preliminary experimentation, with ExplicitSCL and CompactSCL detection of row interchangeability is disabled. For the early termination heuristics, we use $constrLim = 100$ and $litLim = 6000$. These value choices were not rigorously tuned, but the parameters are important to cap the symmetric core generation, since in general there may be exponentially many symmetric cores. (It should be noted that LL also has similar limiting parameters to avoid overloading the solver with symmetry breaking constraints.) Our implementation

applies compact SCL only when the estimated number of cores covered is greater than the estimated number of auxiliary constraints needed. Explicit SCL is applied in each iteration after first applying compact SCL; when explicit SCL is applied to a compacted core, we only use symmetries that do not change the definitions of the count variables in the core.

The experiments reported on were run on 2.40-GHz Intel Xeon Gold 6148 machines with 381-GB RAM in RHEL under a 32-GB memory limit. Implementation code, data and benchmark generation scripts are available at `https://doi.org/10.5281/zenodo.15630156`.

4.1 Experiments on Highly Symmetric Benchmarks

We start by considering two examples of problem domains known to have a high number of symmetries intrinsically. The first, which we will refer to as the CC problem, asks to maximize the size of the largest clique over all graphs with n nodes that admit a k-coloring where n and k are given parameters. Additionally, we consider a variant of the problem in which all nodes are given a distinct weight from $1, \ldots, n$, and the task is to maximize sum-of-weights of the nodes in the clique. We will refer to this variant as "weighted CC" and talk about "unweighted CC" when the task is to maximize the size of the clique. In our experiments, we use an extension of the encoding from [29] to optimization, the full details of the encoding are in Appendix B. Beyond being highly symmetric (as explained in short), the CC problem draws motivations from proof complexity: tautologies of the type "a graph either does not contain a clique of size m, or is not $(m-1)$-colorable", expressed similarly to the encoding we use, are exponentially hard for cutting planes [29].

Note that the size of the largest clique in the graphs that admit a k-coloring is k. There are many symmetric variants of graphs with n nodes that admit both a k-coloring and a clique of size k. The encoding of CC results in PBO instances in which the objective contains n variables that are indicators for a node **not** being included in the clique. With this intuition, any at-least-one core corresponds to a subset of nodes that cannot all be included in the clique. Since the maximum size of the clique is k, it follows that any subset of nodes that contains $k+1$ nodes corresponds to a (subset-minimal) core. Termination of IHS using only cores of this form requires in total $\binom{n}{k+1}$ cores that rule out all such subsets from consideration.

The CC instances contain two types of easy-to-grasp symmetries: "node-symmetries" over the node indices, and "color-symmetries" over color indices. Both types are present in weighted and unweighted CC, but very notably, the node-symmetries that, informally speaking, map nodes to other nodes are strong only in the unweighted variant. To see this, observe that mapping nodes to other nodes preserves the cost of solutions only in the unweighted case. When invoked on the PB encoding of the CC problem (unweighted or weighted) explicit SCL, using the node-symmetries and a single minimal at-least-one core of size $k+1$, can generate $\binom{n}{k+1} - 1$ other cores corresponding to all other $\binom{n}{k+1} - 1$ subsets of size $k+1$. As such, IHS with explicit SCL can terminate after extracting a single core from `Extract-Core`, but enumerates an exponential number of symmetric cores and adds them to the hitting set IP. In contrast, given one core of size $k+1$, and the node-symmetries, compact SCL can generate a single cardinality constraint that limits the maximum number of nodes in any clique to k, resulting in termination of IHS in the next iteration with only a single constraint added to the hitting set IP.

Regarding lex-leader-based symmetry breaking on unweighted CC instances, if the node-symmetries are broken completely (which is not guaranteed to be the case, but is a best-case scenario that LL heuristically aims to achieve), the additional constraints added to the PBO instance essentially enforce that the maximum clique consists of k specific nodes instead of

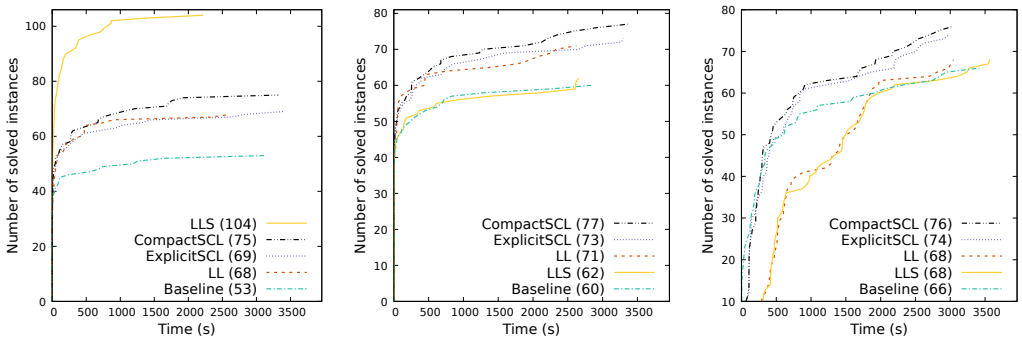


Figure 2 Impact of symmetry techniques on number of instances solved. Left: unweighted CC, Middle: weighted CC, Right: Haplotype inference.

any k nodes. Then the instance extended with lex-leader constraints has $n - k$ unit cores that directly enforce that the other nodes can not be in the maximum clique. When invoked on the extended instance, IHS can terminate after extracting $n - k$ of these (short) cores. (On weighted CC instances, the node-symmetries can be broken quite similarly when weak symmetry detection and dominance constraints are used: then the constraints enforce that the maximum clique consists of the k nodes with largest weights.) Further, in contrast to SCL, the fact that lex-leader symmetry breaking adds constraints to core-extractor can decrease core-extraction times. Thus we expect symmetry breaking to be very effective in these instances. The main question is how close IHS with (explicit/compact) SCL comes to the performance of IHS with lex-leader symmetry breaking.

To experiment with CC, we generated instances with all combinations of $n = 5..29$ and $k = 2..n$, yielding a total of 400 unweighted and 400 weighted benchmarks. On unweighted CC (see Figure 2 left), explicit SCL solves 16 more instances than the Baseline (IHS without symmetry techniques) with 69 vs 53 solved benchmarks. Compact SCL is even more effective, solving 6 more instances than explicit SCL (75 vs 69 instances). Using lex-leader with the full set of weak symmetries (LL) leads to 68 instances solved, which is less than either variant of SCL. Interestingly, restricting lex-leader-based symmetry breaking to the set of strong symmetries (LLS in the figure) leads to the overall best performance on unweighted CC and 104 instances solved. The results suggest that restricting the types of symmetries that BreakID can find (in this case to symmetries that map objective variables to objective variables), allows for better recognition of the relevant structure of the instance.

On the weighted CC instances (Figure 2 middle) the baseline solves 60 instances and lex-leader symmetry breaking restricted to strong symmetries only marginally improves with 62. When allowed to use all weak symmetries, IHS with lex-leader symmetry breaking improves to 71 solved instances. Our SCL approaches obtain the best performance and notable improvements over lex-leader-based breaking; explicit SCL solves 73 instances and compact SCL an impressive 77. A potential explanation for the big difference in performance of LLS between the unweighted and the weighted case is that node-symmetries are not strong in the weighted variant (whereas they are in the unweighted).

As a second, more practically oriented highly symmetric problem domain, we consider haplotype inference pedigrees problem [22], and in particular the task of finding a set of haplotypes that best explains given set of genotypes. For experimenting with this problem domain, we used the 100 instances submitted to Pseudo-Boolean competition 2011. It should be noted that, as explained in [22], these instances already include domain-specific symmetry

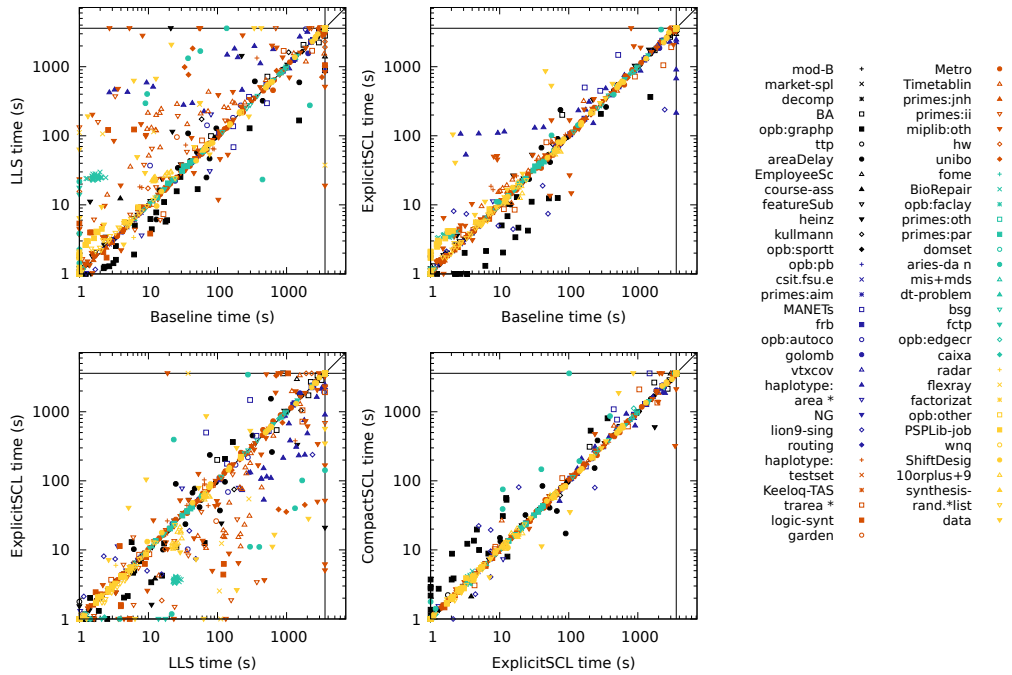


Figure 3 Runtime comparisons. Top left: Baseline vs. LLS, top right: Baseline vs. ExplicitSCL, bottom left: LLS vs. ExplicitSCL, bottom right: ExplicitSCL vs. CompactSCL.

breaking constraints added by hand for increased performance, enforcing a predefined order for the two haplotypes that produce a genotype. Hence an interesting question is whether the different symmetry techniques can improve IHS performance further on these instances. Again, (compact) SCL provides the greatest performance improvements (see Figure 2 right): Baseline without symmetry breaking solves 66 instances, which is improved by lex-leader symmetry breaking to 68 instances by both variants, LL and LLS – however, with explicit SCL the number of solved instances goes up to 74, and with compact SCL further to 76.

4.2 Experiments on a Wider Set of Benchmarks

We also investigate the relative impact of lex-leader symmetry breaking and SCL more generally on a wide set of PBO benchmarks including instances from over 60 domains as used in the the original papers presenting PBO-IHS [37, 38]. Our main interest here is whether the different types of symmetry techniques are viable to use by default even when there are no guarantees that the input instances would be highly symmetric. The benchmark set consists of 1786 instances across tens of different benchmark domains; see [36] for more details. We disregard LL here since LLS turned out to perform better than LL (see Appendix C).

A pairwise per-instance runtime comparison of Baseline, LLS and ExplicitSCL is shown in Figure 3. The total number of instances solved is: Baseline 948, LLS 947, ExplicitSCL 954, and CompactSCL 947. LLS results in significant runtime degradation compared to Baseline (Figure 3 top left). Indeed, the impact lex-leader symmetry breaking has on PBO-IHS appears very different to results presented in [40] on the runtime impact of lex-leader symmetry breaking on other PBO solving approaches, where (mild) runtime improvements were reported. ExplicitSCL consistently and most often significantly outperforms LLS (Figure 3 bottom left). A reason for this is that the PB solver used for core extraction tends to use significantly more

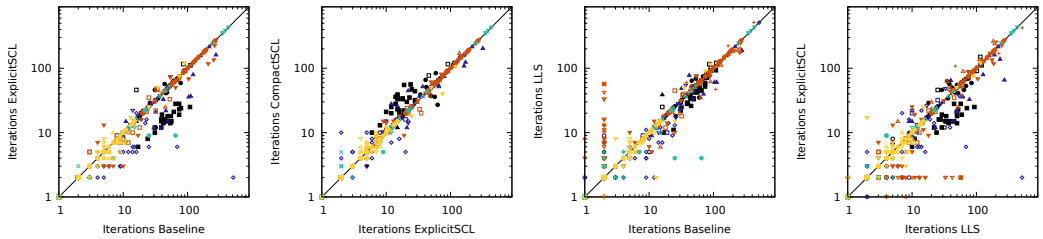


Figure 4 Number of IHS iterations. Left: Baseline vs. ExplicitSCL, Middle left: ExplicitSCL vs. CompactSCL, Middle right: Baseline vs. LLS, Right: LLS vs. ExplicitSCL.

time when invoked on a satisfiable instance. This follows the intuition that, in contrast to the dynamically applied SCL, the lex-leader constraints all added in the beginning of IHS search rule out symmetric solutions and hence may make finding solutions more difficult; more details on the runtime differences are provided in Appendix D. Similarly, ExplicitSCL on most instances significantly outperforms LLS (Figure 3 bottom left). ExplicitSCL also tends to have a performance-improvement impact on PBO-IHS, as observed in Figure 3 (top right), despite the symmetry computation overhead (up to 100 s). In contrast, despite CompactSCL performing very well on selected highly symmetric domains (recall Section 4.1), on this wider heterogenous set of benchmarks, CompactSCL performs slightly worse compared to ExplicitSCL (Figure 3 bottom right).

ExplicitSCL tends to terminate with fewer IHS main loop iterations than Baseline (Figure 4 left), suggesting that symmetric cores added by SCL are indeed helpful in decreasing required number of core extraction steps. Between ExplicitSCL and CompactSCL, there appear to be only more minor differences both ways (Figure 4 middle left). The relative number of iterations between LLS and ExplicitSCL (Figure 4 middle right) and LLS and Baseline (Figure 4 top left) do not fully explain the runtime degradation caused by LLS (recall Figure 3 right), which suggests that lex-leader constraints indeed significantly slow down the IHS iterations on average.

5 Conclusions

We presented symmetric core learning, SCL, as a dynamic approach to symmetry-aware IHS search. The approach works by generating symmetric unsatisfiable cores on-the-fly, thereby strengthen the hitting set IP. SCL has the potential for quickly improving bounds and, especially, avoiding unnecessary costly calls to the core extractor. Beyond explicitly enumerating symmetric cores, we proposed a compact way of representing a set of symmetric cores, with potential for compacting the hitting set IP. Empirically, SCL can improve the runtime of IHS for PBO especially on highly symmetric problem domains and, in contrast to lex-leader symmetry breaking, is not generally detrimental to IHS performance. For future work, SCL mainly assumes that the hitting set problems in IHS are computed using an IP solver; hence SCL could be applied in IHS instantiations for other constraint languages as well. In fact, a technique similar to (explicit) SCL has been added recently to IHS-based generation of MUSes [9] based on generating *symmetric satisfiable subsets* instead of *symmetric cores*. Another direction for future work is to investigate *proof logging* for IHS with SCL. For *symmetry breaking*, proof logging techniques have recently been added to a proof system for reasoning with PB constraints [10]. A proof system for *symmetric learning* in a SAT context has also been proposed [39]. An open question is whether these two ideas could be combined to yield proofs for SCL for pseudo-Boolean optimization.

References

1 Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. ShatterPB: symmetry-breaking for pseudo-Boolean formulas. In Masaharu Imai, editor, *Proceedings of the 2004 Conference on Asia South Pacific Design Automation: Electronic Design and Solution Fair 2004, Yokohama, Japan, January 27-30, 2004*, pages 883–886. IEEE Computer Society, 2004. doi:10.1109/ASPDAC.2004.179.

2 Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for Boolean satisfiability. *IEEE Trans. Computers*, 55(5):549–558, 2006. doi:10.1109/TC.2006.75.

3 Markus Anders, Sofia Brenner, and Gaurav Rattan. satsuma: Structure-based symmetry breaking in SAT. 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 4:1–4:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.SAT.2024.4.

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

5 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.

6 Belaid Benhamou, Tarek Nabhani, Richard Ostrowski, and Mohamed Réda Saïdi. Enhancing clause learning by symmetry in SAT solvers. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, pages 329–335. IEEE Computer Society, 2010. doi:10.1109/ICTAI.2010.55.

7 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.

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 Ignace Bleukx, Hélène Verhaeghe, Bart Bogaerts, and Tias Guns. Exploiting symmetries in MUS computation. 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 11122–11130. AAAI Press, 2025. doi:10.1609/AAAI.V39I11.33209.

10 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.

11 Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. A modular approach to MaxSAT modulo theories. 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 150–165. Springer, 2013. doi:10.1007/978-3-642-39071-5_12.

12 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.

13 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.

14 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.

15 Toby O. Davies, Graeme Gange, and Peter J. Stuckey. Automatic logic-based benders decomposition with minizinc. In Satinder Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 787–793. AAAI Press, 2017. doi:10.1609/AAAI.V31I1.10654.

16 Erin Delisle and Fahiem Bacchus. Solving weighted CSPs by successive relaxations. 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 273–281. Springer, 2013. doi:10.1007/978-3-642-40627-0_23.

17 Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. BreakIDGlucose: On the importance of row symmetry in SAT. In *CSPSAT, Vienna, 18 July 2014*, pages 1–17, July 2014. URL: https://lirias.kuleuven.be/handle/123456789/456639.

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

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

20 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.

21 Katalin Fazekas, Fahiem Bacchus, and Armin Biere. Implicit hitting set algorithms for maximum satisfiability modulo theories. 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 134–151. Springer, 2018. doi:10.1007/978-3-319-94205-6_10.

22 Ana Graça, Inês Lynce, João Marques-Silva, and Arlindo L. Oliveira. Efficient and accurate haplotype inference by combining parsimony and pedigree information. In Katsuhisa Horimoto, Masahiko Nakatsui, and Nikolaj Popov, editors, *Algebraic and Numeric Biology - 4th International Conference, ANB 2010, Hagenberg, Austria, July 31- August 2, 2010, Revised Selected Papers*, volume 6479 of *Lecture Notes in Computer Science*, pages 38–56. Springer, 2010. doi:10.1007/978-3-642-28067-2_3.

23 Alexey Ignatiev, Mikolás Janota, and João Marques-Silva. Quantified maximum satisfiability. *Constraints An Int. J.*, 21(2):277–302, 2016. doi:10.1007/S10601-015-9195-9.

24 Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation and enumeration. *ACM Trans. Comput. Log.*, 25(3):1–30, 2024. doi:10.1145/3670405.

25 Javier Larrosa, Conrado Martínez, and Emma Rollon. Theoretical and empirical analysis of cost-function merging for implicit hitting set WCSP solving. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan, editors, *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 8057–8064. AAAI Press, 2024. doi:10.1609/AAAI.V38I8.28644.

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

27 Erick Moreno-Centeno and Richard M. Karp. The implicit hitting set approach to solve combinatorial optimization problems with an application to multigenome alignment. *Oper. Res.*, 61(2):453–468, 2013. doi:10.1287/OPRE.1120.1139.

28 Andreas Niskanen, Jere Mustonen, Jeremias Berg, and Matti Järvisalo. Computing smallest MUSes of quantified Boolean formulas. 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 301–314. Springer, 2022. doi:10.1007/978-3-031-15707-3_23.

29 Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symb. Log.*, 62(3):981–998, 1997. doi:10.2307/2275583.

30 James A. Reggia, Dana S. Nau, and Pearl Y. Wang. Diagnostic expert systems based on a set covering model. *Int. J. Man Mach. Stud.*, 19(5):437–460, 1983. doi:10.1016/S0020-7373(83)80065-0.

31 Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987. doi:10.1016/0004-3702(87)90062-2.

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

33 Paul Saikko, Jeremias Berg, and Matti Järvisalo. LMHS: A SAT-IP hybrid MaxSAT solver. 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 539–546. Springer, 2016. doi:10.1007/978-3-319-40970-2_34.

34 Paul Saikko, Carmine Dodaro, Mario Alviano, and Matti Järvisalo. A hybrid approach to optimization in answer set programming. In Michael Thielscher, Francesca Toni, and Frank Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018*, pages 32–41. AAAI Press, 2018. URL: https://aaai.org/ocs/index.php/KR/KR18/paper/view/18021.

35 Paul Saikko, Johannes Peter Wallner, and Matti Järvisalo. Implicit hitting set algorithms for reasoning beyond NP. In Chitta Baral, James P. Delgrande, and Frank Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*, pages 104–113. AAAI Press, 2016. URL: http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12812.

36 Pavel Smirnov. Pseudo-boolean optimization by implicit hitting sets. Master's thesis, University of Helsinki, 2021. URL: http://hdl.handle.net/10138/340857.

37 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.

38 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.

39 Rodrigue Konan Tchinda and Clémentin Tayou Djamégni. On certifying the UNSAT result of dynamic symmetry-handling-based SAT solvers. *Constraints An Int. J.*, 25(3-4):251–279, 2020. doi:10.1007/s10601-020-09313-2.

40 Daimy Van Caudenberg and Bart Bogaerts. Symmetry and dominance breaking for pseudo-Boolean optimization. In Toon Calders, Celine Vens, Jefrey Lijffijt, and Bart Goethals, editors, *Artificial Intelligence and Machine Learning - 34th Joint Benelux Conference, BNAIC/Benelearn 2022, Mechelen, Belgium, November 7-9, 2022, Revised Selected Papers*, volume 1805 of *Communications in Computer and Information Science*, pages 149–166. Springer, 2022. doi:10.1007/978-3-031-39144-6_10.

41 Daimy Van Caudenberg, Bart Bogaerts, and Leandro Vendramin. Incremental SAT-based enumeration of solutions to the yang-baxter equation. 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 3–22. Springer, 2025. doi:10.1007/978-3-031-90653-4_1.

A Implementing CompactSCL

Algorithm 3 details our implementation of compact SCL combined with explicit SCL (the instantiation of `SCL` to be called by Algorithm 1). Given a core C extracted by `Extract-Core`, the algorithm initializes the set S of cores to be returned, a set $\mathcal{D}$ of definitions of new count variables, and a queue *coreQueue* of cores to be processed. Each pair in *coreQueue* consists of a core constraint C, and the definitions D of any count variables in C. The cores are then processed in breadth-first order.

When processing a core C, compact SCL is tried first by the `Try-Compact-SCL` method attempting to compute a compact representation of a set of cores symmetric to C (Lines 6–15). The procedure returns a set $\mathcal{C}$ of pairs (C, D), where C is a core constraint, and a set D contains the definitions of count variables that appear in C. Each pair is a compacted core that represents a set of cores that are symmetric to C. When compact SCL on C is successful ($|\mathcal{C}| > 0$), all pairs in $\mathcal{C}$ are added to S, the new definitions to $\mathcal{D}$ and the original C removed (as the constraints in $\mathcal{C}$ represent it). Finally, each pair (C, D) in $\mathcal{C}$ is added to *coreQueue* and the next core popped on Line 5. If instead compact SCL on C is unsuccessful (i.e., if $|\mathcal{C}| = 0$ so the continue statement on Line 15 is not reached), or if the budget for compact SCL is exceeded (at which point the if statement on Line 6 returns false), explicit SCL is instead applied on C (Lines 16–21). Explicit SCL operates as described in Section 3.2 with the exception that now – since we are applying explicit SCL to compacted cores – only a subset of Σ is used. In Line 17 method `Filter-Symmetries`(Σ, D) is invoked to select the symmetries in Σ that keep the definitions of count variables in C intact, i.e., symmetries that can be used when explicit SCL is applied in conjunction with compact SCL.

Algorithm 3 Compact SCL combined with explicit SCL.

```
1  Compact-SCL(C, 𝒦, Σ)
       Input: A core C, a set of all so-far extracted cores 𝒦, and a set of symmetries Σ.
       Output: A set of constraints, S ∪ 𝒟, that compactly encode a set of symmetric
               cores of C.
2      S ← {C};   𝒟 ← ∅;   coreQueue.init({(C, ∅)});
3      constrBudgetCompact ← constrLim;   litBudgetCompact ← litLim;
        constrBudgetExplicit ← constrLim;   litBudgetExplicit ← litLim;
4      while coreQueue.size()>0 do
5          (C, D) ← coreQueue.pop();
6          if constrBudgetCompact > 0 and litBudgetCompact > 0 then
7              𝒞 ← Try-Compact-SCL(C, D, Σ);
8              if |𝒞| > 0 then
9                  S ← S \ {C};
10                 for (C′, D′) ∈ 𝒞 do
11                     𝒟 ← 𝒟 ∪ D′;
12                     S ← S ∪ {C′};
13                     coreQueue.push((C′, D′));
14                     constrBudgetCompact ← constrBudgetCompact − 1;
                        litBudgetCompact ← litBudgetCompact − |LIT(C′)|;
15                 continue;
16         if constrBudgetExplicit > 0 and litBudgetExplicit > 0 then
17             for σ ∈ Filter-Symmetries(Σ, D) do
18                 if σ(C) ∉ 𝒦 ∪ S then
19                     S ← S ∪ {σ(C)};
20                     coreQueue.push((σ(C), D));
21                     constrBudgetExplicit ← constrBudgetExplicit − 1;
                        litBudgetExplicit ← litBudgetExplicit − |LIT(C)|;
22     return S ∪ 𝒟;
```

For intuition of how our approach produces compacted cores in terms of symmetry decompositions, Algorithm 3 can be seen as computing a compact representation of cores symmetric to C by building a partitioning $L_1 + \cdots + L_k \geq A$ of C step-by-step. The first time `Try-Compact-SCL` is invoked on C, different ways of selecting L_1 are tried. For each candidate L_1 resulting in an orbit of size at least 2, a new compacted core with the chosen L_1 replaced by a new count variable is introduced. The compacted cores are then reconsidered in subsequent iterations, in which `Try-Compact-SCL` tries ways of selecting L_2, and so on.

Algorithm 4 details `Try-Compact-SCL`. Given a (potentially compacted) core C, definitions D of the count variables in C, and a set Σ of symmetries, the algorithm first initializes an empty set $\mathcal{C}$ of new compacted cores on Line 2. The loop on Lines 3–17 then looks for a new subset of terms L_i of the symmetry decomposition that is being built for C, prioritizing small $|\text{LIT}(L_i)|$ (i.e., small values of k). For each k, the method `Select-Candidate-Partitions` returns a set $\mathcal{S}$ of "candidate L_is". As computing all subsets of C with size k is infeasible, for $k > 1$, we select the candidate L_i based on the symmetries in Σ. More precisely, $\mathcal{S}$ contains only sets L for which a symmetry in Σ (or row in a matrix-symmetry) shares with the core exactly the literals in L. After computing $\mathcal{S}$, Algorithm 4 first heuristically estimates if applying compact SCL with the sets in $\mathcal{S}$ would result in compacted cores where the number

Algorithm 4 Applying compact symmetric core learning on a single core.

```
1  Try-Compact-SCL(C, D, Σ)
       Input: A core C, definitions of count variables D, and a set of symmetries Σ.
       Output: A set of core constraints 𝒞 and definition constraints 𝒟, that compactly
               encode symmetric cores implied by C.
2      𝒞 ← ∅;
3      for k = 1, ..., |LIT(C)| − 1 do
4          𝒮 ← Select-Candidate-Partitions(C, k);
5          if Check-If-Good(C, S, k) = false then continue;
6          for L ∈ 𝒮 do
7              if |LIT(L)| = |{ℓ}| = 1 then
8                  ℒ_in, ℒ_out ← Find-Interchangeable-Lits(C, D, ℓ, Σ);
9                  if |ℒ_in| > 1 and |ℒ_out| > 0 then
10                     C′, D′ ← Add-Count-Variables(C, ℒ_in, ℒ_out);
11                     𝒞 ← 𝒞 ∪ {(C′, D ∪ D′)};
12                     continue;
13             𝒪 ← Compute-Orbit(C, D, L, Σ);
14             if |𝒪| < 2 then continue;
15             C′, D′ ← Add-Count-Variable(C, L, 𝒪);
16             𝒞 ← 𝒞 ∪ {(C′, D ∪ D′)};
17         if |𝒞| > 0 then return 𝒞;
18     return 𝒞;
```

of auxiliary constraints is smaller than the number of cores covered. (The check is here rather than elsewhere for practical implementation reasons.) If not, the sets $L \in \mathcal{S}$ are not used for compact SCL. Otherwise the algorithm checks for each $L \in \mathcal{S}$ if the L can be used as the next L_i. The special treatment of candidate Ls of size 1 corresponds to Case 1 of Definition 13. When $\text{LIT}(L) = \{\ell\}$ the call to `Find-Interchangeable-Lits`(C, D, ℓ, Σ) first attempts to find a set of literals $\mathcal{L}$ and a set of symmetries that allow permuting the set $\mathcal{L}$ in any order, while keeping rest of C unchanged. The call returns the set $\mathcal{L}$ of interchangeable literals divided into two sets: $\mathcal{L}_{in}$ containing the literals of $\mathcal{L}$ that appear in core C, and $\mathcal{L}_{out}$ containing the rest. If $|\mathcal{L}_{in}| > 1$ and $|\mathcal{L}_{out}| > 0$ i.e. at least two of the interchangeable literals are in C, and at least one is not, the method `Add-Count-Variables` is invoked to generate a new compacted core C', in which the literals in $\mathcal{L}_{in}$ are replaced by count variables defined by constraints in D' (Line 10). Then $\mathcal{C}$ is extended to include C' and $\mathcal{D}$ to include D (Line 11). When $|L| > 1$ or Case 1 optimization does not result in a compacted core (i.e. the if statement on Line 9 returns false), `Compute-Orbit` instead finds an orbit $\mathcal{O}$ for L under a subset of symmetries in Σ that satisfies the requirements for L being part of a symmetry decomposition. If $\mathcal{O}$ includes something in addition to L, `Add-Count-Variable` next generates an compacted core C' in which L is replaced by a count variable defined by the constraints in D (Line 15). C' is then added to $\mathcal{C}$ and D to $\mathcal{D}$ (Line 16). Algorithm 4 continues iterating over increasing sizes of k until a compact core can be generated.

Algorithm 5 details `Find-Interchangeable-Lits`, for computing interchangeable literals for a literal ℓ in a core C. Initially, the candidate set $\mathcal{L}$ only contains ℓ (Line 2). The first stage of search (Lines 3–4) heuristically add literals one by one to $\mathcal{L}$. The second stage (Lines 5–8) checks if the resulting $\mathcal{L}$ is a set of interchangeable literals. If not, it tries to identify a subset of $\mathcal{L}$ that is. The first stage uses two methods to introduce literals into

Algorithm 5 Finding a set of interchangeable literals.

```
1 Find-Interchangeable-Lits(C, D, ℓ, Σ)
    Input: A core C, the definition constraints D of the count variables in C, a
           literal ℓ that appears in C, and a set of symmetries Σ.
    Output: Sets L_in and L_out, where {ℓ} ⊆ L_in ⊆ LIT(C), L_in ∩ LIT(C) = ∅ and
            L_in ∪ L_out is a set of "interchangeable" literals.
2   L ← {ℓ};
3   while Extend-Outside-Core(L, C, D, Σ) or Extend-In-Core(L, C, D, Σ) do
4     continue;
5   while 2 ≤ |O ∩ C| ≤ |O| − 1 do
6     L' ← Check-Interch(L, C, D, Σ);
7     if L' = L then return L ∩ LIT(C), L \ LIT(C) ;
8     else L ← L' ;
9   return {ℓ}, ∅;
```

$\mathcal{L}$. `Extend-Outside-Core`$(\mathcal{L}, C, D, \Sigma)$ looks for symmetries that map a literal $l \in \mathcal{L}$ to some literal $l' \notin \mathcal{L}$, while keeping the rest of $\mathcal{L}$ and C stable. `Extend-In-Core`$(\mathcal{L}, C, D, \Sigma)$ checks for each pair $l \in \mathcal{L}$ and $l' \in (\text{LIT}(C) \setminus \mathcal{L})$, if there is a symmetry that maps l' and l to each other while keeping the rest of C, definitions in D, and the rest of $\mathcal{L}$ stable.

When the construction of the set $\mathcal{L}$ is finished, the algorithm proceeds to the second stage in which `Check-Interch`$(\mathcal{L}, C, D, \Sigma)$ checks if $\mathcal{L}$ contains a set of interchangeable literals. `Check-Interch` tries to rebuild $\mathcal{L}$ by adding new literals one by one, only using symmetries that do not map any $l \in \mathcal{L}$ to some $l' \notin \mathcal{L}$. The implementation ensures that there is a set of symmetries that allow permuting literals in $\mathcal{L}$ into any order. First, `Check-Interch`$(\mathcal{L}, C, D, \Sigma)$ initializes $\mathcal{L}' = \{l\}$ with some $l \in \mathcal{L}$. Then it looks for symmetries that swap some literals $l \in \mathcal{L}'$ and $l' \in (\mathcal{L} \setminus \mathcal{L}')$, while keeping $\mathcal{L}' \setminus \{l\}$, $(\mathcal{L} \setminus \mathcal{L}') \setminus \{l'\}$, and the rest of C stable. If such l and l' are found, $\mathcal{L}'$ is extended to include l'. If in the end $\mathcal{L} = \mathcal{L}'$, `Check-Interch`$(\mathcal{L}, C, D, \Sigma)$ finishes by returning the set $\mathcal{L}$. If $\mathcal{L} \neq \mathcal{L}'$, the process is reiterated by initializing $\mathcal{L}' = \{l\}$ with a different $l \in \mathcal{L}$. If there is no more new $l \in \mathcal{L}$ with which $\mathcal{L}'$ could be initialized, `Check-Interch` returns the $\mathcal{L}'$ computed on the last l, which will always be the first ℓ given to `Find-Interchangeable-Lits` to ensure that the literal ℓ used when calling `Find-Interchangeable-Lits` will be included in the set $\mathcal{L}$ returned. If `Check-Interch`$(\mathcal{L}, C, D, \Sigma)$ finds $\mathcal{L}'$ such that $\mathcal{L} = \mathcal{L}'$, the set $\mathcal{L}$ is divided to literals in the core and outside the core, and these two sets are then returned (Line 7). Otherwise the procedure is reiterated with the set $\mathcal{L}'$ returned by `Check-Interch` (Line 8).

Finally, Algorithm 6 details the computation of an orbit. The procedure resembles explicit SCL as detailed in Algorithm 2 in that, starting from L itself, it explicitly enumerates all members of the orbit in breadth-first order. To ensure, that the symmetries used satisfy the definition of a symmetry decomposition, the set of symmetries to be considered during breadth-first search is filtered in Line 4. In particular, in our implementation, a symmetry $\sigma \in \Sigma$ is removed from consideration if any of the following conditions hold for it. 1) σ maps some literal $\ell \in C$ to a literal not in C, or to a literal in C that has a different coefficient in C. 2) C contains a Boolean count variable s_x introduced in previous iterations of compact learning and σ maps a literal that appears in the definition of s_x to a literal that does not. 3) C contains an integer count variable s_x and σ maps any literal in the definition of σ to any literal other than itself.

Algorithm 6 Computing an orbit for L.

```
1  Compute-Orbit(C, D, L, Σ)
       Input: A core C, the definition constraints D of the count variables in C, a
              subset L of the terms of C, and a set of symmetries Σ.
       Output: An orbit 𝒪 of L computed on a subset of Σ that satisfies the definition
               of symmetry decomposition.
2      𝒪 ← {L};
3      orbitQueue.init(L);
4      Σ' ← Filter-Symmetries(Σ, C, D, L);
5      while orbitQueue.size() > 0 do
6          L ← orbitQueue.pop();
7          for σ ∈ Σ' do
8              if σ(L) ∈ 𝒪 then continue;
9              𝒪 ← 𝒪 ∪ {σ(L)};
10             orbitQueue.push(σ(L));
11     return 𝒪;
```

B Details on the Encoding of the CC instances

Extending the encoding from [29], the PBO encoding of the CC instances has the following variables.

- $e_{i,j}$ for $1 \leq i, j \leq n$ indicates an edge between nodes i and j
- $r_{i,c}$ for each node $1 \leq i \leq n$ and color $1 \leq c \leq t$ indicates that the node i is colored with thecolor c.
- $q_{k,i}$ for $1 \leq i, k \leq n$ indicates that the node i is the kth node to be included in a largest clique.
- b_i for $1 \leq i \leq n$ are objective variables indicating that node i is not in a largest clique.

The objective is to minimize $\sum_{i=1}^{n} b_i$ in the unweighted CC and $\sum_{i=1}^{n} ib_i$ in the weighted CC, subject to

- $\sum_k q_{k,i} \leq 1$ for $1 \leq i \leq n$ (Each node i is the kth node in the clique for at most one value of k.)
- $e_{i,j} - q_{k,i} - q_{k',j} \geq -1$ for $1 \leq i, j, k, k' \leq n$
  (If nodes i and j are the kth and k'th nodes in a largest clique, there must be an edge between i and j.)
- $e_{i,j} + r_{i,c} + r_{j,c} \leq 2$ for $1 \leq i, j \leq n$, $1 \leq c \leq t$
  (Adjacent nodes must have different colors.)
- $b_i + \sum_k q_{k,i} \geq 1$ (If node i is in the clique, it is the k:th node for some k..)

The scripts used to generate the CC instances are provided in the paper supplement.

C Comparing LL and LLS

Runtime comparison of LL and LLS is shown in Figure 5, LLS exhibiting better performance.

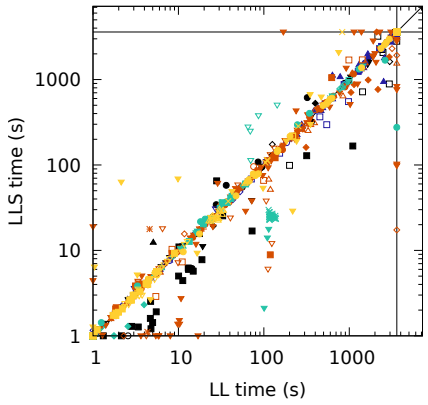


Figure 5 Runtime Comparison: LL vs. LLS.

D Further Analysis of LLS vs ExplicitSCL

Fig. 6 shows a comparison of runtime LLS and ExplicitSCL spent in BreakID, core extraction, the IP solver, and solution-improving search (SIS). ExplicitSCL uses less time in each of the components. However, since ExplicitSCL generally solves instances with a smaller number of iterations (recall Figure 4), the difference in total runtime spent in core extraction, the IP solver and SIS is partially explained by the difference in the number of IHS iterations required for termination. For further details, Fig. 7 shows a comparison of the average time spent per iteration in each of the core extraction, IP solver and SIS component: the main difference in runtimes is due to time spent in core extraction and SIS. Digging deeper into the runtime spent in core extraction and SIS, we observed that for core extraction, the main runtime difference is due to calls which turn out to be satisfiable. This follows the intuition that the lex-leader constraints – all added at the beginning of IHS search – which rule out symmetric solutions may make it harder for a PB solver to find a solution. For SIS the main runtime difference is due to calls on which the PB solver is terminated without solution due exceeding the runtime limit enforced in PBO-IHS on the SIS solver calls.

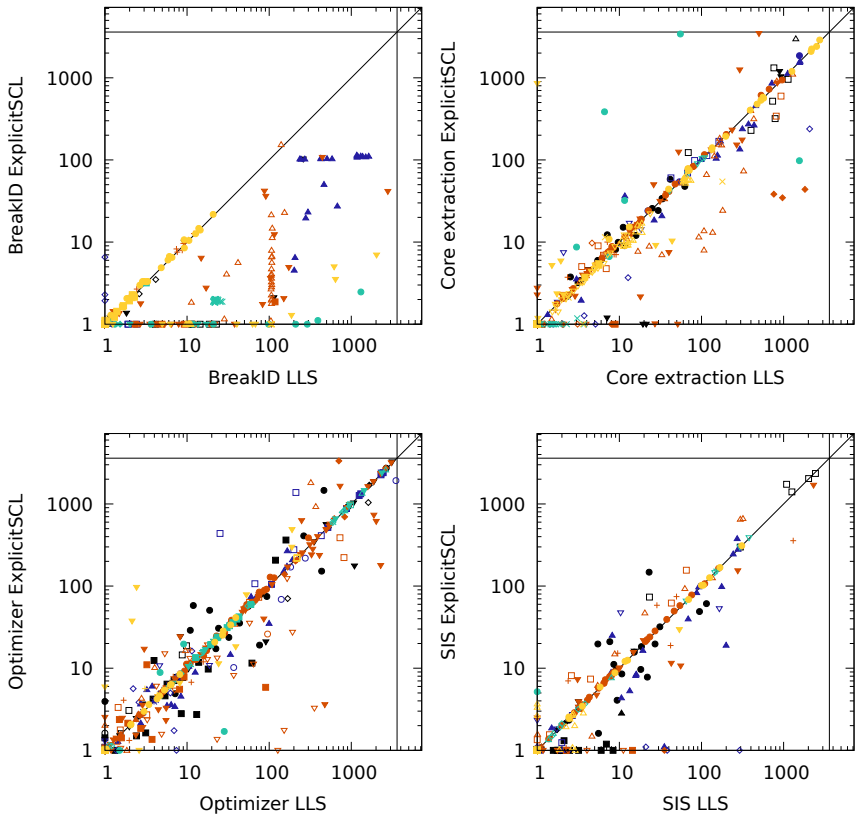


Figure 6 Time spent in BreakID (top left), core extraction (top right), IP solver (bottom left) and SIS (bottom right).

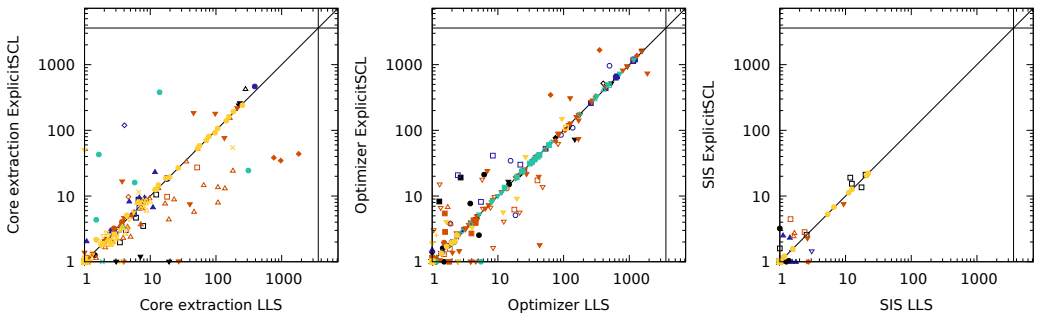


Figure 7 Average time spent per iteration in core extraction (left), IP solver (middle) and SIS (right).

PrintTalk: A Language for Constraint-Based 3D Modelling

Jef Jacobs
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Wolfgang De Meuter
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Jens Nicolay
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Abstract

Programmatic CAD (PCAD) is an emerging alternative to traditional visual CAD software. However, state-of-the-art PCAD tools have limited or no support for constraints. Consequently, these tools depend solely on parametrisation for variability, reusability, and composition of shapes. This leads to problems such as parameter explosion, leaky compositional abstraction, and prevents a declarative approach to defining spatial patterns (linear, grid, circular, etc.) for the constituents of a composition. This paper describes the design of PrintTalk, a PCAD language that supports 3D modelling by composing shapes and expressing relations between them using first-class constraints. Evaluating PrintTalk against state-of-the-art PCAD tools demonstrates that its expressive abstraction and composition mechanisms facilitate the design and promotes the reuse of shapes.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Software and its engineering → Multiparadigm languages; Software and its engineering → Constraints

Keywords and phrases Programmatic 3D Modelling, PCAD, Domain specific language, Constraints

Digital Object Identifier 10.4230/LIPIcs.CP.2025.16

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.15593661`

1 Introduction

Computer-aided design (CAD) systems are central to the modern product design process, in which physical products are represented as digital CAD drawings. Designing digital 3D models often involves sophisticated CAD software, of which the vast majority is *visual* in nature. Visual CAD tools typically have built-in constraint mechanisms to aid designers in taming complexity. For example, *geometric constraints* [21] (parallelism, concentricity, etc.) control relationships between components, while *dimensional constraints* (length, radius, etc.) help specify precise dimensions of shapes in relation to each other. Designing parametrised components of which the parameters can subsequently be modified to adjust the entire design accordingly is called *parametric modelling*.

While visual CAD approaches can be accessible, intuitive, and well-suited for rapid prototyping, they lack several key advantages of programmatic approaches. Programmatic approaches are more amenable to software engineering techniques, enabling finer-grained control over the design process, facilitating the reuse of complex shapes, and offering richer automation, version control, and collaboration features. Consequently, *programmatic CAD* (PCAD) has emerged as a compelling alternative, where 3D models are constructed programmatically using parametric modelling. In addition to the inherent benefits of a code-based approach, PCAD leverages the full power of a programming language (variables, functions, loops, conditionals, modules, etc.), providing rich and expressive abstraction and composition mechanisms. This allows designers to think about 3D modelling at a higher

© Jef Jacobs, Wolfgang De Meuter, and Jens Nicolay;
licensed under Creative Commons License CC-BY 4.0

level, using custom abstractions tailored to their specific needs. For this reason, over the past few years there has been a growing interest in PCAD [10, 17], leading to the development of several PCAD libraries [5, 19, 14] and languages [13, 18].

1.1 Problem

Despite the benefits of a programmatic approach, a significant shortcoming of the state-of-the-art PCAD tools we surveyed is that, unlike their visual counterparts, they have limited or no support for constraints. In the absence of constraints, PCAD tools must *exclusively* rely on parametrisation for variability, reusability, and composition of shapes. In all but the most trivial designs, this reliance on parameters leads to three significant problems.

Parameter explosion. To achieve variability and reusability, designers must expose numerous parameters. As the complexity of the design and the points of variation increase, the number of parameters increases dramatically. Additionally, the programmer is responsible for assigning valid values to all parameters. Any dependencies that may exist between parameters remain implicit, which makes assigning valid values even more difficult.

Leaky compositional abstraction. Composing a shape out of other shapes using only parameters requires exposing internal details as "composition parameters" of constituent shapes whose only purpose is to make composition possible. For instance, to align a hinge with a door, the screw hole positions (normally an internal detail of the hinge) must be exposed as parameters. This breaks encapsulation, as programmers must track intricate details about composition hierarchies, hindering high-level reasoning and modular design. The need for composition parameters also contributes to parameter explosion.

Imperative compositional specification. Parameters alone cannot declaratively express the complex and reusable relations between components. Therefore, creating patterns (linear, grid, circular, etc.) without constraints requires imperative code using sequences and loops, coupling the creation of components with their arrangement. This results in inflexible, non-extensible compositions that cannot be properly abstracted over.

1.2 Solution

This paper introduces PrintTalk, a PCAD language that addresses the limitations of existing tools by tightly integrating constraint solving with parametric modelling. PrintTalk features three main concepts: parametrised shapes, constraints, and patterns (which are specialized constraints for spatial relationships). 3D modelling in PrintTalk involves programmatically composing shapes and expressing relations between them. Each shape definition declares a set of parameters that can partake in constraints and through which the shape can be modified. While shapes and constraints make up the core of the language, PrintTalk also features expressions that act as abstraction mechanisms and syntactic sugar for defining, instantiating, and composing shapes and constraints in a scalable and expressive manner.

PrintTalk overcomes the shortcomings of existing PCAD tools that we identified by meeting the following three criteria.

- **Criterium 1: Flexible variability and reusability.** PrintTalk shapes expose named parameters to which values can be assigned explicitly by the program, in which case they act as *constructor parameters*. Parameters that are not explicitly assigned a value become *constraint variables* whose values are determined by the constraint solver when running the program. Constraints and constraint variables provide a declarative and relational way to define shapes in PrintTalk. These declarative relationships, together with a reduced need for specifying explicit parameter values, avoid the problem of parameter explosion and provides a clean way to balance flexibility and complexity in a design.

- **Criterium 2: Flexible composition.** PrintTalk enables flexible composition by allowing constraints to be placed on the parameters of constituent shapes. A shape can constrain its own parameters and the parameters of its constituents. This avoids requiring detailed knowledge of a shape's internal structure before it can be composed with other shapes in a valid manner. PrintTalk features a set of primitive constraints (e.g., "equal", "greater than", etc.), but also allows users to define their own constraints as a combination of other constraints. For example, users can define their own "on-top-of" or "touching" constraints for composing their shapes.
- **Criterium 3: Declarative and reusable compositional abstraction.** PrintTalk decouples the logic for creating multiple shapes from arranging them in a particular spatial configuration. Multiple copies of shapes can be declaratively created by means of a `for` loop. In a separate step, multiple shapes can be arranged by adding constraints on their parameters. Because constraints are first-class citizens in PrintTalk, they can be composed and named, making them reusable. In PrintTalk, these reusable (compound) composition constraints expressing spatial arrangements are called *patterns*.

1.3 Overview

Section 2 introduces the PrintTalk language, detailing its core components and design principles. Section 3 surveys related work in programmatic CAD and constraint-based systems. Section 4 presents an evaluation of PrintTalk, comparing it to two state-of-the-art PCAD tools and demonstrating how it addresses the problems outlined in Section 1.1.

2 Overview of PrintTalk

This section introduces PrintTalk, a domain-specific language designed for constraint-based 3D modelling. PrintTalk enables the construction of complex 3D shapes by composing simpler shapes and expressing relations between them using constraints. To promote reusability, the language provides components for defining and instantiating new shapes and constraints.

The following subsections detail the fundamental components of the PrintTalk language. First, the concepts of shapes, parameters, instantiation, and the structure for defining composite shapes using scripts and constraints are introduced (Section 2.1). The core mechanism behind PrintTalk's flexible, solver-driven composition in which parameters function dually as constructor arguments or constraint variables, is explained (Section 2.1.1). This discussion covers implicit parameters used for relative positioning (Section 2.1.2) and the language's scoping rules (Section 2.1.3). Next, composite constraints (Section 2.2.1) and spatial patterns (Section 2.2.2) are presented as mechanisms for structuring and reusing constraints. Finally, the process for generating a 3D model as output is described (Section 2.3), along with the integration details of the Z3 constraint solver (Section 2.3.1) and the methodology for determining the constraint hierarchy (Section 2.3.2). A complete overview of PrintTalk's syntax is provided in Appendix A.

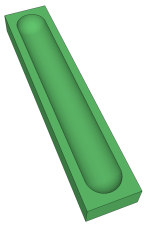
2.1 Shapes and Constraints

In PrintTalk, a shape has a name and a set of named, shape-specific parameters, such as dimensions or quantities. The language provides built-in primitive shapes, including cuboids, cylinders, cones, and spheres.

A shape is instantiated by referring to it by name and providing it zero or more values for its named parameters. For example, the expression `(sphere #:radius 10)` instantiates a sphere with radius `10`.

Listing 1 A PrintTalk program (left) and its resulting shape (right).

```
1  (shape: capsule (h d)
2   (script:
3    (named: cyl (cylinder #:diameter d #:height (- h d)))
4    (named: sp1 (sphere #:diameter d #:z (/ (- h d) -2)))
5    (named: sp2 (sphere #:diameter d #:z (/ (- h d) 2))))
6   (constraints: (assert: (>= h (+ 10 (* 2 d))))))
7
8  (shape: tray (l w)
9   (script:
10   (named: blk (cuboid #:length l #:width w #:height (/ w 2)))
11   (cut: (named: caps (capsule #:h (- l 10)
12    #:z (/ blk.height 4) #:rot-y 90))))
13  (constraints: (assert: (= caps.d (- blk.width 10)))))
14
15 (print: (tray #:l 150 #:w 30) "tray.stl")
```



New shapes are defined as a composition of other primitive or composite shapes using the `shape:` keyword. PrintTalk represents composite shapes as trees, where leaf nodes are primitive shapes.

```
1 (shape: name (<parameters>)
2  (script: <constituents>)
3  (constraints: <assertions>))
```

The `script:` section is used for instantiating constituent shapes and defining their basic composition. By default, shapes instantiated within a script are composed by means of a union, unless specified otherwise by a `(cut: <shape>)` operation, in which case the shape provided as argument to the `cut:` statement is cut from the preceding shapes in the script.

The order in which constituent shapes are instantiated within a script matters when a `cut:` operation is present in the script, as `cut:` operations are not commutative. The shape provided as argument to a `cut:` operation is only cut from previously instantiated shapes, leaving shapes instantiated after the `cut:` unaffected. A constituent shape can be given a name using a `(named: <name> <shape>)` statement, so that it can be referenced using that name. Listing 1 shows a PrintTalk program and its resulting shape, a pencil tray. Two composite shapes are defined: a capsule (lines 1–6) and a tray (lines 8–13). The pencil tray is obtained by cutting a capsule shape `cap` from a cuboid (or "block") shape `blk` (lines 9–12). A capsule itself is a composition of a cylinder with spheres at its top and bottom (lines 2–5).

The `constraints:` section in a `shape:` definition specifies the constraints used between the parameters of that shape. For example, the constraint on line 6 in Listing 1 sets a minimum height for the capsule shape based on its diameter to make sure the capsule has an "elongated" shape. Constraints are also used for defining the layout and precise relationships between constituent shapes (Section 2.1.1). An example of this is the equality constraint on line 13, which defines the relationship between the diameter of the capsule and the width of the block shape, ensuring that the diameter of the cutout capsule shape is smaller that the width of the block shape. PrintTalk provides **built-in primitive constraints** for comparing values (i.e., $=$, $<$, $<=$, $>$, $>=$).

Like shapes, constraints can also be composed as described in Section 2.2.1.

2.1.1 Constraint Variables and Composition

The expressive power of PrintTalk stems from the interaction between shape parameters and constraints. A shape parameter can function in two ways: as a *constructor parameter* if explicitly assigned a value during instantiation, or as a *constraint variable* if left unspecified. In the latter case, its value is determined by the constraint solver based on the relationships defined in the program (Section 2.3.1).

Consider the built-in `sphere` shape definition, which declares `radius` and `diameter` parameters linked by the constraint `(= diameter (* 2 radius))`. If a sphere is instantiated with only a diameter (as on lines 4 and 5 in Listing 1), the solver infers the radius, and vice versa. This bidirectional capability also allows for validation: if both parameters are provided with inconsistent values (e.g., `(sphere #:radius 10 #:diameter 23)`), PrintTalk signals a runtime error, as the constraint is violated.

Fundamentally, PrintTalk treats all parameters as constraint variables. This unified view enables flexible instantiation: programmers can provide explicit values for some parameters (which corresponds to adding an equality constraint), while allowing the solver to derive others based on declared constraints. This significantly promotes component reusability.

This constraint-based mechanism of deriving unspecified values extends naturally to shape composition. Relations between constituent shapes are expressed declaratively using constraints on their parameters, as demonstrated by the relation between the capsule diameter and block width in the tray example (Listing 1). As another example, to ensure two surfaces glue together correctly, constraints can enforce that their corresponding dimensions match, allowing the constituents to adapt to each other. PrintTalk's extensible constraint system facilitates defining such complex spatial relationships.

The bidirectional capabilities of constraint solvers make constraints a powerful tool for expressing relations between shapes. Its ability to both find suitable parameter values and check the consistency of provided ones enables versatile composition strategies that are difficult to achieve with purely manual parameter assignment. Without constraints, programmers must imperatively calculate and assign values for all parameters, often leading to less flexible and reusable designs.

However, not all parameter values can be derived from constraints, particularly when a parameter's value is required structurally *before* constraint solving commences, or when it defines the set of shapes upon which a constraint will operate. In such cases, PrintTalk requires these values to be provided explicitly during instantiation. It is therefore possible to *force* parameters to act strictly as constructor parameters by preceding their name with an exclamation mark in the parameter list. As illustrated later, the `!nr-teeth` parameter in the gear shape definition (Listing 2) must be explicit because it determines the number of teeth to instantiate within a loop. Similarly, the `!shapes` parameter in the circular spatial pattern (Listing 2) must be provided, as it specifies the collection of shapes the constraint acts upon. If an argument for such a forced constructor parameter is omitted during instantiation, PrintTalk raises a runtime error instead of treating it as a constraint variable whose value can be derived.

2.1.2 Relative Position and Rotation

In PrintTalk, every shape definition implicitly contains six parameters that determine its 3D position (`x`, `y`, `z`) and rotation (`rot-x`, `rot-y`, `rot-z`) *relative* to its parent. Root shapes, which have no parent, have an absolute position and rotation of `(0, 0, 0)`. Implicit parameters behave identically to explicit parameters, including how they are bound.

For each implicit parameter, an implicit soft constraint is asserted stating that the values of these constraint variables is `0` (i.e., equal to the parent's value). The underlying solver aims to find a solution that satisfies these soft constraints, but, as discussed in Section 2.3.1, no error is raised when no such solution exists. Default values for implicit parameters implements a flexible environmental acquisition mechanism [11] that enables expressing relative positions and rotations of constituent shapes only when required, while also combatting parameter explosion. This mechanism is illustrated by the capsule shape definition in Listing 1. A capsule shape is defined as a combination of a cylinder and two spheres. The z-component of the spheres is specified, while the other position and rotation parameters are not specified and thus acquired from the parent shape, meaning their offset is set to `0` through implicit soft constraints.

2.1.3 Scoping Mechanism

A shape's parameters are accessible throughout the entire body of its definition (the `script:` and `constraints:` sections). Parameters of named constituent shapes (created with `name:`) within a shape definition's script are visible in subsequent script statements and in the `constraints:` section. These parameters must be referenced using dot notation. However, shapes cannot directly access the parameters of their *grandchildren* (the constituents of their constituents). Composite shape definitions can introduce parameters that are bound to a selection of their constituents' parameters. This allows a grandparent shape to influence its grandchildren through intermediate parent parameters. Therefore, while PrintTalk's scoping rules promote information hiding, programmers retain control over which parts of a composite shape's constituents are exposed upon reuse.

To illustrate this scoping mechanism, consider the PrintTalk program in Listing 1 again. The composite shape definition `capsule` declares parameters `h` (height) and `d` (diameter) (line 1). These parameters are used in the script and constraints of the capsule shape definition. Parameter `d` is bound to the `diameter` parameters of the constituent cylinder and spheres in the capsule's script (lines 2–5). Next, a capsule is instantiated and named `caps` in the script of the `tray` shape (lines 11–12). The capsule's `d` parameter is referenced in the tray's constraint as `caps.d` (line 13). However, the `tray` shape cannot access parameters of the `capsule`'s constituents, such as `caps.sp1.diameter`.

2.2 Constraint Composition and Spatial Patterns

2.2.1 Composite Constraints

Not only shapes, but also constraints can be composed to express more high-level compositions such as "on-top-of", "touching", or "matching-screw-holes". Composite constraints are defined using the `constraint:` keyword followed by a name, a set of parameters, and **assertions of primitive or composite sub-constraints**.

```
1  (constraint: name (<parameters>) <assertions>))
```

Just like in the `constraints:` section of `shape:`, **the asserted sub-constraints are joined by logical conjunction**. Unlike composite shape definitions with only `constraints:` but without a `script:`, a composite constraint definition does not create a shape, and is purely used for combining constraints. As an example, the `valid-edge` composite constraint ensures that the edge between an inner and outer diameter has a value between 2 and 20.

```
1 (constraint: valid-edge (innerDia outerDia edge)
2  (assert: (= edge (- outerDia innerDia)))
3  (assert: (>= edge 2))
4  (assert: (<= edge 20)))
```

Instantiating composite constraints is similar to instantiating shapes, using the same syntax for binding parameter values, as discussed in Section 2.1. Composite constraints can be defined or called without specifying named parameters. However, PrintTalk's primitive comparative constraints, such as `=` and `<`, do not use named parameters. Instead, they operate on positional arguments, all of which are required.

2.2.2 Spatial Patterns Expressed as Constraints

Often, multiple shapes need to be arranged in a predefined pattern, such as the teeth of a gear in a circular pattern. PrintTalk's functionality for expressing spatial patterns is split into two parts: instantiation and positioning.

PrintTalk includes `for` loops through which multiple copies of a certain shape can be instantiated. A `for` loop contains an identifier, an iterable and a body. The iterable can either be a range of numbers or a shape. For each iteration of the loop, the identifier is bound to each number within that range, or each constituent of that shape.

```
1 (for <identifier> in <iterable>
2  <body>)
```

The shapes instantiated in the body of a `for` loop within a shape definition's script are composed with other shapes in the script by means of a union, unless specified otherwise by a `cut:` operation. To promote expressivity, `for` loops introduce pseudo-variables `i` and `n` representing the index and the total number of items to iterate over, respectively.

PrintTalk supports *anonymous shapes* that can be defined and instantiated in the script of other shapes. Hereby, the constituents instantiated in the body of the anonymous shape definition are "grouped" within a new shape, facilitating passing them as arguments to constraints. Apart from the absent name between the `shape:` keyword and the parameter list, the syntax for defining anonymous shapes is identical to defining named shapes.

Instantiating anonymous shapes is also identical to instantiating named shapes, by stating the shape and providing zero or more values for its named parameters. As an example, Listing 2 defines a gear (lines 1–11), in which a shape representing a single tooth is instantiated multiple times using a `for` loop (lines 7–8). These teeth are grouped within an anonymous shape definition without explicit parameters. The anonymous shape is instantiated (without arguments), and the instance is named `teeth` using a `named:` statement (lines 6–8).

To eliminate the need for repetitive manual positioning, PrintTalk allows spatial patterns to be expressed as composite constraints that place constraints on the positions of each of the involved shapes.

As an example, Listing 2 defines the `circular` pattern (lines 13–17) that is used for positioning the teeth of the gear. By means of a `for` loop, each shape in the pattern has its position along the x- and y-axes constrained, together with its rotation along the z-axis.

2.3 Output Statements

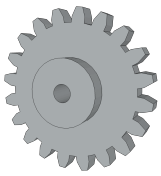
PrintTalk currently defines one output statement `print:` used for generating a 3D model and exporting it in the STL format.

Listing 2 Instantiating shapes multiple times using `for` loops in PrintTalk.

```
(shape: gear (!nr-teeth dia)
 (script:
  (cylinder #:diameter dia #:height 4)
  (cylinder #:diameter (/ dia 2) #:height 4 #:z 4)
  (cut: (cylinder #:diameter 4 #:height 8 #:z 2))
  (named: teeth ((shape: ()
                  (script: (for _ in (range nr-teeth)
                            (tooth #:l 4 #:w 3 #:h 4)))))))
 (constraints:
  (assert: (circular #:shapes teeth
                     #:radius (* (/ dia 2) 1.05)))))

(constraint: circular (!shapes radius)
  (for shape in shapes
   (assert: (= shape.x (* (cos (* (/ 360 n) i)) radius)))
   (assert: (= shape.y (* (sin (* (/ 360 n) i)) radius)))
   (assert: (= shape.rot-z (* (/ 360 n) i)))))

(print: (gear #:dia 32 #:nr-teeth 20) "gear.stl")
```



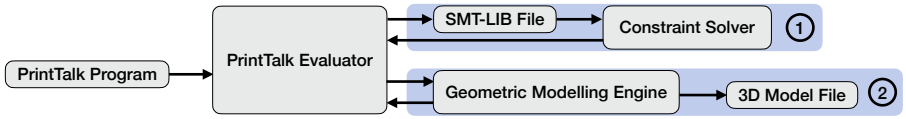


Figure 1 The PrintTalk toolchain.

```
(print: <shape> <filename>)
```

The `print:` statement conceptually ties together all necessary components for 3D model generation. When a shape is printed, that shape and all its constituents are instantiated, their constraints are checked and solved so that all constraint variables have valid values, and finally the 3D model is generated. Currently, `print:` is the only imperative, impure operation in PrintTalk, and is required to make the language practically usable. Imperative operations are executed solely for their side-effects and cannot participate in compositions with shapes or constraints.

2.3.1 Integrating the Z3 Solver

Constraints introduced in a PrintTalk program are solved using the Z3 solver [6]. To do so, the constraints are first compiled into an SMT-LIB file [2] that serves as the input for Z3. After solving these constraints, Z3 returns an S-expression containing the values to be assigned to the constraint variables. This process corresponds to the first part of the PrintTalk toolchain, depicted in Figure 1. Once all constraint variables have a value assigned to them, PrintTalk continues by building the 3D model to be exported as an STL file. PrintTalk makes use of the *Open CASCADE Technology (OCCT)* geometry library for generating STL files. This process corresponds to the second part of the PrintTalk toolchain.

Z3 [6] was selected primarily because it is an open-source solver supporting non-linear constraints, a capability lacking in considered alternatives like Cassowary [1]. PrintTalk actually uses a version of Z3 to which minor modifications were made in order to better

Table 1 Performance metrics of PrintTalk's constraint solving and 3D model generation processes.

3D model	Solving time	STL generation time
Pencil tray (Listing 1)	96 ms	259 ms
Gear (Listing 2)	511 ms	3464 ms
Lamp (Section 4)	2087 ms	15646 ms

suit PrintTalk's requirements. PrintTalk employs a version of Z3 extended via its C++ API to efficiently process both non-linear constraints and the soft constraints generated by the relative positioning mechanism (Section 2.1.2). This extension handles hard constraints first, then iteratively tests and adds satisfiable soft constraints in order of decreasing weight. This approach implements a *locally-predicate-better* comparator for determining the *constraint hierarchy* [4]. For performance considerations, our approach does not aim to maximise the number of constraints of the same weight that can be satisfied. Instead, constraints are added in the specific order that they are retrieved from a PrintTalk program (as described in Section 2.3.2). Soft constraints are rejected when they cannot be satisfied due to incompatibility with previously added (soft) constraints.

Performance is currently not the primary concern for PrintTalk. However, the constraint solving time (part 1 of the toolchain) is typically exceeded by the STL generation time (part 2 of the toolchain), suggesting the solver performance is not a limiting factor. Table 1 visualises the constraint solving and STL generation times for the 3D models described in this paper, measured on an M1 MacBook Pro.

2.3.2 Determining the Constraint Hierarchy

PrintTalk establishes a *deterministic* sequence for constraints based on a depth-first traversal of the shape tree (Section 2.1). Constituent shapes within a node are visited according to their instantiation order in the parent's script. During traversal, upon visiting a shape, its explicit (hard) constraints are first appended to the sequence in their specified order. Subsequently, the implicit (soft) constraints for default position (`x`, `y`, `z`) and rotation (`rot-x`, `rot-y`, `rot-z`) parameters are appended in that specific order. The weight associated with the soft constraints of constituent shapes increases by 1 relative to their parent, making root node constraints the weakest and leaf node constraints the strongest.

Upon solving the constraints, PrintTalk sorts the soft constraints by weight using a stable sorting algorithm, ensuring that the order in which the constraints were originally added to the list of constraints is respected. These rules implement a clear semantics that helps reasoning over PrintTalk programs and ensures that the output of a PrintTalk program corresponds to the programmer's expectations. For example, programmers can anticipate the eventual positioning of constituent shapes, knowing that soft constraints on the positions of earlier instantiated constituents have precedence over subsequent constituents.

3 Related Work

This section situates PrintTalk relative to existing approaches. First, we review the state of the art in programmatic CAD (PCAD) tools, noting their general lack of robust constraint mechanisms which motivates PrintTalk's design (Section 3.1). Given the limited approaches for constraint-solving in PCAD, we explore the extensive body of work on constraints within the related field of 2D layout planning (Section 3.2). Finally, we cover object-constraint languages which, similar to PrintTalk, exhibit a multi-paradigm nature by combining declarative constraints with object-oriented or imperative features (Section 3.3).

3.1 Programmatic CAD Tools

Programmatic CAD tools exist both as stand-alone domain-specific languages (DSLs) and as libraries to general-purpose languages. This section describes OpenSCAD and CadQuery, representing the state of the art in PCAD DSLs and libraries, respectively.

3.1.1 OpenSCAD

OpenSCAD [18] is a stand-alone PCAD language that represents shapes as a tree structure, where the root node represents the complete 3D shape and other nodes represent constituents that are composed by means of set-theoretic operations (union, difference, or intersection). PrintTalk and OpenSCAD therefore employ a similar modelling technique that is based on instantiating shapes and composing instances into more complex shapes. However, OpenSCAD lacks bidirectional constraints capable of both generating valid parameter values and validating provided ones. It only offers `assert` statements to check if parameter values meet specified requirements, for example when a dimension may not exceed a certain length. Apart from the declarative definition of 3D shapes, OpenSCAD also includes `if` statements and `for` loops through which 3D models can be constructed imperatively.

3.1.2 CadQuery

CadQuery [5] is a PCAD library for Python. Compared to PrintTalk and OpenSCAD, CadQuery provides a different way of modelling that relies more on extruding 2D surfaces to 3D shapes. For example, instead of directly instantiating a cylinder, first a circle on plane must be drawn, followed by an extrusion operation to form a 3D cylinder. CadQuery [5] only supports a limited set of geometric constraints [21], and the functionality of these constraints is limited as Python is not a constraint-based language. While programmers can import a constraint-solving library for Python and manually implement constraints, we consider this beyond the scope of CadQuery's inherent functionality. Another limitation of CadQuery is that the programmer must keep track of all constraints that are added within a design and make sure that all constraints are solved at appropriate moments by invoking the `solve()` method.

3.2 Constraints for 2D Layout

Constraint-solving techniques are frequently applied in domains requiring the precise spatial arrangement of 2D objects, such as user interface design and graph layout.

3.2.1 UI Design

A plethora of related work describes tools and approaches for developing UI layouts using constraints that specify how elements should be arranged on a screen [3]. An example of this is Apple's UIKit, which integrates the Cassowary solver [1] to support constraints for designing interfaces of iOS applications. Another layout engine [16] allows UI layouts to be described using ordinal and linear constraints, so that the UI can be dynamically adjusted by resolving constraints. Other related work includes ALM [15], a constraint language through which UI elements can be constrained using both hard and soft constraints. Similar to PrintTalk, ALM makes use of soft constraints and constraint hierarchies to determine which constraints have precedence over others in order to find the most optimal layout.

3.2.2 Graph Layout

Another related domain concerned with positioning elements in a 2D space is the graph-layout domain. Several algorithms have been developed to represent 2D graphs in an optimal manner, where constraints are used for arranging the nodes and edges of graphs on a 2D canvas in such a way that none of the nodes overlap and none of the edges cross each other. One such algorithm takes suggested node positions and constraints on these positions as input [12]. The algorithm aims to find a solution in which the nodes are as close to their suggested positions as possible, while satisfying all constraints placed on these positions. The suggestions for node positions have an associated weight, and the algorithm prioritises suggestions with a higher weight. Another algorithm uses constraints for expressing aesthetic criteria such as symmetry and alignment [7]. These constraints also have an associated weight through which the graph layout's aesthetic can be optimized.

3.3 Object-Constraint Languages

Existing research explores languages that integrate both concepts of object-oriented programming and constraints. Two such languages, s-Comma [20] and Babelsberg [8], share many similarities with PrintTalk. PrintTalk shapes are similar to classes in s-COMMA, as constraints can be placed on the parameters of instances of s-Comma classes, but can also be part of the class definition. Babelsberg [8] is another object-constraint language that combines imperative programming with constraints. Babelsberg is implemented by extending Ruby with constraints and allowing variables to partake in these constraints. Upon evaluating a Babelsberg program, the interpreter switches between an imperative evaluation mode and a constraint evaluation mode at appropriate times. PrintTalk has similar characteristics, as a shape definition's script is evaluated first, after which all constraints are collected and solved so that values can be assigned to the constraint variables.

4 Evaluation

We evaluate PrintTalk against OpenSCAD and CadQuery, representing the state of the art in stand-alone PCAD languages and libraries. Our evaluation focuses on how each tool addresses the problems outlined in Section 1.1: parameter explosion, leaky compositional abstraction, and imperative compositional specification. We compare PrintTalk, OpenSCAD, and CadQuery programs for modelling the same 3D model – a desk lamp illustrated in Figure 2 – with the same level of detail and features, and offering similar control over shape parameters. The lamp is composed out of a lamp shade, a base, arms, and caps for the hinge points. Each program uses the most suitable features that each tool offers for modelling the lamp. The complete code listings are available as supplementary material,[1] and only the most relevant snippets are included in this section. Appendix B contains an extended evaluation of PrintTalk, in which two additional 3D models are covered.

4.1 Criterium 1: Flexible Variability and Reusability

PrintTalk

PrintTalk permits parametrised components, such as shapes and composite constraints, to be instantiated with only a subset of their parameters explicitly assigned values. The constraint solver subsequently determines values for the remaining unbound parameters.

[1] https://doi.org/10.5281/zenodo.15593661

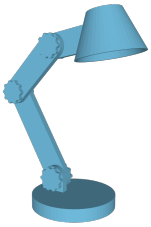


Figure 2 3D model of a lamp used for the evaluation of PrintTalk.

Listing 3 PrintTalk constraints through which valid parameter values can be determined.

```
40 (shape: lamp-base (dia height screw-dia screw-x screw-y screw-z mnt-l mnt-w mnt-th)
41  (script:
42   (named: base (cylinder #:diameter dia #:height height))
43   (cuboid #:height mnt-l #:width mnt-w #:length mnt-th
44    #:x (/ mnt-th -2) #:y (- (/ dia 2) (+ (/ mnt-w 2) 10)) ... )
45   (cylinder #:diameter mnt-w #:height mnt-th
46    #:x (/ mnt-th -2) #:y (- (/ dia 2) (+ (/ mnt-w 2) 10)) ... )
47   (cut: (named: sh (cylinder #:diameter screw-dia #:height mnt-th
48    #:x (/ mnt-th -2) #:y (- (/ dia 2) (+ (/ mnt-w 2) 10)) ... ))))
49  (constraints:
50   (assert: (>= mnt-l (/ mnt-w 2)))
51   (assert: (>= dia (+ mnt-w 5)))
52   (assert: (>= height 5))
53   (assert: (= screw-x (- x (/ mnt-th 2))))
54   (assert: (= screw-y (+ y (- (/ dia 2) (+ (/ mnt-w 2) 10)))))
55   (assert: (= screw-z (+ z (/ height 2) mnt-l)))))
```

Constraints incorporated into a component's definition serve to guide the solver towards valid solutions, for example ensuring positive dimensions for shapes. These constraints also improve reusability by validating explicitly provided parameter values, triggering an error if a value violates an established condition. Responsibility for defining adequate constraints rests with the programmer, as PrintTalk's validation capability is confined to the constraints specified within the component definition.

As an example, Listing 3 defines a shape named `lamp-base` containing parameters `dia` and `height` representing the diameter and height of the base of the lamp, and constraints are in place to set minimum dimensions (lines 51–52).

OpenSCAD

Parametrised OpenSCAD shapes can be defined using the `module` keyword and can, similar to PrintTalk, also be instantiated with optionally bound parameters. Unbound parameters in OpenSCAD get an `undef` value, and a warning message is raised when `undef` values are used within the body of the shape's definition. Like PrintTalk, this body can contain assertions for ensuring that parameter values are valid. Assertions in OpenSCAD take expressions representing conditions as argument, and raise an error when these conditions cannot be satisfied. Contrary to PrintTalk, there is no constraint solver that can improve the reusability of 3D components by using these assertion for determining suitable parameter values automatically. To generate valid shapes, programmers must manually find suitable values for all of a shape's parameters upon instantiation. In Listing 4, the `lamp_base` shape definition contains `assert` statements modelling the same parameter requirements as the equivalent PrintTalk `lamp-base` shape definition in Listing 3.

Listing 4 OpenSCAD assertions through which invalid parameter values can be detected.

```
41 module lamp_base(dia, height, screw_dia, mnt_l, mnt_w, mnt_th) {
42  assert(mnt_l >= mnt_w/2);
43  assert(dia >= (mnt_w + 5));
44  assert(height >= 5);
45  ... }
```

CadQuery

To modularise CadQuery shapes, programmers must make use of Python's language features such as functions or classes for defining new shapes. These allow shapes to be modularized through function parameters or parametrised constructor methods, respectively. While Python functions and constructor methods support default parameter values, there is no mechanism in place that allows programmers to leave parameters unbound, relying on a constraint solver to find suitable parameters. Furthermore, invalid parameter values cannot be detected without resorting to writing tedious `if` tests and raising errors accordingly.

4.2 Criterium 2: Flexible Composition

PrintTalk

Generally, complex 3D shapes are defined by composing simpler constituent shapes. For example, the lamp is designed out of individual components representing the base, arms, hinges, and a lamp shade. To correctly compose these shapes, anchor points indicating where other shapes can be attached in a composition must be defined. In PrintTalk, these points can be defined within a shape's definition, and exposed as parameters of that shape.

In the case of our lamp, the `lamp-base` component exposes compositional parameters `screw-x`, `screw-y`, and `screw-z` as the anchor point of the base to which a lamp arm can be attached. Constraints are in place to express the relations between these compositional parameters and the dimensional parameters of the `lamp-base` shape (lines 53–55 in Listing 3).

Although anchor points represented as compositional parameters increase the parameter count of composable shapes, we find this approach to be the sweet spot between reusability and composability. Because parameters can be left unbound when instatiating PrintTalk shapes, this provides flexibility in the way that shapes can be reused:

- **Anchor points are derived from other parameters:** By specifying the constituent's other parameters, its anchor points can be derived, and are essentially "locked" in place. In PrintTalk, the anchor point defined through composition parameters can be accessed and used to align other constituents in the composition with them.
- **Other parameters are derived from anchor points:** By constraining the constituent's anchor points, and leaving other parameter values such as dimensions *underconstrained*, these dimensions can be derived upon composition, resulting in a shape that fits the composition, without requiring the designer to manually determine all dimensions.

OpenSCAD

OpenSCAD has no support for anchor points, and the modules through which shapes can be abstracted and modularised offer no support for exposing information such as anchor points upon composition. Instead, anchor points must be calculated manually, external to the shape. Yet, calculating the anchor points of a constituent requires information a how the constituent is composed itself. This form of "leaky abstraction" complicates the reuse of shapes in OpenSCAD.

Listing 5 Additional calculations needed to determine the anchor point of the `lamp_base` shape.

```
77 base_screw_y=(base_dia/2)-(arm_width/2)-10;
78 base_screw_z = base_height/2+(arm_width-screw_dia);
79
80 lower_arm_pos_delta_y = sin(lower_arm_angle)*(lower_arm_length/2);
81 lower_arm_y=base_screw_y-lower_arm_pos_delta_y;
82 lower_arm_pos_delta_z = cos(lower_arm_angle)*(lower_arm_length/2);
83 lower_arm_z=base_screw_z+lower_arm_pos_delta_z;
```

Listing 6 Tagging CadQuery geometries in order to use them as composition parameters.

```
47 hole = base_shape.faces("%PLANE").faces(">Z").faces(">X")\
48     .edges("%CIRCLE").edges(">>Z[-2]")
49 hole.tag("hole")
```

For example, in Listing 5, additional calculations for determining the anchor point of the `lamp_base` shape are needed in order to correctly use that shape in a composition where the screw hole of the base needs to line up with the screw hole of the lower arm.

CadQuery

CadQuery allows for shapes to be composed in "Assemblies", and composition happens either by means of constraints, or by manually specifying the location, rotation, and scale of each constituent in the composition. CadQuery provides a limited set of constraints that can be used for expressing relations between parameters of constituent shapes. To determine their composition parameters, shapes can be queried to obtain their geometries such as surfaces, edges, and points, which can be tagged using the `tag` method. For example, the code snippet in Listing 6 is used to tag the screw hole of the base of the lamp.

After being tagged, geometries can be queried from their shape by using their tag, and can be used as composition parameters that can partake in composition constraints. Listing 7 illustrates how the screw hole of the base is constrained to line up with the screw hole of the lower arm of our lamp using CadQuery's built-in `Plane` constraint.

It is only possible to tag geometries that are present in a shape, and it is not possible to tag arbitrary anchor points relative to the shape. For example, it is not possible to tag a specific spot where an on-off switch can be added to our lamp, without there being a geometry that can be tagged on that exact spot.

4.3 Criterium 3: Declarative and Reusable Compositional Abstraction

PrintTalk

Defining shapes that contain constituent shapes placed in a pattern, such as the caps used for the hinges in our lamp, requires the functionality for instantiating a number of shapes, and placing each shape in the desired location. This functionality is split over two components in PrintTalk, facilitating the reuse of spatial patterns through constraints. Similar to shapes, constraints expressing spatial patterns can be defined once and reused later. Moreover, the

Listing 7 Composing CadQuery shapes by constraining their tagged geometries.

```
113 .constrain("base?hole", "lower_arm?front_bottom_hole", "Plane", param=0)\
```

Listing 8 Use of the `circular` spatial pattern in PrintTalk for placing shapes in a circular pattern.

```
1 (shape: cap (dia height)
2  (script:
3   (cylinder #:diameter dia #:height height)
4   (cut: (named: cutouts ((shape: ()
5                          (script:
6                           (for _ in (range 10)
7                            (cylinder #:diameter (/ dia 4) #:height height))))))))
8  (constraints:
9   (assert: (circular #:shapes cutouts #:radius (* dia 0.55)))))
```

Listing 9 OpenSCAD code using `for` loops for placing shapes in a circular pattern.

```
1 module cap(dia, height){
2  difference(){cylinder(h=height, d=dia, center=true);
3   union(){
4    for(i=[0:9]){
5     translate([cos(36*i)*dia*0.55, sin(36*i)*dia*0.55, 0])
6      cylinder(h=height, d=dia/4, center=true);
7 }}}}
```

pattern definition is independent of the shapes to be placed in that pattern.

For example, the `cap` shape definition in Listing 8 reuses the `circular` pattern defined in Listing 2 for modelling the 10 cutouts of the caps of the lamp's hinge-points (line 9).

OpenSCAD and CadQuery

In contrast, OpenSCAD and CadQuery require `for` loops to achieve the same, hampering the abstraction of patterns. This is illustrated in the OpenSCAD snippet in Listing 9 that uses a single `for` loop for instantiating and positioning the cutouts of the caps of the lamp's hinge-points (lines 4–6). As the parameters of shapes are immutable, and it is impossible to reposition shapes after their instantiation, shapes must be positioned correctly upon instantiation, which makes it impossible to split the functionalities in a way that allows for abstracting spatial patterns.

5 Conclusion

We presented PrintTalk, a constraint-based language for programmatic 3D modelling. PrintTalk enables 3D models to partake in compositions, as constituents to more complex shape, and features an composition mechanism that allows relations between constituents and their parameters to be expressed in a declarative manner, using constraints. We implemented PrintTalk as a domain-specific language on top of Racket. In order to evaluate PrintTalk, we compared it to state of the art in programmatic modelling tools, represented by OpenSCAD and CadQuery. From this evaluation, we conclude that PrintTalk's composition mechanism promotes the reusability of 3D shapes, with an underlying constraint solver that helps programmers with finding suitable parameter values for constituents, such that they can be composed into valid 3D models.

References

1 Greg J. Badros, Alan Borning, and Peter J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, December 2001. doi:10.1145/504704.504705.

2 Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

3 Alan Borning and Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Trans. Graph.*, 5(4):345–374, October 1986. doi:10.1145/27623.29354.

4 Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 48–60, New York, NY, USA, 1987. Association for Computing Machinery. doi:10.1145/38765.38812.

5 GitHub - CadQuery/cadquery. https://github.com/CadQuery/cadquery. [Accessed 02-04-2025].

6 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-78800-3_24.

7 E. Dengler, M. Friedell, and J. Marks. Constraint-driven diagram layout. In *Proceedings 1993 IEEE Symposium on Visual Languages*, pages 330–335, 1993. doi:10.1109/VL.1993.269619.

8 Tim Felgentreff, Alan Borning, and Robert Hirschfeld. *Babelsberg: Specifying and solving constraints on object behavior*, volume 81. Universitätsverlag Potsdam, 2014.

9 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Commun. ACM*, 61(3):62–71, February 2018. doi:10.1145/3127323.

10 Jessie Frazelle. A new era for mechanical cad. *Commun. ACM*, 64(10):36–39, September 2021. doi:10.1145/3464909.

11 Joseph Gil and David H. Lorenz. Environmental acquisition: A new inheritance-like abstraction mechanism. *SIGPLAN Not.*, 31(10):214–231, October 1996. doi:10.1145/236338.236358.

12 Weiqing He and Kim Marriott. Constrained graph layout. *Constraints*, 3(4):289–314, 1998. doi:10.1023/A:1009771921595.

13 ImplicitCad — implicitcad.org. https://www.implicitcad.org/. [Accessed 02-04-2025].

14 GitHub - jscad/OpenJSCAD.org. https://github.com/jscad/OpenJSCAD.org. [Accessed 02-04-2025].

15 Christof Lutteroth, Robert Strandh, and Gerald Weber. Domain specific high-level constraints for user interface layout. *Constraints An Int. J.*, 13(3):307–342, 2008. doi:10.1007/S10601-008-9043-2.

16 Christof Lutteroth and Gerald Weber. User interface layout with ordinal and linear constraints. In *User Interfaces 2006, 7th Australasian User Interface Conference (AUIC 2006), Hobart, Tasmania, Australia, January 16-19 2005*, volume 50 of *CRPIT*, pages 53–60. Australian Computer Society, 2006. URL: https://dl.acm.org/doi/10.5555/1151758.1151764.

17 Aman Mathur, Marcus Pirron, and Damien Zufferey. Interactive programming for parametric cad. *Computer Graphics Forum*, 39(6):408–425, 2020. doi:10.1111/cgf.14046.

18 OpenSCAD — openscad.org. https://openscad.org/. [Accessed 02-04-2025].

19 GitHub - jeff-dh/SolidPython. https://github.com/jeff-dh/SolidPython. [Accessed 02-04-2025].

20 Ricardo Soto and Laurent Granvilliers. *On the Pursuit of a Standard Language for Object-Oriented Constraint Modeling*, pages 123–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-79355-7_12.

21 Hiromasa Suzuki, Hidetoshi Ando, and Fumihiko Kimura. Geometric constraints and reasoning for geometrical cad systems. *Computers & Graphics*, 14(2):211–224, 1990. doi:10.1016/0097-8493(90)90033-T.

A Formal Syntax of PrintTalk

Figure 3 gives an overview of PrintTalk's formal syntax. PrintTalk is built on top of Racket [9] and inherits its s-expression syntax. Similar to Racket, PrintTalk makes use of prefix-notation, where function applications are structured with the operator preceding the operands: `(operator operand1 operand2 ...)`.

Identifiers are represented as an arbitrary number of characters, excluding white spaces and colons. Also similar to Racket, identifiers starting with `#:` are reserved for named parameters. Additionally, identifiers starting with an exclamation mark are reserved for parameters that are forced to act as constructor parameters, as discussed in Section 2.1.1.

Strings are represented as an arbitrary number of characters between double quotes (e.g., `"this is a string"`), and are primarily used for representing filenames that are provided as arguments to `print:` statements.

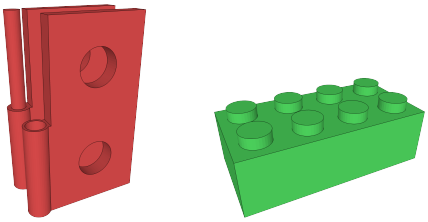
Numbers are either integers (e.g., `42`), or floating point numbers (e.g., `0.33`).

⟨*program*⟩ ::= ⟨*top-level expression*⟩*	*PrintTalk program*
⟨*top-level expression*⟩ ::=	*Top-level expressions*
⟨*expr*⟩	*Expression*
| ⟨*definition*⟩	*Definition*
| `(print:` ⟨*expr*⟩ string`)`	*Exporting shapes*
⟨*expr*⟩ ::=	*Expressions*
⟨*var*⟩	*Variable*
| ⟨*literal*⟩	*Literal*
| ⟨*application*⟩	*Function application*
| ⟨*expr*⟩`.`⟨*var*⟩	*Variable reference*
| `(named:` ⟨*var*⟩ ⟨*expr*⟩`)`	*Naming shapes or drawings*
| `(for` ⟨*var*⟩ `in` ⟨*expr*⟩ ⟨*expr*⟩*`)`	*For loop*
| `(shape: (`⟨*var*⟩*`) (script:` ⟨*expr*⟩*`))`	*Shape constructor*
| `(shape: (`⟨*var*⟩*`) (script:` ⟨*expr*⟩*`) (constraints:` ⟨*assertion*⟩*`))`	*Shape constructor*
⟨*application*⟩ ::= `(`⟨*expr*⟩ ⟨*expr*⟩*`)`	*Function application*
⟨*definition*⟩ ::=	*Definitions*
`(shape:` ⟨*var*⟩ `(`⟨*var*⟩*`) (script:` ⟨*expr*⟩*`))`	*Shape definition*
| `(shape:` ⟨*var*⟩ `(`⟨*var*⟩*`) (script:` ⟨*expr*⟩*`) (constraints:` ⟨*assertion*⟩*`))`	*Shape definition*
| `(constraint:` ⟨*var*⟩ `(`⟨*var*⟩*`)` ⟨*assertion*⟩*`)`	*Constraint definition*
⟨*assertion*⟩ ::= `(assert:` ⟨*expr*⟩`)`	*Assertion*
⟨*var*⟩ ::= identifier	*Variables*
⟨*literal*⟩ ::= number or string	*Literals*

Figure 3 PrintTalk's formal syntax.

B Extended Evaluation

This section provides an extended evaluation in which we consider hinges and building blocks (Figure 4) as two additional cases for comparing PrintTalk to OpenSCAD and CadQuery based on the three criteria of Section 1.1. The complete code listings are available as supplementary material,[2] and only the most relevant snippets are included in this section.



(a) hinge (b) brick

Figure 4 3D model of (a) a hinge, and (b) a building brick.

B.1 Criterium 1: Flexible Variability and Reusability

The hinge shape declares several parameters representing dimensions that must be compatible with each other. For example, the diameter of the screw holes must be smaller than the widths and heights of the leafs, the diameter of the knuckle must be larger than the thickness of the leaf, and the diameter of the pin must be in proportion to the length and width of the leaf.

PrintTalk

In PrintTalk, these dimensional properties can be modelled by constraints, so that the underlying constraint solver can raise an error when the provided parameter values do not satisfy the constraints. If a PrintTalk program does not assign a value to a parameter, it is treated as a constraint variable, and its value is then computed by the constraint solver, based on applicable constraints, before the 3D model is generated.

Listing 10 PrintTalk constraint for ensuring valid dimensions.

```
(shape: leaf (w t h knuckle-dia screw-dia)
 (script:
  (cuboid #:length w #:width t #:height h)
  (cut: (cylinder #:diameter screw-dia #:height t ...))
  (cut: (cylinder #:diameter screw-dia #:height t ...))
  (cuboid #:length (/ knuckle-dia 2) #:width t #:height (/ h 2) ...)
  (cylinder #:diameter knuckle-dia #:height (/ h 2) ...))
 (constraints:
  (assert: (>= screw-dia (/ w 8)))
  (assert: (< screw-dia w))
  (assert: (< screw-dia (/ h 2)))
  (assert: (>= knuckle-dia t))))
```

[2] https://doi.org/10.5281/zenodo.15593661

OpenSCAD

In OpenSCAD, it is possible to add `assert` statements for "constraining" parameters and raising an error when a provided parameter value does not meet an asserted condition. However, as there is no underlying constraint solver, OpenSCAD is not able to automatically compute values for parameters based on assertions.

Listing 11 OpenSCAD assertions for modelling conditions on parameter values.

```
1  module leaf (w, t, h, knuckle_dia, screw_dia){
2     assert(screw_dia>=w/8);
3     assert(screw_dia<w && screw_dia<h/2);
4     assert(knuckle_dia>=t);
5     ...
```

CadQuery

While CadQuery itself does not provide features for asserting conditions that parameter values must adhere to, this functionality can be achieved by manually programming `if` tests and raising Python exceptions accordingly. Similar to OpenSCAD, CadQuery is unable to automatically compute values of parameters for which no value was provided upon shape instantiation.

Listing 12 CadQuery `if`-tests and exceptions for modelling conditions on parameter values.

```
3  def leaf(w, t, h, knuckle_dia, screw_dia):
4      if screw_dia < w/8:
5          raise Exception("minimum value for screw_dia")
6      if screw_dia > w or screw_dia>h/2:
7          raise Exception("maximum value for screw_dia")
8      if knuckle_dia < t:
9          raise Exception("minimum value for knuckle_dia")
10     ...
```

B.2 Criterium 2: Flexible Composition

A `leaf` shape, representing the leaf and the solid part of the knuckle, is reused twice for constructing the complete `hinge` shape, with a pin attached to one leaf and a hole cut from the other leaf. To model a valid hinge, the diameters of the pin and the hole must be determined so that these fit together. Furthermore, the diameter of the knuckle must be sized accordingly, so that it is larger than the pin and larger than the thickness of the leaf.

PrintTalk

In PrintTalk, parameters and constraints can be used for representing dimensions and expressing relations between them. As an example, the `hinge` shape of Listing 13 contains parameters for representing the diameter of the pin (`pin-dia`), the diameter of the knuckle (`knuckle-dia`) and the rim around the pin (`pin-rim`). From within the `hinge`'s `constraints:` section (lines 22–26), composition constraints are placed on these parameters to ensure that they are sized appropriately, relative to each other. One of these parameters – `knuckle-dia` – is passed as an argument to the corresponding parameter of the `leaf` constituents (lines 17 and 20). Through PrintTalk's scoping and parameter-passing mechanism, additional

Listing 13 PrintTalk constraint for expressing a linear pattern.

```
(shape: hinge (w t h screw-dia pin-dia knuckle-dia pin-rim)
 (script:
  ;; Leaf with pin
  (named: leaf1 (leaf ... #:knuckle-dia knuckle-dia #:screw-dia screw-dia))
  (named: pin (cylinder #:diameter (- pin-dia 1) #:height (/ h 2) ...))
  ;; Leaf with hole
  (named: leaf2 (leaf ... #:knuckle-dia knuckle-dia #:screw-dia screw-dia ...)))
  (cut: (named: pin-hole (cylinder #:diameter pin-dia #:height (/ h 2) ...))))
 (constraints:
  (assert: (>= pin-dia (* 0.8 knuckle-dia)))
  (assert: (< pin-dia knuckle-dia))
  (assert: (= pin-rim (/ (- knuckle-dia pin-dia) 2)))
  (assert: (>= pin-rim (/ pin-dia 8)))))
```

constraints can be added to the `knuckle-dia` constraint variable from within the constituent `leaf` shapes. In this case, an additional constraint stating that the diameter of the knuckle (`knuckle-dia`) must be larger than the thickness of the leaf (`t`) is added from within the `leaf` shape (line 12 of Listing 10). In PrintTalk, if the `hinge` shape is instantiated without values specified for the `pin-dia`, `knuckle-dia`, and `pin-rim` parameters, these parameters are treated as constraint variables. Their values are then determined based on constraints originating from both the `hinge` shape itself and its constituent `leaf` shapes.

OpenSCAD and CadQuery

In OpenSCAD and CadQuery, it is not possible to add composition constraints that express relationships between parameter values of constituents, and automatically derive parameter values based on these constraints. Instead, parameter values must be determined manually, and provided as arguments upon shape instantiation. For the `hinge` shape, this means that values for `pin_dia` and `knuckle_dia` must be provided upon instantiating the shape. Determining valid values for these parameters, however, requires inspecting the code of the constituents, taking into account conditions (expressed as assertions or `if`-tests) on parameter values and any exceptions these conditions potentially raise.

B.3 Criterium 3: Declarative and Reusable Compositional Abstraction

The brick shape (Figure 4(b) & Figure 5) includes two rows of cylinders on its top surface and a row of hollow cylinders on its bottom surface. The cylinders within each row are arranged in a linear pattern.

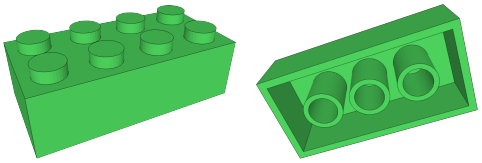


Figure 5 Brick with linear patterns on its top (left image) and on its bottom (right image).

Listing 14 PrintTalk constraint for expressing linear patterns.

```
(constraint: linear-x (!shapes x0 xn dx)
 (for shape in shapes
  (assert: (= xn (+ x0 (* (- n 1) dx))))
  (assert: (= shape.x (+ x0 (* dx i))))))
```

Listing 15 The brick shape modelled by reusing the `linear-x` constraint in PrintTalk.

```
(shape: bottom-cyl (height)
 (script:
  (cylinder #:diameter 6.5 #:height height)
  (cut: (cylinder #:diameter 4.8 #:height height))))

(shape: block (l w h clearance)
 (script:
  (cuboid #:length (- (* 8 l) clearance) #:width (- (* 8 w) clearance) ...)
  (cut: (cuboid #:length (- (- (* 8 l) clearance) 2.4) ...))))

(shape: brick (!l)
 (script:
  (block #:l l #:w 2 #:h 9.6 #:clearance 0.2)
  (named: bot-cyls ((shape: () (script:
           (for i in (range (- l 1))
              (bottom-cyl #:height 8.4 #:z -0.6))))))
  (named: top-cyls-c1 ((shape: () (script:
           (for i in (range l)
              (cylinder #:diameter 4.8 #:height 1.8 #:y -4 #:z 5.7))))))
  (named: top-cyls-c2 ((shape: () (script:
           (for i in (range l)
              (cylinder #:diameter 4.8 #:height 1.8 #:y 4 #:z 5.7)))))))
 (constraints:
  (assert: (linear-x #:shapes bot-cyls #:x0 (- (* (- (/ l 2) 1) 8)) #:dx 8))
  (assert: (linear-x #:shapes top-cyls-c1 #:x0 (- (-(* (/ l 2) 8) 4)) #:dx 8))
  (assert: (linear-x #:shapes top-cyls-c2 #:x0 (-(-(* (/ l 2) 8) 4)) #:dx 8))))

(print: (brick #:l 4) "brick-4.stl")
```

PrintTalk

In PrintTalk, the logic for arranging shapes in a pattern can be abstracted as constraints. This abstraction eliminates the need to repeat positioning calculations each time shapes are subsequently placed in that pattern. For example, the `linear-x` constraint in Listing 14 expresses the pattern used for modelling the rows of cylinders on the top and bottom surfaces of the brick. As a result, the code for instantiating cylinders can be separated from the code for positioning them.

OpenSCAD and CadQuery

OpenSCAD and CadQuery do not feature abstraction mechanisms that allow the code for instantiating shapes to be decoupled from the code for positioning these shapes. As a result, the calculations for linearly positioning the cylinders on the top and bottom surfaces of the brick must be repeated for each row of cylinders.

Listing 16 In OpenSCAD, calculations for determining the position of shapes must be repeated for each row of cylinders.

```
module brick (l){
    cyl_delta = 8;
    bottom_cyl_begin_x = -(l/2-1)*cyl_delta;
    top_cyl_begin_x = -((l/2)*cyl_delta)+cyl_delta/2;
    union(){
        block(l=l, w=2, h=9.6, clearance=0.2);
        for(i=[0:l-2]){
            translate([bottom_cyl_begin_x+i*cyl_delta, 0, -0.6])
                bottom_cyl(height=8.4);
        }
        for(i=[0:l-1]){
            translate([top_cyl_begin_x+i*cyl_delta, -4, 5.7])
                cylinder(h=1.8, d=4.8, center = true);
        }
        for(i=[0:l-1]){
            translate([top_cyl_begin_x+i*cyl_delta, 4, 5.7])
                cylinder(h=1.8, d=4.8, center = true);
        }
    }
}
```

Scalable Counting of Minimal Trap Spaces and Fixed Points in Boolean Networks

Mohimenul Kabir[1]
School of Computing, National University of Singapore, Singapore

Van-Giang Trinh
Inria Saclay, EP Lifeware, Palaiseau, France

Samuel Pastva
Faculty of Informatics, Masaryk University, Brno, Czechia

Kuldeep S Meel
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

Abstract

Boolean Networks (BNs) serve as a fundamental modeling framework for capturing complex dynamical systems across various domains, including systems biology, computational logic, and artificial intelligence. A crucial property of BNs is the presence of trap spaces – subspaces of the state space that, once entered, cannot be exited. Minimal trap spaces, in particular, play a significant role in analyzing the long-term behavior of BNs, making their efficient enumeration and counting essential. The fixed points in BNs are a special case of minimal trap spaces. In this work, we formulate several meaningful counting problems related to minimal trap spaces and fixed points in BNs. These problems provide valuable insights both within BN theory (e.g., in probabilistic reasoning and dynamical analysis) and in broader application areas, including systems biology, abstract argumentation, and logic programming. To address these computational challenges, we propose novel methods based on *approximate answer set counting*, leveraging techniques from answer set programming. Our approach efficiently approximates the number of minimal trap spaces and the number of fixed points without requiring exhaustive enumeration, making it particularly well-suited for large-scale BNs. Our experimental evaluation on an extensive and diverse set of benchmark instances shows that our methods significantly improve the feasibility of counting minimal trap spaces and fixed points, paving the way for new applications in BN analysis and beyond.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Computing methodologies → Logic programming and answer set programming; Applied computing → Systems biology

Keywords and phrases Computational systems biology, Boolean network, Fixed point, Trap space, Answer set counting, Projected counting, Abstract argumentation, Logic programming

Digital Object Identifier 10.4230/LIPIcs.CP.2025.17

Related Version *Full Version*: `https://arxiv.org/abs/2506.06013`

Supplementary Material *Software (Source Code)*: `https://github.com/meelgroup/bn-counting` archived at `swh:1:dir:2a842e8c2c7e9c2ba0d7f9c6249eab0e747bcde8`
Dataset: `https://zenodo.org/records/15141045`

Funding The work was supported by the National University of Singapore and the Natural Sciences and Engineering Research Council of Canada (NSERC) [RGPIN-2024-05956].

Acknowledgements We are thankful to the reviewers for their feedback on the paper. The computational work of this paper was conducted in the Niagara cluster.

[1] Corresponding author

© Mohimenul Kabir, Van-Giang Trinh, Samuel Pastva, and Kuldeep S Meel;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Boolean Networks (BNs) serve as a fundamental modeling framework for representing complex dynamical systems across domains such as systems biology, computational logic, and artificial intelligence [19, 31, 56, 58, 73]. Their ability to capture intricate interactions and complex system behaviors makes them a valuable tool for studying diverse phenomena such as gene regulatory networks, logical circuits, and reasoning processes [5, 56]. A crucial aspect of BN dynamics is the existence of trap spaces – subspaces of the state space that, once entered, cannot be exited [44]. These structures are instrumental in understanding the long-term behavior of BNs, as they often correspond to stable system configurations or attractors [44].

Among trap spaces, minimal trap spaces (fixed points are a special case of minimal trap spaces in which all variables are *fixed* [44]) are of particular interest due to their role in characterizing the dynamical landscape of a BN [44, 75]. The identification and counting of minimal trap spaces provide valuable insights into the system stability, attractor structure, and probabilistic behavior [44, 32]. However, enumerating these structures poses a significant challenge due to their computational complexity, especially for large-scale BNs encountered in practical applications [68]. Scalable methods for counting minimal trap spaces are therefore essential for advancing the BN analysis.

Unfortunately, minimal trap spaces have received little attention in the context of counting problems for Boolean networks. In contrast, the problem of counting fixed points has been studied in several works [67, 66, 32, 12]. These works show that fixed point counting is generally #P-complete and examine its complexity under structural restrictions [67, 66] or specific classes of Boolean update functions [32]. Some studies also consider scenarios in which the update logic is only partially known [12]. On the practical side, while no dedicated implementation exists for fixed point counting, the task can be reduced to propositional model counting [60] via a CNF encoding of fixed points [20]. To the best of our knowledge, no prior work – either theoretical or practical – has addressed the problem of counting minimal trap spaces in Boolean networks.

Our paper addresses the research gap by formulating six meaningful problems related to counting minimal trap spaces and fixed points in BNs. These counting problems capture core tasks such as counting all minimal trap spaces or fixed points, counting those that satisfy a given *property* (e.g., a *phenotype*), and counting solutions under *perturbations*. These problems provide valuable insights not only within BN theory (e.g., probabilistic reasoning and dynamical analysis) but also in broader application areas such as abstract argumentation and logic programming (discussed in Section 4). We subsequently propose novel and efficient methods to solve the counting problems by exploiting the expressive power of Answer Set Programming (ASP) [50]. ASP is a declarative problem solving paradigm and has widely been applied in the field of systems biology [57, 76], in particular in the analysis and control of BNs [1, 42, 44, 55, 68, 71] (see Section 3). As in existing ASP counting literature [39], our ASP-based reduction leverages the approximate answer set counter ApproxASP [38], which employs a *hashing-based* technique for approximate answer set counting. Finally, we conduct an extensive experimental evaluation on a diverse benchmark dataset. Our analysis shows that ApproxASP efficiently estimates the number of minimal trap spaces and fixed points, and ApproxMC [77] efficiently estimates fixed point counts. Both approaches avoid exhaustive enumeration, significantly improving the feasibility of counting compared to enumeration and BDD-based methods used in other tools.

The remainder of the paper is structured as follows. In Section 2, we review the necessary background on propositional logic, BNs, answer set programming, and model counting. Section 3 surveys related work. Section 4 defines the six counting problems we consider

and discusses their applications. Section 5 introduces our ASP-based reduction methods for solving these problems. Section 6 presents experimental results, demonstrating the effectiveness of our methods. Finally, Section 7 concludes the paper and outlines potential directions for future research.

2 Preliminaries

In this section, we present background and some notations from propositional logic, Boolean networks, and answer set programming.

2.1 Propositional Formulas

In this work, we employ $\mathbb{B} = \{0, 1\}$ as the Boolean domain and $\mathbb{B}_\star = \{0, 1, \star\}$ as the three-valued domain. Under propositional semantics, each *propositional variable* v takes its value from $\mathbb{B}$. A propositional formula is defined recursively: the basic formulas include the Boolean constants 0 and 1, any variable v, and its negation $\neg v$. Moreover, if a formula is prefixed by $\neg$ (not) or if two formulas are connected by a logical connective – $\wedge$ (conjunction), $\vee$ (disjunction), and $\leftrightarrow$ (bi-implication) – the resulting expression is also a formula. A formula is in Conjunctive Normal Form (CNF) if it is expressed as a conjunction of disjunctions of literals (variables or their negations), and it is in Negation Normal Form (NNF) if all negations are applied directly to propositional variables. Any propositional formula can be converted into a semantically equivalent CNF or NNF by recursively applying specific rewriting rules for its logical connectives [10, 51].

2.2 Boolean Networks

A Boolean Network (BN) f is defined as a finite set of Boolean functions over a finite set of Boolean variables, denoted by $\mathsf{Var}(f)$. Each variable $v \in \mathsf{Var}(f)$ is associated with a Boolean function $f_v \colon \mathbb{B}^{|\mathsf{Var}(f)|} \to \mathbb{B}$. A function f_v is termed *constant* if it is always either 0 or 1 regardless of the values of its arguments. A variable v is considered a *source variable* if f_v is the identity function on v, i.e., $f_v = v$. A state s of f is a Boolean vector $s \in \mathbb{B}^{|\mathsf{Var}(f)|}$ that can be viewed as a mapping: $s \colon \mathsf{Var}(f) \to \mathbb{B}$; we denote the value of variable v in state s by s_v. For convenience, a state is often represented as a string of values (e.g., "0110" instead of (0, 1, 1, 0)).

At each discrete time step t, each variable v can update its state according to its Boolean function f_v; that is, v's state at time $t+1$ is given by $s'_v = f_v(s)$. An *update scheme* specifies how these state updates occur over time [58]. The two primary schemes are synchronous, in which all variables update simultaneously, and fully asynchronous, where a single variable is chosen non-deterministically to update. Under arbitrary update scheme, the BN transitions from one state to another – a process known as a *state transition*. The overall dynamics of the BN are captured by the *State Transition Graph* (STG), a directed graph whose nodes represent states and edges represent transitions. We denote the STG under the synchronous update scheme as $\mathsf{sstg}(f)$ and that under the fully asynchronous scheme as $\mathsf{astg}(f)$.

A non-empty set A of states is a *trap set* if there is no transition from a state in A to a state outside A in the State Transition Graph (STG) of f (i.e., there is no pair $x \in A$ and $y \notin A$ such that (x, y) is an arc in the STG) [44]. A trap set that is minimal with respect to set inclusion is termed an *attractor*. In particular, an attractor containing a single state is called a *fixed point*, while one with two or more states is referred to as a *cyclic* attractor. A *sub-space* m of a BN f is a mapping $m \colon \mathsf{Var}(f) \to \mathbb{B}_\star$. A variable $v \in \mathsf{Var}(f)$ is said to be

fixed (resp. *free*) in m if $m(v) \neq \star$ (resp. $m(v) = \star$). For convenience, a sub-space is often represented as a string of values (e.g., $0\star$ instead of $\{v_1 = 0, v_2 = \star\}$). The sub-space m represents a set of states, denoted by $\mathcal{S}[m]$, defined as

$$\mathcal{S}[m] = \{s \in \mathbb{B}^{|\mathsf{Var}(f)|} \mid s_v = m(v), \forall v \in \mathsf{Var}(f), m(v) \neq \star\}$$

For example, if $m = \star 11$, then $\mathcal{S}[m] = \{011, 111\}$. If a sub-space is also a trap set, it is a *trap space*. Unlike trap sets and attractors, trap spaces are independent of the update scheme employed [44]. Notably, a fixed point of f is a special trap space in which all variables are fixed. A trap space m is *minimal* if there is no trap space m' such that $\mathcal{S}[m'] \subset \mathcal{S}[m]$. Since an attractor is a subset-minimal trap set, a minimal trap space contains at least one attractor of the BN, regardless of the update scheme employed [44].

▶ **Example 1.** Let us consider BN f with $\mathsf{Var}(f) = \{a, b\}$, $f_a = a \wedge \neg b$, and $f_b = a$. The synchronous STG of f is shown in Figure 1a. The set $\{00, 01, 11\}$ is a trap set but not a trap space. It is easy to check that f has three trap spaces: $m_1 = 00$, $m_2 = 0\star$, and $m_3 = \star\star$. Among these, m_1 is a minimal trap space (also a fixed point) of f. In this case, m_1 is also the only synchronous attractor of f.

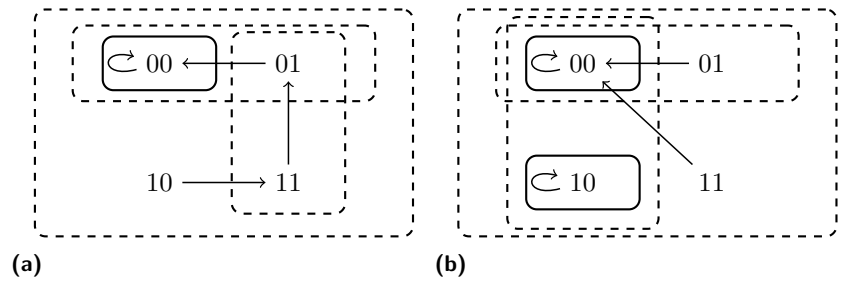


Figure 1 (a) Synchronous STG $\mathsf{sstg}(f)$ of BN f from Example 1. (b) Synchronous STG $\mathsf{sstg}(f)$ where variable b is subject to a *knockout* (i.e., its value is forced to 0). Trap spaces (resp. minimal trap spaces) are enclosed by dashed (resp. solid) rectangular frames.

2.3 Answer Set Programming

An Answer Set Program (ASP) P expresses logical constraints between a set of propositional variables. In answer set programming, such variables are also called *atoms*, and the set of atoms appearing in P is denoted by $\mathsf{at}(P)$. For notational convenience, we will use the terms "variable" and "atom" interchangeably throughout the paper.
An ASP is a finite set of *rules* of the form

$$a_1 \vee \ldots \vee a_k \leftarrow b_1, \ldots, b_m, {\sim}c_1, \ldots, {\sim}c_n$$

where $k, m, n \geq 0$ and $\sim$ denotes *default negation* [17]. Given a rule r of the above form, we define $\mathsf{h}(r) = \{a_1, \ldots, a_n\}$, $\mathsf{b}^+(r) = \{b_1, \ldots, b_m\}$, and $\mathsf{b}^-(r) = \{c_1, \ldots, c_n\}$ as the *head*, *positive body*, and *negative body* of r, respectively. The rule r is a *fact* if $\mathsf{b}^+(r) = \mathsf{b}^-(r) = \emptyset$ and an (integrity) *constraint* if $\mathsf{h}(r) = \emptyset$. We often use the notations $\top$ and $\bot$ to denote empty body ($\mathsf{b}^+(r) \cup \mathsf{b}^-(r) = \emptyset$) and empty head ($\mathsf{h}(r) = \emptyset$), respectively. The rule r is called *normal* if $|\mathsf{h}(r)| \leq 1$, and the ASP program is normal (or a normal logic program) if all its rules are normal. A program is called *disjunctive* if there is a rule $r \in P$ such that $|\mathsf{h}(r)| > 1$ [7].

Answer Set Semantics

A subset M of atoms (called an *interpretation*) *satisfies* a rule r if $(\mathsf{h}(r) \cup \mathsf{b}^-(r)) \cap M \neq \emptyset$ or $\mathsf{b}^+(r) \setminus M \neq \emptyset$. The interpretation M is a *supported model* of P if it satisfies all rules of P, denoted as $M \models P$. The *Gelfond-Lifschitz (GL) reduct* of the program P with respect to the interpretation M is defined as follows: $P^M := \{\mathsf{h}(r) \leftarrow \mathsf{b}^+(r) | r \in P, M \cap \mathsf{b}^-(r) = \emptyset\}$ [30]. The interpretation M is an *answer set* (or a *stable model*) of P if $\nexists\, M' \subsetneq M$ such that $M' \models P^M$. We use the notation $\mathsf{AS}(P)$ to denote the set of answer sets of P.

Answer Set Counting

Given an ASP P, the *answer set counting* problem (denoted as #ASP) seeks to compute the number of answer sets of P, written as $|\mathsf{AS}(P)|$. The projected answer set counting problem (denoted as #PASP) extends #ASP by counting the number of distinct answer sets of P with respect to a given set of *projection atoms* $I \subseteq \mathsf{at}(P)$. Two answer sets are considered equivalent if they differ only on atoms in $\mathsf{at}(P) \setminus I$ [25]. We denote the set of projected answer sets as $\mathsf{AS}(P, I)$, so that #PASP computes $|\{M \cap I \mid M \in \mathsf{AS}(P)\}|$. As a special case, when $I = \mathsf{at}(P)$, #PASP reduces to #ASP. We use the notation $\#\mathsf{PASP}(P, I)$ to denote the *projected answer set count* of program P w.r.t. projection atoms I.

In the probably approximately correct (PAC) framework for answer set counting, given a program P and a projection set I, the goal is to estimate a count cnt satisfying

$$\Pr\left[\frac{|\mathsf{AS}(P,I)|}{(1+\varepsilon)} \leq \mathsf{cnt} \leq (1+\varepsilon) \times |\mathsf{AS}(P,I)|\right] \geq 1-\delta$$

where $0 < \varepsilon < 1$ is the *tolerance* and $0 < \delta < 1$ is the *confidence* [38, 14].

3 Related Work

ASP-Based Computation of Fixed Points and Minimal Trap Spaces

Several ASP encodings have been proposed to characterize fixed points and minimal trap spaces in BNs. In many cases, an ASP encoding for fixed points is derived from its minimal trap space counterpart by adding *integrity constraints* to capture the specific properties of fixed points [69, 72]. The first ASP encoding that requires the computation of *prime implicants* for each Boolean function was proposed and implemented in [44]. A major bottleneck of this encoding is that computing even a single prime implicant is NP-hard and the total number of prime implicants can be exponential in the number of function inputs [15]. Subsequent encodings [55, 69, 72] that were proposed to overcome this bottleneck still suffer from scalability and efficiency issues, particularly for very large and complex models, primarily because they require the *disjunctive normal forms* of all the Boolean functions of the original BN. For fixed points, the ASP encoding by Trinh et al. [71] (called `fASP`) uses a *negation normal form* for each Boolean function, which is much more efficient to obtain. This encoding was later generalized for minimal trap spaces [68] (called `tsconj`); however, when dealing with *unsafe formulas* that might yield incorrect solutions, it still requires a disjunctive normal form to ensure the correctness.

BN Encoding of Normal Logic Programs

The theoretical work by Inoue [33] was among the first to establish a connection between ASP and BNs. It defines a BN encoding for finite ground normal logic programs, which relies on the notion of the Clark's completion [17], and points out that the two-valued models of

the Clark's completion of a finite ground normal logic program one-to-one correspond to the fixed points of the encoded BN. The subsequent work [34] points out that the strict supported classes of a finite ground normal logic program one-to-one correspond to the synchronous attractors of the encoded BN. Very recently, Trinh et al. [73] related the regular models in a finite ground normal logic program and the minimal trap spaces in the respective BN, and further applied these theoretical results to explore graphical conditions for the existence, uniqueness, and number of regular models.

SAT Characterization of Fixed Points

The set of fixed points of a BN f can be characterized as the set of *satisfying assignments* of the propositional formula $\bigwedge_{v \in \mathsf{Var}(f)} (v \leftrightarrow f_v)$ [20]. Hence, we can apply #SAT tools [64, 13, 60] to counting the number of fixed points of a BN. To the best of our knowledge, to date, there is no SAT characterization for the set of minimal trap spaces of a BN.

Answer Set Counting

For general ASP programs, #ASP is $\# \cdot$coNP-complete [27], while #ASP is $\# \cdot$P-complete for normal ASP programs, which follows from standard reductions [35]. The projected answer set counting #PASP is simply Σ_2^p-complete if the set of projection atoms is empty; otherwise, the complexity is $\# \cdot \Sigma_2^p$-complete [25]. Fichte et al. [25, 27] exploited the *tree decomposition*-based technique for counting answer sets. Kabir et al. [38] introduced the hashing-based approximate counting technique for answer set counting.

4 Problem Formulation

In this section, we introduce several counting problems related to minimal trap spaces in Boolean networks (BNs), covering both minimal trap spaces and fixed points. We begin with a straightforward counting variant, then propose a specialized variant that requires solutions to satisfy a specific *property*, and finally present a more complex variant focused on phenotype measurement in BNs under perturbations. To highlight the biological utility of these theoretical problems, a brief case study is also given in Appendix D.

4.1 Counting Minimal Trap Spaces and Fixed Points

We introduce two fundamental counting problems for Boolean networks (BNs): one that counts the number of minimal trap spaces (Definition 2) and another one counts the number of fixed points (Definition 3). These problems address the basic question: How many minimal trap spaces or fixed points does a given BN have?

▶ **Definition 2** (C-MTS-1). *Given a BN f, compute the number of minimal trap spaces of f.*

▶ **Definition 3** (C-FIX-1). *Given a BN f, compute the number of fixed points of f.*

In BN research, both counting problems – C-MTS-1 and C-FIX-1 – are valuable when full enumeration is infeasible, as in gene regulatory network models with many source variables [2, 68, 71]. They are also useful when divergent solutions are sought [16]. Notably, the fixed point counting problem (C-FIX-1) can enhance methods for enumerating asynchronous attractors by enabling the selection of a smaller candidate set, thereby potentially speeding up the filtering process [74]. Finally, both C-MTS-1 and C-FIX-1 can lay the groundwork for probabilistic reasoning in BNs [61].

Recent research has connected Boolean networks (BNs) with two broad fields: abstract argumentation and logic programming. Although we omit the preliminaries on Abstract Argumentation Frameworks (AFs) [18], Abstract Dialectical Frameworks (ADFs) [49], and Normal Logic Programs (NLPs) [73], interested readers can consult the cited papers for details. Specifically, Trinh et al. [70] and Dimopoulos et al. [19] demonstrate that the *preferred* (resp. *stable*) extensions of an AF one-to-one correspond to the minimal trap spaces (resp. fixed points) of the respective BN. Similarly, Heyninck et al. [31] and Azpeitia et al. [5] demonstrate that ADFs and BNs are identical – in this context, the *preferred* (resp. *two-valued*) interpretations of an ADF match the minimal trap spaces (resp. fixed points) of the respective BN. In the realm of logic programming, the subset-minimal *supported partial* models (resp. *supported* models) of a *finite ground* NLP correspond one-to-one with the minimal trap spaces (resp. fixed points) of the respective BN [33, 73]. Moreover, if the finite ground NLP is *tight* [22, 47], then its regular models (resp. stable models) one-to-one correspond to the minimal trap spaces (resp. fixed points) of the respective BN. These types of extensions, interpretations, and models are central to the study of AFs, ADFs, and finite ground NLPs [26, 36, 49].

Recent work on AFs has focused on counting stable and preferred extensions, yielding both complexity results [26] and dynamic programming methods [18] (note that the dynamic programming methods do not support preferred extensions). In contrast, counting problems for ADFs and finite ground NLPs remain largely unexplored, aside from a few studies on answer sets [4, 38]. Thanks to the connections between BNs and these formalisms, these results from C-MTS-1 and C-FIX-1 can be extended to ADFs – where they correspond to preferred and two-valued interpretations – as well as to general finite ground NLPs (for supported partial and supported models) and tight finite ground NLPs (for regular and stable models).

4.2 Counting with Satisfying Properties

We examine a specialized variant of problems C-MTS-1 and C-FIX-1, focusing on counting minimal trap spaces (Definition 4) and fixed points (Definition 5) that satisfy a specified property. This formulation addresses the natural question: How many minimal trap spaces (or fixed points) in a BN exhibit a given property or assumption?

In systems biology, BNs are used to model biological phenotypes, which reflect an organism's functional characteristics [48]. Several definitions of phenotype in BNs have been proposed [9, 45, 46]; in this work, we adopt one of the most widely used notions. Given a BN f, we define *trait* as a statement of the form $(v \leftrightarrow e)$, where $v \in \mathsf{Var}(f)$ and $e \in \mathbb{B}_\star$. Note that $v \leftrightarrow \star$ is evaluated true if and ony if $v = \star$. A phenotype β is then defined as the conjunction of a set of traits. A sub-space m satisfies a phenotype β (denoted by $m \models \beta$) if, upon replacing each variable $v \in \beta$ with its value $m(v)$, the resulting formula evaluates to true under propositional semantics. Unlike minimal trap spaces, fixed points require that all variables take Boolean values ($e \in \mathbb{B}$), and hence phenotypes involving "$\star$" are not meaningful in this context.

In our problem formulation, the property of interest is a desirable phenotype. In systems biology, a minimal trap space satisfying a phenotype suggests the phenotype's potential emergence in vivo. Furthermore, this counting variant is applicable beyond systems biology. For instance, in abstract argumentation frameworks (resp. normal logic programs), a phenotype may represent the presence of particular set of arguments (resp. atoms) in an extension (resp. a model) [26, 36], a concept closely linked to *credulous reasoning*.

▶ **Definition 4** (C-MTS-2). *Given a BN f and a phenotype β, compute the number of minimal trap spaces of f that satisfy β.*

▶ **Definition 5** (C-FIX-2). *Given a BN f and a phenotype β, compute the number of fixed points of f that satisfy β.*

When the specified phenotype is a *tautology*, C-MTS-2 (resp. C-FIX-2) reduces directly to C-MTS-1 (resp. C-FIX-1). Thus, all implications discussed for C-MTS-1 and C-FIX-1 also apply to C-MTS-2 and C-FIX-2, respectively. In the following, we examine these applications through the lens of abstract argumentation frameworks.

First, C-MTS-2 and C-FIX-2 facilitate more nuanced reasoning between *skeptical* and *credulous* approaches by integrating quantitative and probabilistic methods [23]. For instance, by running C-MTS-2 (or C-FIX-2) twice – once with a specific set of arguments and once with a tautology – we can gain insights into preferred (or stable) extensions [26]. Second, this approach shifts reasoning from simple decision-making (i.e., determining whether a set of arguments is present in an extension) to probabilistic reasoning (i.e., assessing the likelihood of a set of arguments appears in an extension). Moreover, this framework supports the development of advanced probabilistic semantics for abstract argumentation frameworks [41]. To our knowledge, these aspects have not yet been explored in the context of BNs, and our work paves the way for integrating such reasoning into BN analysis.

▶ **Example 6.** Consider again the BN f shown in Example 1. It has a unique minimal trap space $m_1 = 00$, which is also its only fixed point of f. Hence, the answer to C-MTS-1 (resp. C-FIX-1) is 1. Considering the phenotype $\beta = (b \leftrightarrow \star)$, the answer to C-MTS-2 is 0.

4.3 Counting Under Perturbations and Measuring Robustness

We consider a main contribution of this paper to be the identification of new, relevant counting problems (other than C-MTS-2 and C-FIX-2, which were previously known): C-MTS-3 and C-FIX-3. To formalize these problems, we introduce the concept of a *perturbation*.

Consider a BN f. A *perturbation* σ [63] on a set $\mathcal{X} \subseteq \mathsf{Var}(f)$ of *perturbable* variables is defined as a mapping from $\mathcal{X} \to \mathbb{B}_\star$. In practice, $\mathcal{X}$ can be any subset of variables whose perturbation is biologically meaningful. Since each variable in $\mathcal{X}$ can assume one of three values under perturbation, there are $3^{|\mathcal{X}|}$ possible perturbations. For each $v \in \mathcal{X}$, setting $\sigma(v) = 0$ forces $f_v = 0$ (*knockout perturbation*), setting $\sigma(v) = 1$ forces $f_v = 1$ (*over-expression perturbation*), and setting $\sigma(v) = \star$ leaves f_v unchanged. Consequently, the perturbed BN, denoted f^σ, is defined by $\mathsf{Var}(f^\sigma) = \mathsf{Var}(f)$ and, for every $v \in \mathsf{Var}(f^\sigma)$,

$$f_v^\sigma = \begin{cases} \sigma(v) & \text{if } v \in \mathcal{X} \text{ and } \sigma(v) \neq \star \\ f_v & \text{otherwise} \end{cases}$$

Note that, the value $\star$ is used to distinguish imperturbable variables and perturbable variables that are unchanged under a certain perturbation.

In biological systems, a perturbation refers to any disturbance that disrupts the normal functioning of a BN. Such disturbances may arise from *genetic mutations* [62], external factors such as *medications* [11], or other influences [52]. These perturbations can substantially alter the phenotypes exhibited by a BN, making it crucial to quantify their effects on network behavior. In systems biology, this impact is commonly assessed in terms of *robustness* [43], which motivates our definitions of problems C-MTS-3 and C-FIX-3.

▶ **Definition 7** (C-MTS-3). *Given a BN f, a set of perturbable variables $\mathcal{X}$, and a target phenotype β, determine the number of perturbations σ on $\mathcal{X}$ such that the perturbed BN f^σ exhibits at least one minimal trap space that satisfies β.*

▶ **Definition 8** (C-FIX-3). *Given a BN f, a set of perturbable variables $\mathcal{X}$, and a target phenotype β, determine the number of perturbations σ on $\mathcal{X}$ such that the perturbed BN f^σ exhibits at least one fixed point that satisfies β.*

Leveraging the results of C-MTS-3 (or C-FIX-3 if focusing solely on fixed points), we define the robustness of a phenotype as the fraction of perturbations that preserve the phenotype relative to the total number of perturbations ($3^{|\mathcal{X}|}$). In other words, robustness measures the probability that a phenotype remains active following a random, admissible perturbation is applied to the network. This measure of phenotype robustness can inform the selection of specific perturbations and guide targeted treatment strategies [9, 65]. A small case study on an interferon model with 121 variables presents these concepts more practically in Appendix D.

▶ **Example 9.** Consider BN f given in Example 1. Consider $\mathcal{X} = \{b\}$ denote the set of perturbable variables and define the set of desirable phenotype as $\beta = (a \leftrightarrow 0 \wedge b \leftrightarrow 0)$. There are three possible perturbations: $\sigma_1 = \{b = 0\}$, $\sigma_2 = \{b = 1\}$, and $\sigma_3 = \{b = \star\}$. The perturbation σ_1 represents the knockout perturbation of variable b and $\mathsf{sstg}(f^{\sigma_1})$ is given in Figure 1b. It is straightforward to observe that f^{σ_1} has two minimal trap spaces 00 and 10. In constrast, the BN f^{σ_2} has one minimal trap space 01, which does not satisfy the given phenotype. The BN f^{σ_3} equals f and has one minimal trap space 00. Therefore, for BN f, phenotype β and perturbable variables $\mathcal{X}$, the answer to C-MTS-3 is 2 and the perturbation robustness of phenotype β in BN f w.r.t. $\mathcal{X}$ is $2/3$.

On the impact of precision. Finally, it should be noted that while these problems are defined *exactly*, in practice their results mainly serve as a means of comparison. For example, the results of C-MTS-2 could be used to compare the abundance of two biological phenotypes. Then the exact count may not be important, as long as the two phenotypes can be compared reliably. As such, these problems are particularly suitable for approximate counting. This also impacts the choice of method parameters (ϵ and δ), as in practice, even low precision (as used in our benchmarks) can be sufficient to distinguish between significantly different phenotypes. For closely matched results, the precision can be then increased as needed.

5 Computational Methods

Given a BN f, our approach is to construct an ASP program P such that the answer sets of P one-to-one correspond to the minimal trap spaces (or fixed points) of f. This reduction allows us to leverage existing answer set counters to efficiently count the answer sets of P. For C-MTS-1 and C-FIX-1, we simply use the ASP encodings of `tsconj` and `fASP`, respectively. For C-MTS-2 and C-FIX-2, we complement the encoding of the given phenotype. For C-MTS-3 and C-FIX-3, we propose a new perturbation encoding, which we consider to be a main contribution of this paper. Now, we begin by briefly reviewing the `tsconj` and `fASP` encodings.

5.1 `tsconj` and `fASP` Encodings

We first present the common components of the two encodings, followed by their differences. The ASP encodings represent sub-spaces using atoms $\mathsf{p}(v)$ and $\mathsf{n}(v)$, indicating whether variable v is fixed to 1, 0, or left free. The goal is to translate BN trap space and fixed point properties into ASP rules whose answer sets correspond to these configurations.

Given a BN f, the encodings compute an ASP program P as follows: for each variable $v \in \mathsf{Var}(f)$, two atoms $\mathsf{p}(v)$ and $\mathsf{n}(v)$ are introduced to indicate positive and negative assignments of the variable v, respectively. Additionally, for every $v \in \mathsf{Var}(f)$, one rule of the form: $\mathsf{p}(v) \vee \mathsf{n}(v) \leftarrow \top$ is added to ensure that each answer set corresponds to a sub-space of f. The translation from an answer set M to a sub-space m is defined as follows: for each $v \in \mathsf{Var}(f)$, we have (i) $m(v) = 1$ if and only if $\mathsf{p}(v) \in M \wedge \mathsf{n}(v) \notin M$, (ii) $m(v) = 0$ if and only if $\mathsf{p}(v) \notin M \wedge \mathsf{n}(v) \in M$, and (iii) $m(v) = \star$ if and only if $\mathsf{p}(v) \in M \wedge \mathsf{n}(v) \in M$. Recall that a trap space of f can be characterized by $\bigwedge_{v \in \mathsf{Var}(f)} (v \leftarrow f_v) \wedge (\neg v \leftarrow \neg f_v)$ [68]. For every $v \in \mathsf{Var}(f)$, two ASP rules – (i) $\gamma(v) \leftarrow \gamma(\mathsf{NNF}(f_v))$ and (ii) $\gamma(\neg v) \leftarrow \gamma(\mathsf{NNF}(\neg f_v))$ – are added to P to express the characterization, where $\mathsf{NNF}(\Phi)$ denotes a negation normal form of a Boolean formula Φ and γ is a procedure defined as follows:

$$\gamma(v) = \mathsf{p}(v), \gamma(\neg v) = \mathsf{n}(v), v \in \mathsf{Var}(f),$$
$$\gamma(\bigwedge_{1 \le j \le J} \alpha_j) = \gamma(\alpha_1) \wedge \ldots \wedge \gamma(\alpha_J), \gamma(\bigvee_{1 \le j \le J} \alpha_j) = \mathsf{aux}_k,$$

where aux_k is a new *auxiliary* atom, k is a global counter starting from 1 and shall be increased by 1 after each new auxiliary atom is created, and for each j, the rule $\mathsf{aux}_k \leftarrow \gamma(\alpha_j)$ is added to P.

For the correctness of the `tsconj` encoding, Trinh et al. [68] defined a syntactic safeness condition for a Boolean formula Φ: Φ is considered safe if it does not contain any conjunction of two subformulas Φ_1 and Φ_2 such that there exists a variable x appearing in Φ_1 with $\neg x$ appearing in Φ_2. When both f_v and $\neg f_v$ are safe for every $v \in \mathsf{Var}(f)$, then the set of answer sets of P one-to-one corresponds with the set of minimal trap spaces of f. When a Boolean formula Φ (f_v or $\neg f_v$) is unsafe, the Disjunctive Normal Form (DNF) of Φ is used instead, which is always safe by definition.

There is no notion of "safeness" in the `fASP` encoding. Rather an additional rule $\bot \leftarrow \mathsf{p}(v), \mathsf{n}(v)$ is added to P, for every $v \in \mathsf{Var}(f)$ such that the variable v cannot take the value of $\star$. The answer sets of P one-to-one correspond to the fixed points of f.

Following on, given a BN f, we use the notations $\mathsf{P\text{-}tsconj}(f)$ and $\mathsf{P\text{-}fASP}(f)$ to denote the encoded ASP programs of f, according to the `tsconj` and `fASP` encodings, respectively.

5.2 Methods for Problems C-MTS-1 and C-FIX-1

We make use of `tsconj` and `fASP` encodings for C-MTS-1 and C-FIX-1, respectively. The choice of encodings is due to following two reasons: first, these encodings rely on less expensive representations – specifically, negation normal forms (ref. Section 3). Second, they establish a one-to-one correspondence between the minimal trap spaces (or fixed points) of the original BN and the answer sets of the encoded ASP program, whereas other encodings yield a one-to-one correspondence with the subset-minimal (or subset-maximal) answer sets [3], which prevents the direct use of existing ASP counters.

5.3 Methods for Problems C-MTS-2 and C-FIX-2

We add the encoding of phenotype to the encodings of `tsconj` and `fASP` to solve counting problems C-MTS-2 and C-FIX-2, respectively. Given a BN f and a phenotype β, we compute an ASP program $\mathsf{ToASP}(\beta)$ to capture the phenotype β by invoking Algorithm 1. The algorithm exploits atoms introduced in the `tsconj` encoding to interpret the phenotype β. The main idea of Algorithm 1 is exploiting *faceted answer set navigation* to constrain the search space of answer sets [24]. We prove that the minimal trap spaces of f satisfying the phenotype β one-to-one correspond to the answer sets of $\mathsf{P\text{-}tsconj}(f) \cup \mathsf{ToASP}(\beta)$ (Theorem 10).

Algorithm 1 ToASP(β).

Input: Phenotype β
Output: ASP program Q

$Q \leftarrow \emptyset$
for each $(v \leftrightarrow e) \in \beta$ **do**
  if $e = 1$ **then**
    Q.add($\bot \leftarrow \sim\mathsf{p}(v)$, $\bot \leftarrow \mathsf{n}(v)$)
  else if $e = 0$ **then**
    Q.add($\bot \leftarrow \mathsf{p}(v)$, $\bot \leftarrow \sim\mathsf{n}(v)$)
  else if $e = \star$ **then**
    Q.add($\bot \leftarrow \sim\mathsf{p}(v)$, $\bot \leftarrow \sim\mathsf{n}(v)$)
return Q

▶ **Theorem 10.** *Given a BN f and a phenotype β, the minimal trap spaces of f satisfying β one-to-one correspond to the answer sets of* P-tsconj(f) ∪ ToASP(β).

For C-FIX-2, we can apply the `fASP` encoding and Algorithm 1 similarly. Note that in the counting problem C-FIX-2, the term e is either 0 or 1, for each $(v \leftrightarrow e) \in \beta$, since fixed points require all variables to be fixed. We formally prove the correctness of our proposed method for C-FIX-2 in Theorem 11.

▶ **Theorem 11.** *Given a BN f and a phenotype β, the fixed points of f satisfying β one-to-one correspond to the answer sets of* P-fASP(f) ∪ ToASP(β).

▶ **Example 12.** Consider the BN f of Example 1. The program P-tsconj(f) is as follows:

$\mathsf{p}(a) \vee \mathsf{n}(a) \leftarrow \top$ $\quad \mathsf{p}(a) \leftarrow \mathsf{p}(a), \mathsf{n}(b)$ $\quad \mathsf{n}(a) \leftarrow \mathsf{aux}_1$ $\quad \mathsf{aux}_1 \leftarrow \mathsf{n}(a)$ $\quad \mathsf{aux}_1 \leftarrow \mathsf{p}(b)$
$\mathsf{p}(b) \vee \mathsf{n}(b) \leftarrow \top$ $\quad \mathsf{p}(b) \leftarrow \mathsf{p}(a)$ $\quad \mathsf{n}(b) \leftarrow \mathsf{n}(a)$

The program P-tsconj(f) has a unique answer set $\{\mathsf{n}(a), \mathsf{n}(b), \mathsf{aux}_k\}$ corresponding to the unique minimal trap space 00 of f. Consider the phenotype $\beta = (b \leftrightarrow \star)$, the BN f has no minimal trap space satisfying β (see Example 6), thus C-MTS-2 returns 0. Following the procedure outlined above, the ASP program ToASP(β) is as follows:

$\bot \leftarrow \sim\mathsf{p}(b)$ $\quad \bot \leftarrow \sim\mathsf{n}(b)$

Indeed, the program P-tsconj(f) ∪ ToASP(β) has no answer set.

5.4 Methods for Problems C-MTS-3 and C-FIX-3

Given a BN f, a phenotype β, and a set of perturbable variables $\mathcal{X}$, one possible approach to solving these problems is to introduce new atoms to represent possible perturbations over perturbable variables, then correspondingly to intervene in the encoded ASP program obtained by applying the encodings proposed for C-MTS-2 and C-FIX-2, and finally to apply projected counting restricted to these new atoms. Instead, we propose a more convenient approach that reduces C-MTS-3 (resp. C-FIX-3) to C-MTS-2 (resp. C-FIX-2) along with projected answer set counting.

▶ **Definition 13.** *Consider a BN f and a set of perturbable variables $\mathcal{X} \subseteq \mathsf{Var}(f)$, we construct a new BN g such that for every $v \in \mathsf{Var}(f)$, if $v \in \mathsf{Var}(f) \setminus \mathcal{X}$, then the variable $v \in \mathsf{Var}(g)$ and $g_v = f_v$, and if $v \in \mathcal{X}$, then three variables $v, v^k, v^o \in \mathsf{Var}(g)$ and*

$$g_v = \neg v^k \wedge (v^o \vee f_v),$$
$$g_{v^k} = v^k,$$
$$g_{v^o} = v^o \wedge \neg v^k.$$

Instead of modifying the ASP encoding to model perturbations, we construct a perturbed version of the original BN, thereby preserving the semantics of the original encodings. For each perturbable variable $v \in \mathcal{X}$, we introduce two new variables in g: v^k, encoding whether v is knocked out, and v^o, encoding whether it is over-expressed. By construction:

- If $v^k = 1$, then $v^o = 0$ and $v = 0$, modeling a knockout of v.
- If $v^k = 0$ and $v^o = 1$, then $v = 1$, modeling over-expression.
- If $v^k = 0$ and $v^o = 0$, then v follows its original update function f_v (v is unperturbed).
- The case $v^k = 1$ and $v^o = 1$ is infeasible due to the constraint $g_{v^o} = v^o \wedge \neg v^k$.

Since $\mathsf{Var}(f) \subseteq \mathsf{Var}(g)$, any phenotype β defined over f remains valid in g. The minimal trap spaces (resp. fixed points) of g correspond to those of f under all possible perturbations over $\mathcal{X}$. Thus, the minimal trap spaces (resp. fixed points) of g that satisfy the phenotype β represent the perturbed solutions of f that also satisfy β. Crucially, for each perturbation, multiple satisfying minimal trap spaces (or fixed points) are counted once, enabling us to count the number of satisfying perturbations. Hence, the counting problem C-MTS-3 (resp. C-FIX-3) reduces to the projected version of C-MTS-2 (resp. C-FIX-2) over the newly defined BN g.

We now discuss how we solve C-MTS-3 and C-FIX-3 by applying projected answer set counting. We focus on the case for minimal trap spaces, and the case of fixed points is trivially similar. Following Theorem 10, the set of answer sets of $\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta)$ represents the set of minimal trap spaces of g satisfying the phenotype β. Let $\Omega = \bigcup_{v \in \Delta} \{\mathsf{p}(v), \mathsf{n}(v)\}$ denote the set of perturbation-related variables, where $\Delta = \bigcup_{v \in \mathcal{X}} \{v^k, v^o\}$ be the set of ASP atoms used in the encoding. It follows that the number of answer sets of $\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta)$, projected onto the set Ω is equal to the number of perturbation settings (i.e., assignments to variables in $\mathcal{X}$) under which g admits a minimal trap space satisfying β.

▶ **Theorem 14.** *Given a BN f, a set of perturbable variables $\mathcal{X} \subseteq \mathsf{Var}(f)$, a phenotype β, and $\Omega = \bigcup_{v \in \Delta} \{\mathsf{p}(v), \mathsf{n}(v)\}$, where $\Delta = \bigcup_{v \in \mathcal{X}} \{v^k, v^o\}$, then C-MTS-3 can be computed as $\#\mathsf{PASP}(\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta), \Omega)$, where g is the new BN according to Definition 13.*

▶ **Theorem 15.** *Given a BN f, a set of perturbable variables $\mathcal{X} \subseteq \mathsf{Var}(f)$, a phenotype β, and $\Omega = \bigcup_{v \in \Delta} \{\mathsf{p}(v), \mathsf{n}(v)\}$, where $\Delta = \bigcup_{v \in \mathcal{X}} \{v^k, v^o\}$, then C-FIX-3 can be computed as $\#\mathsf{PASP}(\mathsf{P\text{-}fASP}(g) \cup \mathsf{ToASP}(\beta), \Omega)$, where g is the new BN following Definition 13.*

▶ **Example 16.** Consider again Example 9. Following Definition 13, we obtain the BN g: $\mathsf{Var}(g) = \{a, b, b^k, b^o\}$, $g_a = a \wedge \neg b$, $g_{b^k} = b^k$, $g_{b^o} = b^o \wedge \neg b^k$, and $g_b = \neg b^k \wedge (b^o \vee a)$. (see Appendix B for the details of $\mathsf{P\text{-}tsconj}(g)$ and $\mathsf{ToASP}(\beta)$). Let $\Omega = \{\mathsf{p}(b^k), \mathsf{n}(b^k), \mathsf{p}(b^o), \mathsf{n}(b^o)\}$. Then C-MTS-3 can be computed as $\#\mathsf{PASP}(\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta), \Omega)$. Indeed, $\#\mathsf{PASP}(\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta), \Omega)$ returns 2, which is consistent with the result shown in Example 9.

6 Experimental Evaluation

This section presents the experimental evaluation of the presented methods. We use existing minimal trap space and fixed point computation tools as baselines – namely, AEON [8], k++ADF (ADF) [49], and clingo [29]. The ADF tool is applicable here due to the equivalence

between ADFs and BNs [5, 31], which supports only C-MTS-1 and C-FIX-1. k++ADF and Clingo count minimal trap spaces and fixed points via enumeration on the ADF and the encoded ASP respectively, and AEON via BDD-based encoding. For the fixed point problem variants, we further considered the propositional model counter [60] (GANAK) and the approximate model counter [77] (ApproxMC), using the translation to CNF (see Section 3). However, note that these techniques cannot be directly used for minimal trap space counting. For approximate answer set counting, we employed the hashing-based approximate answer set counter ApproxASP [38] with parameters $\varepsilon = 0.8$ and $\delta = 0.2$. Following prior work on counting [40], we provided ApproxASP with an independent support of a disjunctive ASP program exploiting Padoa theorem [53]. We also tested the tree decomposition-based answer set counter DynASP [27]; however, we did not include it in the final analysis because it has been significantly outperformed by the remaining baselines. Finally, note that we could not include #SAT-based ASP counters aspmc [21] and sharpASP [37] as baselines, since these counters are designed for normal logic programs, while `tsconj` and `fASP` encodings produce disjunctive ASP programs.

We compiled our benchmark set from prior studies on minimal trap spaces and fixed points in BNs [54, 68, 71]. The set comprises 645 BN instances – 245 real-world models and 400 randomly generated – with up to 5,000 variables. To evaluate counting problems C-MTS-2, C-FIX-2, C-MTS-3, and C-FIX-3, we pseudo-randomly fixed three variables to represent the target phenotype and selected up to 50 perturbable variables (yielding as many as 3^{50} possible perturbations). Phenotypes are typically not published in machine-readable format, thus we maintain biological interpretability by deriving the phenotype from a known trap space, linking it to an existing biological feature of the network. As for the chosen size, only a few variables are sufficient to identify phenotypes: e.g., in [28], only 10-200 entities are needed out of 10,000. Since our tests are often significantly smaller, we scaled down the phenotypes accordingly. Appendix C provides further details about the benchmark. The code and dataset of experiment evaluation is available at: `https://zenodo.org/records/15141045`

Environmental Settings. All experiments were conducted on a high-performance computing cluster, with each node consisting of Intel Xeon Gold 6248 CPUs. Each benchmark instance was allocated one core, with runtime and memory limits set to 5000 seconds and 8 GB respectively, for all the tools considered.

6.1 Experimental Results

C-MTS-1 and C-FIX-1. The results for C-MTS-1 and C-FIX-1 are shown in Table 1 and Table 2, respectively. Each table reports the number of instances solved (i.e., instances for which a count was successfully returned) by each tool and their corresponding PAR2 scores [6] (PAR2 score is a runtime metric that also penalizes benchmark timeouts). Here, approximate counting (ApproxASP and ApproxMC) clearly outperform all existing solutions. Even compared to exact model counting (GANAK), this approach achieves significantly better performance. Note that for C-FIX-1, ApproxASP and ApproxMC are roughly comparable, but (as discussed in detail later), ApproxASP is faster on simpler instances.

It is worth noting that unsafe formulas are quite rare in 245 real-world models, which is consistent with the observation in [68]. All 400 randomly generated models have no unsafe formulas because of the nature of the generation [68].

C-MTS-2 and C-FIX-2. The results for C-MTS-2 and C-FIX-2 are shown in Table 3 and Table 4, respectively, using the same metric (solved instances and PAR2 score) as Table 1 and Table 2. Here, the relative performance of individual tools mirrors that observed for

Table 1 The performance comparison of different counters on C-MTS-1 counting problem.

	AEON	ADF	clingo	ApproxASP
#Solved	179	200	211	**364**
PAR2	7255	6923	6742	**4448**

Table 2 The performance comparison of different counters on C-FIX-1 counting problem.

	AEON	ADF	clingo	GANAK	ApproxMC	ApproxASP
#Solved	247	217	227	317	**420**	413
PAR2	6172	6656	6493	5269	3801	**3760**

C-MTS-1 and C-FIX-1. However, all tools solved more instances overall, largely due to the inclusion of the phenotype property, which typically reduces the number of solutions. Since phenotype properties are generally much simpler than the update functions describing network dynamics, we expect them to simplify the counting problem – often substantially.

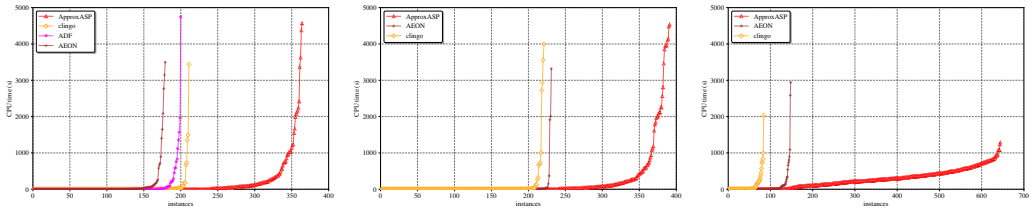
Table 3 The performance comparison of different counters on C-MTS-2 counting problem.

	AEON	clingo	ApproxASP
#Solved	231	308	**464**
PAR2	6428	5237	**2919**

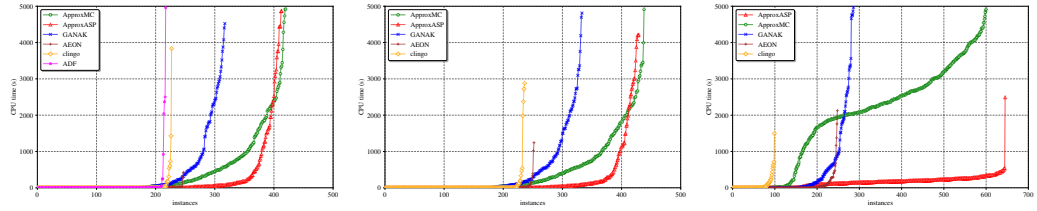
Table 4 The performance comparison of different counters on C-FIX-2 counting problem.

	AEON	clingo	GANAK	ApproxMC	ApproxASP
#Solved	252	236	333	**438**	429
PAR2	6099	6360	5030	3527	**3499**

C-MTS-3 and C-FIX-3. The results for problems C-MTS-3 and C-FIX-3 are presented in Tables 5 and 6, respectively. This benchmark confirms the leading performance of ApproxASP in minimal trap space and fixed point counting as it was able to solve 644/645 and 645/645 problem instances for C-MTS-3 and C-FIX-3, respectively. Here, ApproxASP significantly outperforms even ApproxMC, and outperforms all other tools by a factor of 2× or more. In contrast to C-MTS-2 and C-FIX-2, where all tools benefited from a reduced number of solutions, the presence of perturbations in this setting generally increases both the number of solutions and the underlying complexity of BNs. The key performance differentiation lies in the tools' ability to handle *projected counting*. For this problem, the *independent support* for XOR constraints is derived from the BN perturbable variables. Since this is only a subset of the network variables, the independent support size is relatively small, reducing the size of the XORs. This results in the superior performance of ApproxASP and ApproxMC. Note that in these countings, the number of perturbable variables is at most 50 and the count is upper-bounded by 3^{50}.



(a) C-MTS-1. **(b)** C-MTS-2. **(c)** C-MTS-3.



(d) C-FIX-1. **(e)** C-FIX-2. **(f)** C-FIX-3.

Figure 2 Performance comparison of different counters across all counting problems. The x axis shows the number of benchmarks completed before the corresponding CPU runtime on the y axis.

Table 5 The performance comparison of different counters on C-MTS-3 counting problem.

	AEON	clingo	ApproxASP
#Solved	148	84	**644**
PAR2	7725	8711	**283**

Table 6 The performance comparison of different counters on C-FIX-3 counting problem.

	AEON	clingo	GANAK	ApproxMC	ApproxASP
#Solved	248	99	286	600	**645**
PAR2	6176	8476	5757	2481	**150**

6.2 Detailed Runtime Analysis

The runtime performance of different tools is depicted in Figure 2. In these cactus plots, a point (x, y) indicates that a tool successfully completes x benchmark instances, with each instance taking at most y seconds. The plots highlight the superiority of the hashing-based counting techniques, ApproxASP and ApproxMC. Notably, even in cases where ApproxASP solves fewer instances than ApproxMC (e.g. C-FIX-1), it is typically faster on simpler problem instances, which is also reflected in its PAR2 score.

By examining the number of solutions successfully computed for different tasks, we observe that only ApproxASP, ApproxMC, and GANAK can reliably count instances having a large number of solutions (e.g. $\geq 10^{30}$). Here, BDD-based counters like AEON perform somewhat better on fixed point problems compared to tools using plain enumeration (ADF, clingo), but cannot compete in the (arguably more complex) trap space problems.

The performance of GANAK and ApproxMC is also severely affected by the time required to compute their input CNF formulas (ref. Section 3). Here, deriving the CNF problem representation is often considerably more time-consuming than computing the comparable

ASP encoding. Our results reveal that, on average, it took about 545 seconds to compute the CNF formula for each BN. Moreover, for 39 BNs, the corresponding CNF could not be computed within the 5000-second timeout. In contrast, the ASP encodings for all BNs – including the more challenging ones with perturbable variables – were generated within seconds. This demonstrates the superior flexibility of the ASP-based approach.

Note that the approximate counters ApproxMC and ApproxASP provide an (ε, δ)-guarantee. Thus for each solved instance, we computed the *observed tolerance*, which is defined as $\mathsf{max}(\mathsf{cnt}/|\mathsf{AS}(P)|, |\mathsf{AS}(P)|/\mathsf{cnt}) - 1$, where cnt is the count returned by ApproxASP or ApproxMC, and $\mathsf{AS}(P)$ denotes the answer set count of program P. On average, ApproxMC and ApproxASP exhibit observed tolerances of 0.032 and 0.007, respectively. The maximum observed tolerances were 0.39 for ApproxASP and 0.07 for ApproxMC, both of which are well below the theoretical bound of $\varepsilon = 0.8$.

We evaluated ApproxMC and ApproxASP with a tighter guarantee, setting $\varepsilon = 0.01$ and $\delta = 0.05$. In these setting, ApproxASP solved $244, 243, 114, 258, 259, 130$ for C-MTS-1, C-MTS-2, C-MTS-3, C-FIX-1, C-FIX-2, and C-FIX-3, respectively, while ApproxMC solved $263, 259, 120$ for C-FIX-1, C-FIX-2, and C-FIX-3, respectively. Consequently, the higher precision significantly reduces the number of solved instances.

We compared the size of instances solved by different counting techniques across various counting problems. We observed that, for minimal trap spaces, ApproxASP solved instances with up to 4000 variables, while `tsconj` and AEON solved instances with up to 321 variables. For fixed points, although there were large instances (up to 4000 variables), these did not contain fixed points, making a direct comparison unfeasible.

7 Conclusion

This paper addresses the problem of counting minimal trap spaces and fixed points in Boolean networks. These are critical concepts in understanding of long-term BN behavior and are relevant across a diverse set of application domains, including systems biology, abstract argumentation, and logic programming. Trap space counting is especially important in systems biology: due to the inherent robustness of biological phenomena, biologically motivated BNs admit a high degree of redundancy, resulting in a vast repertoire of closely related trap spaces (or fixed points) that cannot be explored solely through enumeration.

Here, we propose novel methods for determining trap space and fixed point counts using approximate answer set counting, thus entirely avoiding costly enumeration. We apply this methodology to three biologically motivated problems: (a) general counting; (b) counting occurrences of a known biological phenotype, and (c) projected counting of perturbations that ensure the emergence of a known biological phenotype. The last problem is particularly timely, as it allows us to determine *perturbation robustness* [9, 43], a vital measure that determines how stable a phenotype appears under external stimuli. Through extensive experiments on a diverse set of benchmarks, we show that approximate counting substantially improves the feasibility of counting in this domain, outperforming traditional enumeration-based and exact approaches whenever applicable.

Our work opens several promising directions for future research: First is to integrate reduction techniques – previously shown to be effective in BN analysis [56, 59, 65] – to further improve counting accuracy and scalability. Second direction explores hybrid strategies that combine exact and approximate counting, aiming to strike a balance between efficiency and precision. Finally, a deeper investigation into the computational complexity of the counting problems would help refine our understanding of their theoretical underpinnings.

References

1 Emna Ben Abdallah, Maxime Folschette, Olivier F. Roux, and Morgan Magnin. ASP-based method for the enumeration of attractors in non-deterministic synchronous and asynchronous multi-valued networks. *Algorithms Mol. Biol.*, 12(1):20:1–20:23, 2017. doi:10.1186/S13015-017-0111-2.

2 Sara Sadat Aghamiri, Vidisha Singh, Aurélien Naldi, Tomás Helikar, Sylvain Soliman, Anna Niarakis, and Jinbo Xu. Automated inference of Boolean models from molecular interaction maps using CaSQ. *Bioinform.*, 36(16):4473–4482, 2020. doi:10.1093/BIOINFORMATICS/BTAA484.

3 Mario Alviano, Carmine Dodaro, Salvatore Fiorentino, Alessandro Previti, and Francesco Ricca. ASP and subset minimality: Enumeration, cautious reasoning and MUSes. *Artif. Intell.*, 320:103931, 2023. doi:10.1016/J.ARTINT.2023.103931.

4 Rehan Abdul Aziz, Geoffrey Chu, Christian J. Muise, and Peter James Stuckey. Stable model counting and its application in probabilistic logic programming. In *AAAI*, pages 3468–3474. AAAI Press, 2015. doi:10.1609/AAAI.V29I1.9691.

5 Eugenio Azpeitia, Stan Muñoz Gutiérrez, David A. Rosenblueth, and Octavio Zapata. Bridging abstract dialectical argumentation and Boolean gene regulation. *CoRR*, abs/2407.06106, 2024. doi:10.48550/arXiv.2407.06106.

6 Tomas Balyo, Marijn JH Heule, and Matti Järvisalo. SAT competition 2017–solver and benchmark descriptions, 2017.

7 Rachel Ben-Eliyahu and Rina Dechter. Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell.*, 12(1-2):53–87, 1994. doi:10.1007/BF01530761.

8 Nikola Benes, Lubos Brim, Jakub Kadlecaj, Samuel Pastva, and David Safránek. AEON: attractor bifurcation analysis of parametrised Boolean networks. In *CAV*, pages 569–581. Springer, 2020. doi:10.1007/978-3-030-53288-8_28.

9 Nikola Benes, Lubos Brim, Samuel Pastva, David Safránek, and Eva Smijáková. Phenotype control of partially specified Boolean networks. In *CMSB*, pages 18–35. Springer, 2023. doi:10.1007/978-3-031-42697-1_2.

10 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

11 Peter Bloomingdale, Van Anh Nguyen, Jin Niu, and Donald E Mager. Boolean network modeling in systems pharmacology. *J. Pharmacokinet. Pharmacodyn.*, 45:159–180, 2018.

12 Florian Bridoux, Amélia Durbec, Kévin Perrot, and Adrien Richard. Complexity of fixed point counting problems in Boolean networks. *J. Comput. Syst. Sci.*, 126:138–164, 2022. doi:10.1016/J.JCSS.2022.01.004.

13 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In Christian Schulte, editor, *CP*, pages 200–216. Springer, 2013. doi:10.1007/978-3-642-40627-0_18.

14 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *IJCAI*, pages 3569–3576. IJCAI/AAAI Press, 2016. URL: http://www.ijcai.org/Abstract/16/503.

15 Ashok K. Chandra and George Markowsky. On the number of prime implicants. *Discrete Math.*, 24(1):7–11, 1978. doi:10.1016/0012-365X(78)90168-1.

16 Stéphanie Chevalier, Vincent Noël, Laurence Calzone, Andrei Yu. Zinovyev, and Loïc Paulevé. Synthesis and simulation of ensembles of Boolean networks for cell fate decision. In *CMSB*, pages 193–209. Springer, 2020. doi:10.1007/978-3-030-60327-4_11.

17 Keith L. Clark. *Negation as Failure*, pages 293–322. Springer US, 1978. doi:10.1007/978-1-4684-3384-5_11.

18 Ridhwan Dewoprabowo, Johannes Klaus Fichte, Piotr Jerzy Gorczyca, and Markus Hecher. A practical account into counting Dung's extensions by dynamic programming. In *LPNMR*, pages 387–400. Springer, 2022. doi:10.1007/978-3-031-15707-3_30.

19 Yannis Dimopoulos, Wolfgang Dvorák, and Matthias König. Connecting abstract argumentation and Boolean networks. In *COMMA*, pages 85–96. IOS Press, 2024. doi:10.3233/FAIA240312.

20 Elena Dubrova and Maxim Teslenko. A SAT-based algorithm for finding attractors in synchronous Boolean networks. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 8(5):1393–1399, 2011. doi:10.1109/TCBB.2010.20.

21 Thomas Eiter, Markus Hecher, and Rafael Kiesel. aspmc: New frontiers of algebraic answer set counting. *Artif. Intell.*, 330:104109, 2024. doi:10.1016/J.ARTINT.2024.104109.

22 François Fages. Consistency of Clark's completion and existence of stable models. *Methods Log. Comput. Sci.*, 1(1):51–60, 1994.

23 Johannes K. Fichte, Markus Hecher, and Mohamed A. Nadeem. Plausibility reasoning via projected answer set counting - a hybrid approach. In *IJCAI*, volume 22, pages 2620–2626, 2022. doi:10.24963/IJCAI.2022/363.

24 Johannes Klaus Fichte, Sarah Alice Gaggl, and Dominik Rusovac. Rushing and strolling among answer sets - navigation made easy. In *AAAI*, pages 5651–5659. AAAI Press, 2022. doi:10.1609/AAAI.V36I5.20506.

25 Johannes Klaus Fichte and Markus Hecher. Treewidth and counting projected answer sets. In *LPNMR*, pages 105–119. Springer, 2019. doi:10.1007/978-3-030-20528-7_9.

26 Johannes Klaus Fichte, Markus Hecher, and Arne Meier. Counting complexity for reasoning in abstract argumentation. *J. Artif. Intell. Res.*, 80, 2024. doi:10.1613/JAIR.1.16210.

27 Johannes Klaus Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Answer set solving with bounded treewidth revisited. In *LPNMR*, pages 132–145. Springer, 2017. doi:10.1007/978-3-319-61660-5_13.

28 Stephan Fischer and Jesse Gillis. How many markers are needed to robustly determine a cell's type? *iScience*, 24(11):103292, November 2021.

29 Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The Potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011. doi:10.3233/AIC-2011-0491.

30 Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP*, pages 1070–1080. MIT Press, 1988.

31 Jesse Heyninck, Matthias Knorr, and João Leite. Abstract dialectical frameworks are Boolean networks. In *LPNMR*, pages 98–111. Springer, 2024. doi:10.1007/978-3-031-74209-5_8.

32 Christopher M. Homan and Sven Kosub. Dichotomy results for fixed point counting in Boolean dynamical systems. *Theor. Comput. Sci.*, 573:16–25, 2015. doi:10.1016/J.TCS.2015.01.040.

33 Katsumi Inoue. Logic programming for Boolean networks. In *IJCAI*, pages 924–930. IJCAI/AAAI, 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-160.

34 Katsumi Inoue and Chiaki Sakama. Oscillating behavior of logic programs. In *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, pages 345–362. Springer, 2012. doi:10.1007/978-3-642-30743-0_23.

35 Tomi Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *J. Appl. Non Class. Logics*, 16(1-2):35–86, 2006. doi:10.3166/JANCL.16.35-86.

36 Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding partiality and disjunctions in stable model semantics. *ACM TOCL*, 7(1):1–37, 2006. doi:10.1145/1119439.1119440.

37 Mohimenul Kabir, Supratik Chakraborty, and Kuldeep S. Meel. Exact ASP counting with compact encodings. In *AAAI*, pages 10571–10580. AAAI Press, 2024. doi:10.1609/AAAI.V38I9.28927.

38 Mohimenul Kabir, Flavio O. Everardo, Ankit K. Shukla, Markus Hecher, Johannes Klaus Fichte, and Kuldeep S. Meel. ApproxASP - a scalable approximate answer set counter. In *AAAI*, pages 5755–5764. AAAI Press, 2022. doi:10.1609/AAAI.V36I5.20518.

39 Mohimenul Kabir and Kuldeep S Meel. A fast and accurate ASP counting based network reliability estimator. In *LPAR*, volume 94, pages 270–287, 2023. doi:10.29007/KC6Q.

40 Mohimenul Kabir and Kuldeep S Meel. On lower bounding minimal model count. *TPLP*, 24(4):586–605, July 2024. doi:10.1017/S147106842400036X.

41 Nikolai Käfer, Christel Baier, Martin Diller, Clemens Dubslaff, Sarah Alice Gaggl, and Holger Hermanns. Admissibility in probabilistic argumentation. *J. Artif. Intell. Res.*, 74, 2022. doi:10.1613/JAIR.1.13530.

42 Roland Kaminski, Torsten Schaub, Anne Siegel, and Santiago Videla. Minimal intervention strategies in logical signaling networks with ASP. *TPLP*, 13(4-5):675–690, 2013. doi:10.1017/S1471068413000422.

43 Hiroaki Kitano. Towards a theory of biological robustness. *Mol. Syst. Biol.*, 3(1):137, 2007.

44 Hannes Klarner, Alexander Bockmayr, and Heike Siebert. Computing maximal and minimal trap spaces of Boolean networks. *Nat. Comput.*, 14(4):535–544, 2015. doi:10.1007/S11047-015-9520-7.

45 Hannes Klarner, Frederike Heinitz, Sarah Nee, and Heike Siebert. Basins of attraction, commitment sets, and phenotypes of Boolean networks. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 17(4):1115–1124, 2020. doi:10.1109/TCBB.2018.2879097.

46 Hannes Klarner, Elisa Tonello, Laura Fontanals, Florence Janody, Claudine Chaouiya, and Heike Siebert. Detection of markers for discrete phenotypes. In *CSBio*, pages 64–68, 2021.

47 Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In *ICLP*, pages 451–465. Springer, 2003. doi:10.1007/978-3-540-24599-5_31.

48 Qin Li, Anders Wennborg, Erik Aurell, Erez Dekel, Jie-Zhi Zou, Yuting Xu, Sui Huang, and Ingemar Ernberg. Dynamics inside the cancer cell attractor reveal cell heterogeneity, limits of stability, and escape. *Proc. Natl. Acad. Sci. U.S.A.*, 113(10):2672–2677, February 2016.

49 Thomas Linsbichler, Marco Maratea, Andreas Niskanen, Johannes Peter Wallner, and Stefan Woltran. Advanced algorithms for abstract dialectical frameworks based on complexity analysis of subclasses and SAT solving. *Artif. Intell.*, 307:103697, 2022. doi:10.1016/J.ARTINT.2022.103697.

50 Victor W Marek and Miroslaw Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm*, pages 375–398. Springer, 1999. doi:10.1007/978-3-642-60085-2_17.

51 Gabriele Masina, Giuseppe Spallitta, and Roberto Sebastiani. On CNF conversion for disjoint SAT enumeration. In *SAT*, pages 15:1–15:16, 2023. doi:10.4230/LIPICS.SAT.2023.15.

52 Arnau Montagud, Jonas Béal, Luis Tobalina, Pauline Traynard, Vigneshwari Subramanian, Bence Szalai, Róbert Alföldi, László Puskás, Alfonso Valencia, Emmanuel Barillot, et al. Patient-specific Boolean models of signalling networks guide personalised treatments. *Elife*, 11:e72626, 2022.

53 Alessandro Padoa. Essai d'une théorie algébrique des nombres entiers, précédé d'une introduction logique à une théorie déductive quelconque. *Bibliothèque du Congrès International de Philosophie*, 3:309–365, 1901.

54 Samuel Pastva, David Šafránek, Nikola Beneš, Luboš Brim, and Thomas Henzinger. Repository of logically consistent real-world Boolean network models. *bioRxiv*, 2023. URL: https://www.biorxiv.org/content/early/2023/06/12/2023.06.12.544361.

55 Loïc Paulevé, Juraj Kolčák, Thomas Chatain, and Stefan Haar. Reconciling qualitative, abstract, and scalable modeling of biological networks. *Nat. Commun.*, 11(1):1–7, August 2020.

56 Jordan C. Rozum, Jorge Gómez Tejeda Zañudo, Xiao Gan, Dávid Deritei, and Réka Albert. Parity and time reversal elucidate both decision-making in empirical models and attractor scaling in critical Boolean networks. *Sci. Adv.*, 7(29):eabf8124, July 2021.

57 Torsten Schaub and Sven Thiele. Metabolic network expansion with answer set programming. In *ICLP*, pages 312–326. Springer, 2009. doi:10.1007/978-3-642-02846-5_27.

58 Julian D Schwab, Silke D Kühlwein, Nensi Ikonomi, Michael Kühl, and Hans A Kestler. Concepts in Boolean network modeling: What do they all mean? *Comput. Struct. Biotechnol. J.*, 18:571–582, 2020.

59 Robert Schwieger and Elisa Tonello. Reduction for asynchronous Boolean networks: elimination of negatively autoregulated components. *Discret. Math. Theor. Comput. Sci.*, 25(2), 2023. doi:10.46298/DMTCS.10930.

60 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In *IJCAI*, pages 1169–1176. ijcai.org, 2019. doi:10.24963/IJCAI.2019/163.

61 Ilya Shmulevich, Edward R. Dougherty, and Wei Zhang. From Boolean to probabilistic Boolean networks as models of genetic regulatory networks. *Proc. IEEE*, 90(11):1778–1792, 2002. doi:10.1109/JPROC.2002.804686.

62 Ilya Shmulevich, Edward R Dougherty, and Wei Zhang. Gene perturbation and intervention in probabilistic Boolean networks. *Bioinf.*, 18(10):1319–1331, 2002.

63 Cui Su and Jun Pang. Sequential temporary and permanent control of Boolean networks. In *CMSB*, pages 234–251. Springer, 2020. doi:10.1007/978-3-030-60327-4_13.

64 Marc Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. In *SAT*, pages 424–429. Springer, 2006. doi:10.1007/11814948_38.

65 Elisa Tonello and Loïc Paulevé. Phenotype control and elimination of variables in Boolean networks. *Peer Community Journal*, 4, August 2024.

66 Predrag T. Tosic. On the complexity of counting fixed points and gardens of eden in sequential dynamical systems on planar bipartite graphs. *Int. J. Found. Comput. Sci.*, 17(5):1179–1204, 2006. doi:10.1142/S0129054106004339.

67 Predrag T. Tosic and Gul A. Agha. On computational complexity of counting fixed points in symmetric Boolean graph automata. In *UC*, pages 191–205. Springer, 2005. doi:10.1007/11560319_18.

68 Van-Giang Trinh, Belaid Benhamou, Samuel Pastva, and Sylvain Soliman. Scalable enumeration of trap spaces in Boolean networks via answer set programming. In *AAAI*, pages 10714–10722. AAAI Press, 2024. doi:10.1609/AAAI.V38I9.28943.

69 Van-Giang Trinh, Belaid Benhamou, and Loïc Paulevé. mpbn: a simple tool for efficient edition and analysis of elementary properties of Boolean networks. *CoRR*, abs/2403.06255, 2024. doi:10.48550/arXiv.2403.06255.

70 Van-Giang Trinh, Belaid Benhamou, and Vincent Risch. Graphical analysis of abstract argumentation frameworks via Boolean networks. In *ICAART*, pages 745–756, 2025. doi:10.5220/0013346400003890.

71 Van-Giang Trinh, Belaid Benhamou, and Sylvain Soliman. Efficient enumeration of fixed points in complex Boolean networks using answer set programming. In *CP*, pages 35:1–35:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CP.2023.35.

72 Van-Giang Trinh, Belaid Benhamou, and Sylvain Soliman. Trap spaces of Boolean networks are conflict-free siphons of their Petri net encoding. *Theor. Comput. Sci.*, 971:114073, September 2023. doi:10.1016/J.TCS.2023.114073.

73 Van-Giang Trinh, Belaid Benhamou, Sylvain Soliman, and François Fages. Graphical conditions for the existence, unicity and number of regular models. In *ICLP*, pages 175–187, 2024. doi:10.4204/EPTCS.416.16.

74 Van-Giang Trinh, Kunihiko Hiraishi, and Belaid Benhamou. Computing attractors of large-scale asynchronous Boolean networks using minimal trap spaces. In *ACM-BCB*, pages 13:1–13:10. ACM, 2022. doi:10.1145/3535508.3545520.

75 Van-Giang Trinh, Kyu Hyong Park, Samuel Pastva, and Jordan C Rozum. Mapping the attractor landscape of Boolean networks with biobalm. *Bioinformatics*, 41(5):btaf280, 2025.

76 Santiago Videla, Carito Guziolowski, Federica Eduati, Sven Thiele, Martin Gebser, Jacques Nicolas, Julio Saez-Rodriguez, Torsten Schaub, and Anne Siegel. Learning Boolean logic models of signaling networks with ASP. *Theor. Comput. Sci.*, 599:79–101, 2015. doi:10.1016/J.TCS.2014.06.022.

77 Jiong Yang and Kuldeep S Meel. Rounding meets approximate model counting. In *CAV*, pages 132–162. Springer, 2023. doi:10.1007/978-3-031-37703-7_7.

A Details of Proofs

▶ **Theorem 10.** *Given a BN f and a phenotype β, the minimal trap spaces of f satisfying β one-to-one correspond to the answer sets of* $\mathsf{P\text{-}tsconj}(f) \cup \mathsf{ToASP}(\beta)$.

Proof. To prove the correctness of Theorem 10, we reuse the correctness proof of Theorem 2 of [68], which establishes that the answer sets of $\mathsf{P\text{-}tsconj}(f)$ one-to-one correspond to the minimal trap spaces of f. Let us recall the translation between an answer set M of $\mathsf{P\text{-}tsconj}(f)$ and its respective minimal trap space m of f: for each variable $v \in \mathsf{Var}(f)$, $m(v) = 1$ if and only if $\mathsf{p}(v) \in M \wedge \mathsf{n}(v) \notin M$, $m(v) = 0$ if and only if $\mathsf{p}(v) \notin M \wedge \mathsf{n}(v) \in M$, and $m(v) = \star$ if and only if $\mathsf{p}(v) \in M \wedge \mathsf{n}(v) \in M$.

We employ the theories of faceted answer set navigation [24] to prove the correctness of Theorem 10. According to faceted navigation, for a program P and an atom $a \in \mathsf{at}(P)$, adding the integrity constraint $\{\bot \leftarrow a\}$ to program P restricts the search space of P, where no answer set contains the atom a. Conversely, adding the integrity constraint $\{\bot \leftarrow \sim a\}$ to program P ensures that every answer set in the modified program contains the atom a.

We prove the correctness of our encoding for each trait $(v \leftrightarrow e) \in \beta$. We show that Algorithm 1 selects *facets* in such a way that the answer sets of $\mathsf{P\text{-}tsconj}(f) \cup \mathsf{ToASP}(\beta)$ one-to-one correspond to the minimal trap spaces satisfying β of f. When $(v \leftrightarrow 1) \in \beta$ (line 4 in Algorithm 1), two constraints are added to $\mathsf{ToASP}(\beta)$ to ensure that every answer set contains the atom $\mathsf{p}(v)$ and none contains the atom $\mathsf{n}(v)$. These constraints effectively assign the value 1 to variable v. When $(v \leftrightarrow 0) \in \beta$ (line 6 in Algorithm 1), two added constraints ensure that every answer set contains the atom $\mathsf{n}(v)$ and none contains $\mathsf{p}(v)$, thereby assigning the value 0 to variable v. When $(v \leftrightarrow \star) \in \beta$ (line 8 in Algorithm 1), two added constraints ensure that every answer set contains both $\mathsf{p}(v)$ and $\mathsf{n}(v)$. These constraints effectively assign the value $\star$ to variable v.
Combining all the cases of Algorithm 1, we can claim the correctness of our encoding. ◀

▶ **Theorem 11.** *Given a BN f and a phenotype β, the fixed points of f satisfying β one-to-one correspond to the answer sets of* $\mathsf{P\text{-}fASP}(f) \cup \mathsf{ToASP}(\beta)$.

Proof. The proof technique of Theorem 10 can be similarly extended for fixed point counting with the program $\mathsf{P\text{-}fASP}(f)$. ◀

To prove Theorems 14 and 15, we prepare the following preliminaries.

▶ **Definition 17.** *The total order $\leq_t$ on $\mathbb{B}_\star$ is defined by $0 <_t \star <_t 1$.*

▶ **Definition 18.** *The partial order $\leq_s$ on $\mathbb{B}_\star$ is defined by $0 <_s \star$, $1 <_s \star$, and it contains no other relation.*

▶ **Definition 19.** *Consider a BN f, a sub-space m of f, and a Boolean expression e over $\mathsf{Var}(f)$. The value of e under sub-space m w.r.t. the Kleene three-valued logic, denoted as $m(e)$, is recursively defined as follows:*

$$m(e) = \begin{cases} e & \text{if } e \in \mathbb{B}_\star \\ m(a) & \text{if } e = a, a \in \mathsf{Var}(f) \\ \neg m(e_1) & \text{if } e = \neg e_1 \\ min_{\leq_t}(m(e_1), m(e_2)) & \text{if } e = e_1 \wedge e_2 \\ max_{\leq_t}(m(e_1), m(e_2)) & \text{if } e = e_1 \vee e_2 \end{cases}$$

where $\neg 1 = 0, \neg 0 = 1, \neg\star = \star$, *and* $min_{\leq_t}$ *(resp.* $max_{\leq_t}$*) is the function to get the minimum (resp. maximum) value of two values w.r.t. the order* $\leq_t$.

▶ **Theorem 20** (Theorem 1 of [44])**.** *Consider a BN* f *and a sub-space* m *of* f*. A sub-space* m *is a trap space of* f *iff* $m(f_v) \leq_s m(v)$ *for every* $v \in \mathsf{Var}(f)$.

▶ **Corollary 21.** *Consider a BN* f *and a sub-space* m *of* f*. A sub-space* m *is a minimal trap space of* f *iff* m *is a* $\leq_s$*-minimal trap space of* f.

▶ **Corollary 22.** *Consider a BN* f *and a sub-space* m *of* f*. A sub-space* m *is a fixed point of* f *iff* m *is a trap space of* f *and* $m(v) \neq \star$ *for every* $v \in \mathsf{Var}(f)$.

Now, we show the formal proof of Theorem 14.

▶ **Lemma 23.** *Consider a BN* f *and a set of perturbable variables* $\mathcal{X} \subseteq \mathsf{Var}(f)$*. Let* g *be the BN obtained from* f*, according to Definition 13. Let* Δ *be the set* $\bigcup_{v\in\mathcal{X}}\{v^k, v^o\}$*. If* m *is a minimal trap space of* g*, then* $m(v) \neq \star$ *for every* $v \in \Delta$.

Proof. Let m be a minimal trap space of g. Assume that there exists $v^k \in \Delta$ such that $m(v^k) = \star$ or $v^o \in \Delta$ such that $m(v^o) = \star$. We consider two cases as follows.

Case 1: $m(v^k) = \star$**.** Let m' be a sub-space of g such that $m'(u) = m(u)$, for every $u \in \mathsf{Var}(g) \setminus \{v^k\}$ and $m'(v^k) = 0$. We have $m'(g_{v^k}) = m'(v^k)$. The variable v^k only affects v^o and v. Regarding v^o, we have $m'(g_{v^o}) = m'(v^o \wedge \neg v^k) = \min_{\leq_t}(m'(v^o), \neg m'(v^k)) = \min_{\leq_t}(m'(v^o), 1) = m'(v^o)$. Regarding v, we have $m'(g_v) = m'(\neg v^k \wedge (v^o \vee f_v)) = \min_{\leq_t}(\neg m'(v^k), \max_{\leq_t}(m'(v^o), m'(f_v))) = \min_{\leq_t}(1, \max_{\leq_t}(m'(v^o), m'(f_v))) = \max_{\leq_t}(m'(v^o), m'(f_v)) = \max_{\leq_t}(m(v^o), m(f_v))$. Following Theorem 20, $m(g_v) \leq_s m(v)$. We have $m(g_v) = m(\neg v^k \wedge (v^o \vee f_v)) = \min_{\leq_t}(\neg m(v^k), \max_{\leq_t}(m(v^o), m(f_v))) = \min_{\leq_t}(\star, \max_{\leq_t}(m(v^o), m(f_v)))$. Since $m(v) <_s \max_{\leq_t}(m(v^o), m(f_v))$ implies $m(v) <_s m(g_v)$ which is a contradiction, we derive that $\max_{\leq_t}(m(v^o), m(f_v)) \leq_s m(v)$. This implies $m'(g_v) \leq_s m(v) = m'(v)$. For any $u \in \mathsf{Var}(g) \setminus \{v, v^k, v^o\}$, $m'(g_u) = m(g_u) \leq_s m(u) = m'(u)$. Hence, m' is trap space of g and $m' <_s m$, which is a contradiction.

Case 2: $m(v^o) = \star$**.** Let m' be a sub-space of g such that $m'(u) = m(u)$ for every $u \in \mathsf{Var}(g) \setminus \{v^o\}$ and $m'(v^o) = 0$. We have $m'(g_{v^o}) = m'(v^o \wedge \neg v^k) = \min_{\leq_t}(m'(v^o), \neg m'(v^k)) = \min_{\leq_t}(0, \neg m'(v^k)) = 0 = m'(v^o)$. The variable v^o only affects v. We have $m'(g_v) = m'(\neg v^k \wedge (v^o \vee f_v)) = \min_{\leq_t}(\neg m'(v^k), \max_{\leq_t}(m'(v^o), m'(f_v))) = \min_{\leq_t}(\neg m'(v^k), \max_{\leq_t}(0, m'(f_v))) = \min_{\leq_t}(\neg m'(v^k), m'(f_v)) = \min_{\leq_t}(\neg m(v^k), m(f_v))$. Following Theorem 20, $m(g_v) \leq_s m(v)$. We have $m(g_v) = m(\neg v^k \wedge (v^o \vee f_v)) = \min_{\leq_t}(\neg m(v^k), \max_{\leq_t}(m(v^o), m(f_v))) = \min_{\leq_t}(\neg m(v^k), \max_{\leq_t}(\star, m(f_v)))$. Suppose that $m'(v) = m(v) <_s \min_{\leq_t}(\neg m(v^k), m(f_v))$. If $m(f_v) = 0$, then $m(v) <_s 0$ which contradicts to the definition of $\leq_s$. Hence $m(f_v) \neq 0$, leading to $\max_{\leq_t}(\star, m(f_v)) = m(f_v)$. Then $m(v) <_s \min_{\leq_t}(\neg m(v^k), m(f_v)) = m(g_v)$, which is a contradiction. Hence, $\min_{\leq_t}(\neg m(v^k), m(f_v)) \leq_s m'(v)$. This implies that $m'(g_v) \leq_s m'(v)$. For any $u \in \mathsf{Var}(g) \setminus \{v, v^o\}$, $m'(g_u) = m(g_u) \leq_s m(u) = m'(u)$. Hence, m' is trap space of g and $m' <_s m$, which is a contradiction.

Combining Case 1 and Case 2, we can conclude that for $m(v) \neq \star$ for every $v \in \Delta$. ◀

▶ **Lemma 24.** *Consider a BN f and a set of perturbable variables $\mathcal{X} \subseteq \mathsf{Var}(f)$. Let g be the BN obtained from f, according to Definition 13 and Δ be the set $\bigcup_{v\in\mathcal{X}}\{v^k, v^o\}$. Let m be a trap space of g such that $m(v) \neq \star$ for every $v \in \Delta$. Let σ be the perturbation of f such that $\sigma(v) = 1$ if and only if $m(v^k) = 0$ and $m(v^o) = 1$, $\sigma(v) = 0$ if and only if $m(v^k) = 1$ and $m(v^o) = 0$, and $\sigma(v) = \star$ if and only if $m(v^k) = 0$ and $m(v^o) = 0$. Then m' is a trap space of f^σ where $m'(v) = m(v)$ for every $v \in \mathsf{Var}(f)$.*

Proof. Assume that there exists $v \in \mathcal{X}$ such that $m(v^k) = m(v^o) = 1$. Since m is a trap space of g, $m(g_{v^o}) \leq_s m(v^o)$. We have $m(g_{v^o}) = m(v^o \wedge \neg v^k) = 0$, whereas $m(v^o) = 1$, leading to $0 \leq_s 1$, which contradicts to the definition of $\leq_s$. Hence, the case of $m(v^k) = m(v^o) = 1$ is impossible for every $v \in \mathcal{X}$. Since $m(v) \neq \star$ for every $v \in \Delta$, σ is well specified.

Recall that $\mathsf{Var}(g) = \mathsf{Var}(f) \cup \Delta$ and $\mathsf{Var}(f) = \mathsf{Var}(f^\sigma)$. Consider $v \in \mathsf{Var}(f^\sigma)$. If $v \notin \mathcal{X}$, we have $m'(f_v^\sigma) = m'(f_v) = m'(g_v) = m(g_v) \leq_s m(v) = m'(v)$. If $v \in \mathcal{X}$, we have the following cases:

Case 1: $m(v^k) = 0$ and $m(v^o) = 0$. Then $\sigma(v) = \star$, thus $m'(f_v^\sigma) = m'(f_v)$. We have $m(g_v) = m(\neg v^k \wedge (v^o \vee f_v)) = m(f_v) \leq_s m(v) = m'(v)$. Since $m'(f_v) = m(f_v)$, it follows that $m'(f_v^\sigma) \leq_s m'(v)$.

Case 2: $m(v^k) = 1$ and $m(v^o) = 0$. Then $\sigma(v) = 0$, thus $m'(f_v^\sigma) = 0$. We have $m(g_v) = m(\neg v^k \wedge (v^o \vee f_v)) = 0 \leq_s m(v) = m'(v)$. Hence, $m'(f_v^\sigma) \leq_s m'(v)$.

Case 3: $m(v^k) = 0$ and $m(v^o) = 1$. Then $\sigma(v) = 1$, thus $m'(f_v^\sigma) = 1$. We have $m(g_v) = m(\neg v^k \wedge (v^o \vee f_v)) = 1 \leq_s m(v) = m'(v)$. Hence, $m'(f_v^\sigma) \leq_s m'(v)$.

Now we can conclude that $m'(f_v^\sigma) \leq_s m'(v)$ for every $v \in \mathsf{Var}(f^\sigma)$. Hence, m' is a trap space of f^σ. ◀

▶ **Theorem 14.** *Given a BN f, a set of perturbable variables $\mathcal{X} \subseteq \mathsf{Var}(f)$, a phenotype β, and $\Omega = \bigcup_{v\in\Delta}\{\mathsf{p}(v), \mathsf{n}(v)\}$, where $\Delta = \bigcup_{v\in\mathcal{X}}\{v^k, v^o\}$, then C-MTS-3 can be computed as $\#\mathsf{PASP}(\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta), \Omega)$, where g is the new BN according to Definition 13.*

Proof. First, we consider a perturbation $\sigma\colon \mathcal{X} \to \mathbb{B}_\star$ of BN f. Let $m_\Delta\colon \Delta \to \mathbb{B}$ be a mapping such that for every $v \in \mathcal{X}$, $\sigma(v) = 1$ if and only if $m_\Delta(v^k) = 0$ and $m_\Delta(v^o) = 1$, $\sigma(v) = 0$ if and only if $m_\Delta(v^k) = 1$ and $m_\Delta(v^o) = 0$, and $\sigma(v) = \star$ if and only if $m_\Delta(v^k) = 0$ and $m_\Delta(v^o) = 0$. Recall that $\mathsf{Var}(g) = \mathsf{Var}(f) \cup \Delta$ and $\mathsf{Var}(f) = \mathsf{Var}(f^\sigma)$. Let m be a minimal trap space of f^σ and $m \models \beta$. Let m' be a sub-space of g such that $m'(v) = m(v)$ if $v \in \mathsf{Var}(f)$ and $m'(v) = m_\Delta(v)$ if $v \in \Delta$. We show that m' is a minimal trap space of g and $m' \models \beta$ (1).

Consider $v \in \mathcal{X}$, we have $m'(g_{v^k}) = m'(v^k)$. If $m'(v^k) = 0$, then $m'(v^o \wedge \neg v^k) = m'(v^o)$. If $m'(v^k) = 1$, then $m'(v^o) = 0$ due to the definition of m_Δ, leading to $m'(v^o \wedge \neg v^k) = 0 = m'(v^o)$. Hence, we can derive that $m'(g_{v^o}) = m'(v^o \wedge \neg v^k) = m'(v^o)$. Regarding $m'(g_v) = m'(\neg v^k \wedge (v^o \vee f_v))$, we have the following cases:

Case 1: $\sigma(v) = \star$. Then $m'(v^k) = 0$ and $m'(v^o) = 0$. We have $m'(g_v) = m'(f_v) = m(f_v) = m(f_v^\sigma) \leq_s m(v) = m'(v)$.

Case 2: $\sigma(v) = 1$. Then $m'(v^k) = 0$ and $m'(v^o) = 1$. We have $m'(g_v) = 1 = m(f_v^\sigma) \leq_s m(v) = m'(v)$.

Case 3: $\sigma(v) = 0$. Then $m'(v^k) = 1$ and $m'(v^o) = 0$. We have $m'(g_v) = 0 = m(f_v^\sigma) \leq_s m(v) = m'(v)$. Consider $v \in \mathsf{Var}(f) \setminus \mathcal{X}$. We have $m'(g_v) = m'(f_v) = m(f_v) = m(f_v^\sigma) \leq_s m(v) = m'(v)$.

Now, we can conclude that m' is a trap space of g. Assume that m' is not minimal. Then there is a trap space n of g such that $n <_s m'$. Since $m'(v) \neq \star$ for every $v \in \Delta$, $n(v) = m'(v)$ for every $v \in \Delta$, leading to $n(v) \neq \star$ for every $v \in \Delta$. Following the Lemma 24, n' is a trap

space of f^σ where $n'(v) = n(v)$ for every $v \in \mathsf{Var}(f)$. Since $n(v) = m'(v)$ for every $v \in \Delta$, we have $n' <_s m$, which contradicts to the $\leq_s$-minimality of m w.r.t. f^σ. Hence, m' is a minimal trap space of g. In addition, since β only contains the variables in $\mathsf{Var}(f)$, it is trivial that $m' \models \beta$.

Second, we consider a minimal trap space m of g such that $m \models \beta$. By Lemma 23, $m(v) \neq \star$ for every $v \in \Delta$. The case of $m(v^k) = m(v^o) = 1$ is impossible because if it holds, then $m(g_{v^o}) = m(v^o \wedge \neg v^k) = 0 \leq_s m(v^o) = 1$, which is a contradiction. Let σ be the perturbation of f such that $\sigma(v) = 1$ if and only if $m(v^k) = 0$ and $m(v^o) = 1$, $\sigma(v) = 0$ if and only if $m(v^k) = 1$ and $m(v^o) = 0$, and $\sigma(v) = \star$ if and only if $m(v^k) = 0$ and $m(v^o) = 0$. Let m' be a sub-space of f such that $m'(v) = m(v)$ for every $v \in \mathsf{Var}(f)$. We show that m' is a minimal trap space of f^σ and $m' \models \beta$ (2).

By Lemma 24, m' is a trap space of f^σ. Assume that m' is not minimal. Then there is a minimal trap space n of f^σ such that $n <_s m'$. Let n' be a sub-space of g such that $n'(v) = m(v)$ for every $v \in \Delta$ and $n'(v) = n(v)$ for every $v \in \mathsf{Var}(f)$. By following the same reasoning for (1), we have n' is a trap space of g. However, $n' <_s m$, which contradicts to the $\leq_s$-minimality of m w.r.t. g. Hence, m' is a minimal trap space of f^σ. In addition, since β only contains the variables in $\mathsf{Var}(f)$, it is trivial that $m' \models \beta$.

From (1) and (2), we can conclude that, given $f, \mathcal{X}$, and β, the result of the counting problem C-MTS-3 is equivalent to the number of minimal trap spaces of g that satisfy β where multiple minimal trap spaces with the same values on the variables in Δ are only counted once. By Theorem 10, the answer sets of $\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta)$ one-to-one correspond to the minimal trap spaces of g satisfying β. The set Ω includes the atoms of $\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta)$ corresponding to the variables in Δ of g. It follows that the number of answer sets of $\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta)$ projected to Ω is equal to the number of minimal trap spaces of g satisfying β projected to Δ. This implies that C-MTS-3 can be computed as the projected answer set counting query $\#\mathsf{PASP}(\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta), \Omega)$. ◀

Finally, we show the formal proof of Theorem 15.

▶ **Theorem 15.** *Given a BN f, a set of perturbable variables $\mathcal{X} \subseteq \mathsf{Var}(f)$, a phenotype β, and $\Omega = \bigcup_{v \in \Delta} \{\mathsf{p}(v), \mathsf{n}(v)\}$, where $\Delta = \bigcup_{v \in \mathcal{X}} \{v^k, v^o\}$, then C-FIX-3 can be computed as $\#\mathsf{PASP}(\mathsf{P\text{-}fASP}(g) \cup \mathsf{ToASP}(\beta), \Omega)$, where g is the new BN following Definition 13.*

Proof. The proof technique of Theorem 14 can be similarly extended for C-FIX-3 with the program $\mathsf{P\text{-}fASP}(g)$. ◀

B Details of Example ASP Programs

The ASP program $\mathsf{P\text{-}tsconj}(f)$ considered in Example 16 is:

$\mathsf{p}(a) \vee \mathsf{n}(a) \leftarrow \top$ $\quad \mathsf{p}(a) \leftarrow \mathsf{p}(a), \mathsf{n}(b)$ $\quad \mathsf{n}(a) \leftarrow \mathsf{aux}_1$ $\quad \mathsf{aux}_1 \leftarrow \mathsf{n}(a)$ $\quad \mathsf{aux}_1 \leftarrow \mathsf{p}(b)$

$\mathsf{p}(b) \vee \mathsf{n}(b) \leftarrow \top$

$\mathsf{p}(b) \leftarrow \mathsf{n}(b^k), \mathsf{aux}_2$ $\quad \mathsf{aux}_2 \leftarrow \mathsf{p}(b^o)$ $\quad \mathsf{aux}_2 \leftarrow \mathsf{p}(a)$

$\mathsf{n}(b) \leftarrow \mathsf{aux}_3$ $\quad \mathsf{aux}_3 \leftarrow \mathsf{p}(b^k)$ $\quad \mathsf{aux}_3 \leftarrow \mathsf{n}(b^o), \mathsf{n}(a)$

$\mathsf{p}(b^k) \vee \mathsf{n}(b^k) \leftarrow \top$ $\quad \mathsf{p}(b^k) \leftarrow \mathsf{p}(b^k)$ $\quad \mathsf{n}(b^k) \leftarrow \mathsf{n}(b^k)$

$\mathsf{p}(b^o) \vee \mathsf{n}(b^o) \leftarrow \top$ $\quad \mathsf{p}(b^o) \leftarrow \mathsf{p}(b^o), \mathsf{n}(b^k)$ $\quad \mathsf{n}(b^o) \leftarrow \mathsf{aux}_4$ $\quad \mathsf{aux}_4 \leftarrow \mathsf{n}(b^o)$ $\quad \mathsf{aux}_4 \leftarrow \mathsf{p}(b^k)$

The ASP program $\mathsf{ToASP}(\beta)$ considered in Example 16 is:

$$\bot \leftarrow \mathsf{p}(a) \qquad \bot \leftarrow \sim\mathsf{n}(a)$$
$$\bot \leftarrow \mathsf{p}(b) \qquad \bot \leftarrow \sim\mathsf{n}(b)$$

C Details of Benchmark Instances

Our benchmarks are based on the 245 real-world biological models from the BBM dataset [54] available at the time of writing, plus 400 random Boolean networks that were first tested as part of [68]. The real-world Boolean models range up to 1076 variables, out of which up to 223 are source variables. However, the median network size in this dataset is less than 100 variables. As such, we also consider larger, random Boolean networks ranging between 1,000 and 5,000 variables. All source variables across all networks are left unrestricted, meaning they can take the value 0 or 1, thereby maximizing the number of admissible trap spaces.

These 645 instances are used directly as inputs for benchmarking the C-FIX-1 and C-MTS-1 problems. To test C-FIX-2 and C-MTS-2, we augment each network with a pseudo-random phenotype specification. Here, the specification is chosen as follows: we first compute an arbitrary, fixed minimal trap space using `tsconj`. We then randomly select the values of three fixed variables – excluding all the source variables of the network. The conjunction of these values represents the tested phenotype. This process ensures that for each Boolean network, problem C-MTS-2 always has at least one valid minimal trap space solution (existence of a fixed point solution cannot be guaranteed regardless of the chosen phenotype).

Finally, to evaluate C-FIX-3 and C-MTS-3, we use the same phenotype but also pseudo-randomly select up to 50 perturbable variables, excluding both the source variables and those fixed by the phenotype. For networks with less than 50 such candidates, we simply select all viable variables as perturbable. We then use the transformation proposed in Definition 13 to construct a new variant of each benchmark network in which the selected variables can be perturbed. Each such perturbed network, together with the phenotype, represents the input for C-FIX-3 and C-MTS-3.

D Phenotype robustness analysis, case study of Interferon 1 model

Model no. 118 in the BBM dataset [54] represents an interaction network related to the activation of the so-called *Interferon 1*, a biochemical species closely tied to immune response present in T-cells. The model was initially derived using [2] and then later tuned by domain experts to correctly capture relevant biological phenotypes. It consists of 121 variables, of which 55 are "inputs", meaning they are not regulated by other variables.

The model defines three phenotype variables, *ISG* (full name *ISG expression antiviral response phenotype*), *PCK* (full name *Proinflammatory cytokine expression inflammation phenotype*), and *IFN* (full name *Type 1 IFN response phenotype*). As these are separate output variables of the network, each trap space can exhibit any combination of active and inactive phenotype variables.

Since the model defines response of T-cells to immunological stimuli and environmental factors, it is important to understand how these mechanisms respond to potential permanent perturbations, either due to genetic mutations or therapeutic treatments. Here, we provide an overview of the model phenotypes through the lens of phenotype robustness.

Table 7 Phenotype robustness analysis for the Interferon 1 model. First three columns describe the desired phenotype (− meaning the value is unconstrained). The C-MTS-3 column lists the number of perturbations for which said phenotype appears in the network. Finally, robustness r indicates what portion of the possible perturbations still enable the corresponding phenotype.

ISG	PCK	IFN	C-MTS-3	Robustness (r)
1	–	–	3486784401	1.000
–	1	–	2114072298	0.606
–	–	1	2313362673	0.663
0	0	0	478296900	0.137
1	0	0	621785970	0.178
0	1	0	478296900	0.137
0	0	1	813104730	0.233
1	1	0	782989740	0.224
1	0	1	1096362783	0.314
0	1	1	813104730	0.233
1	1	1	1409735826	0.404

For simplicity, we selected 20 variables of the model as potential perturbation targets. In reality, this choice would be further influenced by known genetic risk factors or drug targets. Consequently, this choice results in $3^{20} = 3486784401$ admissible perturbations in our Boolean system. We then consider two phenotype variants: First, where a single phenotype variable is expected to be 1 and the remaining phenotype variables are unconstrained (i.e. they can be 0, 1, or $\star$), yielding three combinations. And second, more specific one, where each phenotype variable is fixed to either 1 or 0, yielding eight combinations.

The results for C-MTS-3 and the subsequent robustness computation are given in Table 7. Here, we can notice several biologically interesting outcomes:

- Regardless of perturbation, the network always exhibits a trap space with the *ISG* phenotype. The remaining phenotypes are usually also present ($r = 0.606$ and $r = 0.663$), but are significantly less robust than *ISG*. This indicates that (assuming favorable environmental conditions), expression of ISG as a response to viral activity is robust and cannot be disrupted by a perturbation.
- Among the fully defined phenotypes, 010 and 000 are the least robust while 111 is the most robust. This shows the general tendency of T-cells to reliably and consistently respond to immunological stimuli, as the 111 phenotype indicates the maximal level of immunological activity.
- Even though *ISG* is the most robust phenotype when taken in isolation, phenotypes with active *IFN* generally achieve higher robustness when other phenotypes are required to be inactive.

Such outcomes serve several functions: First, they can be used to validate (or refute) assumptions about the biological tendencies of the studied system. Second, if observations do not match model predictions, better quantitative understanding can provide possible sources of further model refinement. Third, this knowledge can enable us to better (more reliably or efficiently) select a phenotype that should be targeted by a treatment among several related but distinct cellular phenotypes. Finally, note that the number of solutions in each case is significantly higher than what would be countable using standard enumeration, underscoring the importance of dedicated counting methods, and approximate counting in particular.

Greed Is Slow on Sparse Graphs of Oriented Valued Constraints

Artem Kaznatcheev
Department of Mathematics, and Department of Information and Computing Sciences, Utrecht University, The Netherlands

Sofia Vazquez Alferez
Department of Mathematics, and Department of Information and Computing Sciences, Utrecht University, The Netherlands

Abstract

Greedy local search is especially popular for solving valued constraint satisfaction problems (VCSPs). Since any method will be slow for some VCSPs, we ask: what is the simplest VCSP on which greedy local search is slow? We construct a VCSP on $6n$ Boolean variables for which greedy local search takes $7(2^n - 1)$ steps to find the unique peak. Our VCSP is simple in two ways. First, it is very sparse: its constraint graph has pathwidth 2 and maximum degree 3. This is the simplest VCSP on which some local search could be slow. Second, it is "oriented" – there is an ordering on the variables such that later variables are conditionally-independent of earlier ones. Being oriented allows many non-greedy local search methods to find the unique peak in a quadratic number of steps. Thus, we conclude that – among local search methods – greed is particularly slow.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases valued constraint satisfaction problem, local search, algorithm analysis, constraint graphs

Digital Object Identifier 10.4230/LIPIcs.CP.2025.18

Related Version *Full Version*: https://arxiv.org/abs/2506.11662

Acknowledgements We want to thank Martin C. Cooper for his comments which greatly improved the quality of our writing and Melle van Marle for his insightful discussions.

1 Introduction

We cannot hope for a polynomial time algorithm to solve arbitrary combinatorial optimization problems. But we still need to try to do something to solve these hard problems. Given that many hard problems can be defined as finding an assignment $x \in \{0,1\}^n$ that maximizes some pseudo-Boolean function – that we call a *fitness function* – many turn to local search as a heuristic method [1]. Local search methods start at an initial assignment, then apply a fixed update rule to select a better adjacent assignment to move to until no further improvement is possible. Such a sequence of adjacent improving assignments is called an ascent. One of the most popular update rules is to always select the adjacent assignment that increases fitness by the largest amount, that is, to follow the steepest ascent– this update rule defines greedy local search [22]. Since greed is just one of many possible update rules, a natural question arises: is greed good? Or more specifically: is greedy local search fast or slow compared to other local search methods?

Since no local search method will be able to solve *all* instances of hard combinatorial optimization problems in a polynomial number of steps, we need a more nuanced criterium for what makes a local search method fast or slow. We find this nuance in looking at the performance of our methods on subsets of instances of differing simplicity instead of on all instances. Now, if a method cannot solve particularly simple sets of instances in a polynomial number of steps – at least when compared to other similar methods – then we call it slow.

© Artem Kaznatcheev and Sofia Vazquez Alferez;
licensed under Creative Commons License CC-BY 4.0

To define our set of all instances and refine that to simple instances, we turn to valued constraint satisfaction problems (VCSPs). In the language of constraint programming, finding the maximum of a pseudo-Boolean function is the same as solving a Boolean VCSP. Given that even binary Boolean VCSPs can express both NP-hard and PLS-complete problems [3, 11], we define – in Section 2 – our set of all instances as the set of all binary Boolean VCSPs. Each binary VCSP can be interpreted as defining a constraint graph with an edge between any two variables that share a constraint. This allows us to use the sparsity of the resulting graphs as a measure of simplicity: we focus on sets of instances of bounded degree and of bounded treewidth or – more restrictively – pathwidth.

In terms of degree, Elsässer and Tscheuschner [8] showed that binary Boolean VCSPs are PLS-complete under tight reductions even if the constraints are restricted to MAXCUT. The tight PLS-reduction means that there are degree 5 instances where all ascents are exponential – including the steepest ascent taken by greedy local search. Monien and Tscheuschner [17] constructed a family of instances of degree 4 for which they claim all ascents are exponential. Finally, all ascents are quadratic when the constraint graph has maximum degree 2 (Theorem 5.6 in Kaznatcheev [13]). Taken together, this only leaves open the question of efficiency of greedy local search for binary Boolean VCSPs of degree 3.

Treewidth as a sparsity parameter has a more complicated – and perhaps more interesting – relationship to the efficiency of local search. There exists a polynomial time non-local search algorithm for finding not just a local maximum but the global maximum for VCSPs of bounded treewidth [2, 4]. Thus, binary Boolean VCSPs of bounded treewidth are not hard for PLS. Unlike bounded degree, bounded treewidth instances really are a class of simple instances. But the existence of a polynomial-time non-local search algorithm for solving bounded treewidth VCSPs, does not mean that local search will be efficient at finding even a local peak. Cohen et al. [5] have already shown that greedy local search requires an exponential number of steps for Boolean VCSPs of pathwidth 7. The simplest set of instances on which greedy local search fails was subsequently lowered to pathwidth 4 [15]. Recently, Kaznatcheev and Vazquez Alferez [16] showed that an old construction of Hopfield networks by Haken and Luby [9] provides a binary Boolean VCSP of pathwidth 3 on which greedy local search takes an exponential number of steps. Since Kaznatcheev, Cohen and Jeavons [14] have shown that *all* local search methods will take at most a quadratic number of steps on tree-structured binary Boolean VCSPs (i.e., treewidth 1), this leaves only the case of treewidth 2 as open for the question of whether greedy local search is fast or slow.

There are partial results for these two open questions on degree and treewidth – mostly focused on *all* or *some* ascents, rather than *steepest* ascents – but they cannot be further extended. Poljak [18] showed that if the VCSP instance is allowed only MAXCUT constraints then for degree 3 instances, all ascents are quadratic. However, Poljak's [18] technique of rewriting degree 3 MAXCUT constrain graphs by instances with small weights cannot extend to arbitrary binary constraints. There are degree 3 VCSP-instances that require exponentially large weights (Example 5.10 in Kaznatcheev [13]) and, in their Example 7.2, Kaznatcheev, Cohen and Jeavons [14] gave an explicit family of instances with max degree 3 where some ascents are exponential. Similarly, the encouragement-path proof techniques for showing that all local search methods are efficient on VCSPs of treewidth 1 [14] cannot be extended to treewidth 2. Specifically, Kaznatcheev, Cohen and Jeavons [14]'s Example 7.2 not only has maximum degree 3 but also pathwidth 2. However, although some ascents are exponentially long in this family of instances, the steepest ascents are all short and greedy local search can find the peak in a linear number of steps. More generally, Kaznatcheev and van Marle [15] optimistically conjectured that on *any* family of VCSP-instances of pathwidth 2, greedy local search will take at most a polynomial number of steps.

In this paper, we resolve these two open questions on the efficiency of greedy local search for VCSP-instances of degree 3 and treewidth 2.[1] In Section 3, we construct VCSP-instances $\mathcal{C}^{\pm}_{n,\leq n}$ on $6n$ Boolean variables with $7n-1$ binary constraints and $6n$ unary constraints with a constraint graph of maximum degree 3 and pathwidth 2 (Proposition 3). In Section 4, we show that greedy local search takes $7(2^n-1)$ steps to solve these simple VCSP-instances (Theorem 6). We construct $\mathcal{C}^{\pm}_{n,\leq m}$ as a path of m gadgets $\mathcal{C}^{\pm}_{n,m}, \mathcal{C}^{-}_{n,m-1}, \ldots, \mathcal{C}^{-}_{n,2}, \mathcal{C}^{-}_{n,1}$ with consecutive gadgets joined by a single binary constraint. Our proof of long steepest ascents is by induction on the number of gadgets in the path. We show that the steepest ascent first flips bits only in the first gadget $\mathcal{C}^{\pm}_{n,m}$ until it flips the variable that participates in the binary constraint linking $\mathcal{C}^{\pm}_{n,m}$ to $\mathcal{C}^{-}_{n,m-1}$. When this linking variable flips to 1 this gives us the inductive hypothesis of $\mathcal{C}^{+}_{n,\leq m-1}$ and when it flips to 0, it gives the inductive hypothesis of $\mathcal{C}^{-}_{n,\leq m-1}$. The constraint weights in the gadget are chosen in such a way that after this linking variable is flipped, the next potential flip in $\mathcal{C}^{\pm}_{n,m}$ increases fitness by a lower amount than the flips in $\mathcal{C}^{\pm}_{n,\leq m-1}$. Thus the steepest ascent continues in the part of the VCSP corresponding to the inductive hypothesis for $7(2^{m-1}-1)$ steps, until it reaches the local peak of the sub-instance and finally allows flips to continue in $\mathcal{C}^{\pm}_{n,m}$ that eventually result in the linking variable flipping back, repeating the recursive process for a second time.

We also show that our VCSP-instances are what Kaznatcheev and Vazquez Alferez [16] defined as "oriented" (Proposition 4). This means that the exponential running time of greedy local search on our instances stands in stark contrast to many other non-greedy local search methods that Kaznatcheev and Vazquez Alferez [16] have shown to take a quadratic or fewer steps to solve oriented VCSPs. From this we conclude that among local search methods, greed is particularly slow.

2 Background

Given some finite set V of variable indexes with size $|V| = d$,[2] an *assignment* is a d-dimensional Boolean vector $x \in \{0,1\}^d$. For every $i \in V$, x_i refers to the ith entry of x. To refer to a substring with variable indexes $S \subseteq V$, we write the partial assignment $x[S] \in \{0,1\}^S$. To complete a partial assignment, we write $x \in \{0,1\}^S y[V \setminus S]$ to mean that $x_i = y_i$ for $i \in V \setminus S$ and free otherwise. When assignments are sufficiently short we give x and y explicitly, for example, $x_1 0 x_3$ refers to an assignment to 3 variables where the second variable is set to 0. If we want to change just the assignment x_i at index i to a value $b \in \{0,1\}$ then we will write $x[i:b]$. Two assignments are *adjacent* if they differ on a single variable. Or more formally: $x, y \in \{0,1\}^d$ are adjacent if there exists an index $i \in [d]$ such that $y = x[i : \overline{x_i}]$, where $\overline{x_i} = 1 - x_i$.

A *fitness landscape* is any pseudo-Boolean function $f : \{0,1\}^V \to \mathbb{Z}$ together with the above notion of adjacent assignments. Given any $S \subseteq V$, the sublandscape on S given background $y \in \{0,1\}^{V \setminus S}$ is the function f restricted to inputs $x \in \{0,1\}^S y[V \setminus S]$. An

[1] In parallel work to ours, van Marle [21] developed a similar construction of exponential steepest ascent. His construction has the same constraint graph – with degree 3 and pathwidth 2 – but slightly different weights. The key difference is a much more involved proof of exponential steepest ascents: van Marle [21] shows how to simulate a particular prior construction of *some* exponential ascent as a *steepest* ascent on a "padded" instance. This is similar to the prior approaches taken by Cohen et al. [5] and Kaznteheev and van Marle [15]. In contrast, we focus on how to compose individual gadget's fitness landscapes and their steepest (partial) ascents rather than introducing new subgadgets for simulating whole ascents. We find our approach simpler, and think that future work can more easily generalize our approach to other local search methods.

[2] Most often this is $V = [d]$, but in our construction of $\mathcal{C}^{\pm}_{n,\leq m}$ in Section 3 it will be $V = [m] \times [6]$.

assignment x is called a *local peak* in a fitness landscape if $f(x) \geq f(y)$ for all y adjacent to x. A sequence of assignments $x^0, x^1, \ldots, x^T$ is called an *ascent* in the fitness landscape if x^{t-1} and x^t are adjacent for all $t \in [T]$, $f(x^{t-1}) < f(x^t)$, and x^T is a local peak. An ascent is called a *steepest* ascent, if for all $t \in [T]$ and all assignments y adjacent to x^{t-1} it is the case that $f(x^t) \geq f(y)$. Any local search method (that only takes increasing steps) follows an ascent. Greedy local search follows a steepest ascent.

A binary Boolean valued constraint satisfaction problem (VCSP) on d Boolean variables with indexes in V is a set of constraints $\mathcal{C} = \{c_S\}$. Each constraint is a weight $c_S \in \mathbb{Z} \setminus \{0\}$ with an associated *scope* $S \subseteq V$ of size $|S| \leq 2$. We say that c_S is a *unary* constraint if $|S| = 1$, and a *binary* constraint if $|S| = 2$. Overloading notation, the set of constraint $\mathcal{C}$ *implements* a pseudo-Boolean function $\mathcal{C} : \{0,1\}^d \to \mathbb{Z}$ that we call the *fitness function*:

$$\mathcal{C}(x) = c_\emptyset + \sum_{c_i \in \mathcal{C}} c_i x_i + \sum_{c_{\{i,j\}} \in \mathcal{C}} c_{\{i,j\}} x_i x_j \tag{1}$$

Solving $\mathcal{C}$ means finding an assignment x, that maximizes the fitness function $\mathcal{C}(x)$. [3]

When discussing graph-theoretical properties of $\mathcal{C}$, we treat the scopes of binary constraints as edges. So $V(\mathcal{C}) = V$ and $E(\mathcal{C}) = \{i, j \mid i \neq j \text{ and } c_{\{i,j\}} \in \mathcal{C}\}$ is the set of scopes of the binary constraints of $\mathcal{C}$. For each variable index $i \in V$, we define the neighbourhood $N_{\mathcal{C}}(i) = \{j \mid i, j \in E(\mathcal{C})\}$ as the set of variable indexes in $V \setminus \{i\}$ that appear in a constraint with i. In this paper, we measure the simplicity of a VCSP-instance $\mathcal{C}$ by the maximum degree and by the pathwidth of its constraint graph. Given a graph $G = (V, E)$, the *pathwidth* of G is the minimum possible width of a path decomposition of G. A *path decomposition* of G is a sequence of sets $P = \{X_1, X_2, \ldots, X_p\}$ where $X_r \subset V(G)$ for $r \in [p]$ with the following three properties:

1. Every vertex $v \in V(G)$ is in at least one set X_r.
2. For every edge $\{u, v\} \in E(G)$ there exists an $r \in [p]$ such that X_r contains both u and v.
3. For every vertex $v \in V(G)$, if $v \in X_r \cap X_s$ then $u \in X_\ell$ for all ℓ such that $r \leq \ell \leq s$.

The *width* of a path decomposition is defined as $\max_{X_r \in P} |X_r| - 1$. We refer the reader to [6] for more details and for the definition of treewidth, and to [7] for standard graph terminology.

It is often useful to see how the value of the fitness function $\mathcal{C}$ changes when a single variable is modified. In particular, we denote with $\nabla_i \mathcal{C}(x) = \mathcal{C}(x[i:1]) - \mathcal{C}(x[i:0])$ the fitness change associated with changing variable $x_i = 0$ to $x_i = 1$ given some background assignment x. It is easy to see that $\nabla_i \mathcal{C}(x) = c_i + \sum_{j \in N_{\mathcal{C}}(i)} c_{\{i,j\}} x_j$, and the value of $\nabla_i \mathcal{C}(x)$ depends only on the assignment to variables with indexes in $N_{\mathcal{C}}(i)$. We use this to overload $\nabla_i \mathcal{C}$ to partial assignments: if $y \in \{0,1\}^{N(i)}$ we consider $\nabla_i \mathcal{C}(y)$ to be well defined.

We say that x_i has *preferred assignment* 1 in background x if $\nabla_i \mathcal{C}(x) > 0$ and preferred assignment 0 in background x if $\nabla_i \mathcal{C}(x) < 0$.

---

[3] One could also consider a VCSP-instance as a set of constraints $\hat{\mathcal{C}} = \{C_S\}$ where each $C_S : \{0,1\}^S \to \mathbb{Z}$ is a binary function. This formulation is equivalent to our formulation above because an arbitrary constraint with scope S can be expressed as a polynomial. For example, take $S = \{i, j\}$:

$$\begin{aligned} C_S(x_i, x_j) &= C_S(0,0)(1-x_i)(1-x_j) + C_S(1,0)x_i(1-x_j) + C_S(0,1)(1-x_i)x_j + C_S(1,1)x_i x_j \\ &= C_S(0,0) + (C_S(1,0) - C_S(0,0))x_i + (C_S(0,1) - C_S(0,0))x_j \\ &\quad + (C_S(1,1) - C_S(0,1) - C_S(1,0) + C_S(0,0))x_i x_j \end{aligned}$$

The second equality groups alike monomials. One can convert from the $\hat{\mathcal{C}}$ formulation to the $\mathcal{C}$ formulation by summing alike monomial terms across all the constraints. That is, the constant terms $C_S(0,0)$ are aggregated into $c_\emptyset$, all the $C_{\{i,_\}}$ (i.e., coefficients appearing before x_i) aggregate into c_i, and $c_{\{i,j\}} = C_{\{i,j\}}(1,1) - C_{\{i,j\}}(0,1) - C_{\{i,j\}}(1,0) + C_{\{i,j\}}(0,0)$ (i.e., coefficient appearing before $x_i x_j$). It takes linear time to covert from $\hat{\mathcal{C}}$ to $\mathcal{C}$ (see Theorem 3.4 in Kaznatcheev, Cohen and Jeavons [14]).

▶ **Definition 1.** *Given two indexes $i \neq j$ we say that i* sign-depends *on j in background assignment x when $sign(\nabla_i \mathcal{C}(x)) \neq sign(\nabla_i \mathcal{C}(x[j : \overline{x_j}]))$. If there is no background assignment x such that i sign-depends on j then we say that i does not sign depend on j. If for all $j \neq i$ we have that i does not sign-depend on j then we say that i is sign independent.*

In other words, Definition 1 tells us that if i sign-depends on j, the preferred assignment of variable x_i depends on the assignment to variable x_j.

▶ **Definition 2** (Kaznatcheev and Vazquez Alferez [16])**.** *A* VCSP*-instance $\mathcal{C}$ is* oriented *if for every pair of indexes $\{i, j\}$ we have that either i does not sign-depend on j or j does not sign depend on i. If a* VCSP*-instance $\mathcal{C}$ is oriented and j sign depends on i we assign a direction from i to j to the edge $\{i, j\} \in E(\mathcal{C})$ (i.e. we orient the edge, hence the name).*

The constraint graphs of oriented VCSPs have no directed cycles [16]. Thus, if y is any assignment to the first k variables of a topological ordering of the variables of an oriented constraint graph, then k is sign-independent given background y. In other words, if $\mathcal{C}$ is oriented, there is an ordering on the variables such that later variables are conditionally-independent of earlier ones. This implies that the fitness landscape of $\mathcal{C}$ is single peaked on every sublandscape [16]. Depending on the research community, such landscapes are known as semismooth fitness landscapes [12, 16], completely unimodal pseudo-Boolean functions [10], or acyclic unique-sink orientations of the hypercube [19, 20]. Given that fitness landscapes implemented by oriented VCSPs are single-peaked, we use $x^*(\mathcal{C})$ to denote the peak of the landscape implemented by the oriented VCSP-instance $\mathcal{C}$.

3 Construction of $\mathcal{C}^{\pm}_{n,\leq m}$

Given two parameters $1 \leq m \leq n$, in this section, we construct the VCSP-instances $\mathcal{C}^{+}_{n,\leq m}$ and $\mathcal{C}^{-}_{n,\leq m}$ on $6m$ variables and, in Section 4, show that they both have a steepest ascent of length $7(2^m - 1)$. We construct the $\mathcal{C}^{\pm}_{n,\leq m}$ as a path of m gadgets $\mathcal{C}^{\pm}_{n,m}, \mathcal{C}^{-}_{n,m-1}, \ldots, \mathcal{C}^{-}_{n,2}, \mathcal{C}^{-}_{n,1}$ where each gadget $\mathcal{C}^{\pm}_{n,k}$ is defined on 6 variables $V_k = \{(k, i) \mid 1 \leq i \leq 6\}$. For notational convenience, we define $V_{\leq m} := \cup_{k=1}^{m} V_k$. Note that the two VCSPs are exactly the same except on the mth gadget where $\mathcal{C}^{+}_{n,\leq m}$ has $\mathcal{C}^{+}_{n,m}$ and $\mathcal{C}^{-}_{n,\leq m}$ has $\mathcal{C}^{-}_{n,m}$. We will present the construction of the gadgets $\mathcal{C}^{\pm}_{n,k}$ in two stages. First, we will define the scopes of all the constraints to get the constraint graph and show that these constraint graphs are very sparse (Proposition 3). Second, we will assign weights to the constraints and show that the VCSPs are oriented (Proposition 4).

Both the $\mathcal{C}^{-}_{n,k}$ and $\mathcal{C}^{+}_{n,k}$ gadgets have all six unary constraints and the same six binary constraints with scopes $\{(k, 1), (k, 2)\}$, $\{(k, 2), (k, 3)\}$, $\{(k, 3), (k, 6)\}$, $\{(k, 1), (k, 4)\}$, $\{(k, 4), (k, 5)\}$, $\{(k, 5), (k, 6)\}$. Finally, we connect adjacent gadgets with a single binary constraint with scope $\{(k, 6), (k-1, 1)\}$. The constraint graph of $\mathcal{C}^{-}_{n,k}$ along with the connections to the adjacent gadgets at $k+1$ and $k-1$ are shown in Figure 1. It is not hard to check based on the above definition that the constraint graphs of $\mathcal{C}^{\pm}_{n,\leq m}$ are sparse:

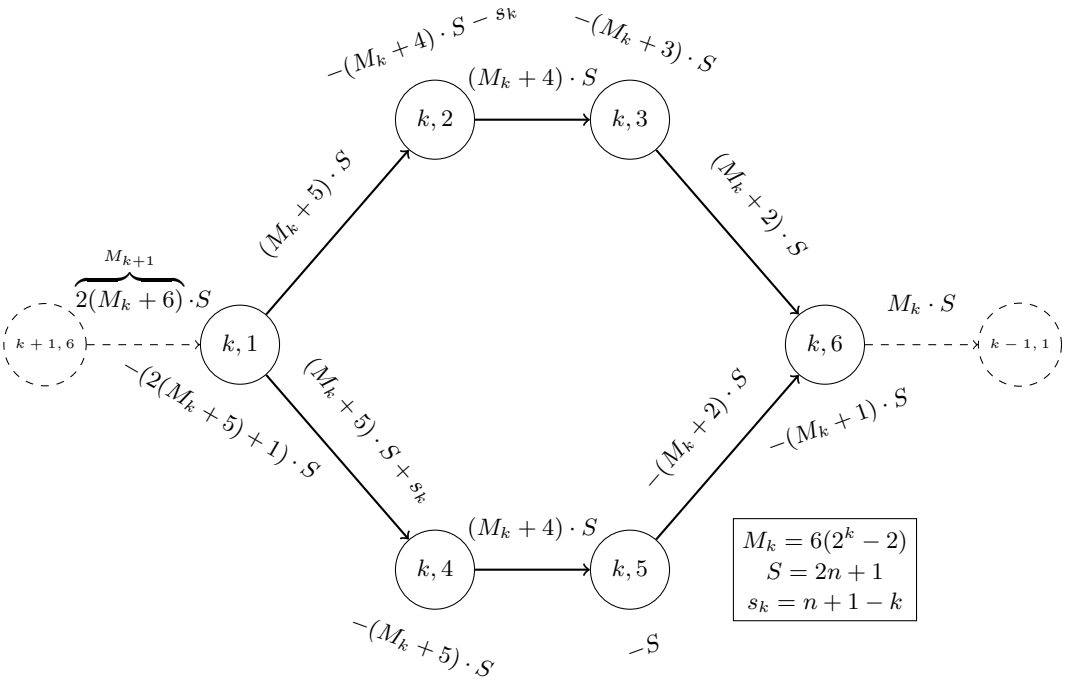


Figure 1 A gadget $\mathcal{C}^-_{n,k}$ with $M_k = 6(2^k-2)$, $S = 2n+1$, and $s_k = n+1-k$. The constraints of the kth of n gadgets are shown: the weights of unary constraints are next to their variables and the weights of binary constraints are above the edges that specify their scope. The orientation of the arcs is displayed, showing the instance is oriented. The dotted edges and vertices illustrate the connection to the neighboring gadgets. For the right boundary: there is no 0th gadget and thus no constraint with scope $\{(1,6),(0,1)\}$. For the left boundary: both of $\mathcal{C}^{\pm}_{n,\leq m}$ have no constraint with scope $\{(m+1,6),(m,1)\}$ but $\mathcal{C}^+_{n,m}$ also changes the weight of the unary on $(m,1)$ to $c^+_{(m,1)} = c^-_{(m,1)} + c_{(m+1,6),(m,1))} = S$.

▶ **Proposition 3.** *The constraint graph of $\mathcal{C}^{\pm}_{n,\leq m}$ has maximum degree 3 and pathwidth 2.*

Proof. The constraint graph of each $\mathcal{C}^{\pm}_{n,k}$ is a cycle so every vertex has degree 2. To create $\mathcal{C}^{\pm}_{n,\leq m}$ we add a single edge between consecutive gadgets, this raises the maximum degree to 3. For the pathwidth:
($\Rightarrow$) Path decomposition for $\mathcal{C}^{\pm}_{n,k}$ and adjacent variables: $\{(k+1,6),(k,1)\}$, $\{(k,1),(k,2),(k,4)\}$, $\{(k,2),(k,3),(k,4)\}$, $\{(k,3),(k,4),(k,5)\}$, $\{(k,3),(k,5),(k,6)\}$, $\{(k,6),(k-1,1)\}$.
($\Leftarrow$) Contracting $\{(k,1),(k,2)\}$, $\{(k,3),(k,6)\}$ and $\{(k,4),(k,5)\}$ shows K_3 is a minor of $\mathcal{C}^{\pm}_{n,k}$. ◀

We define the weights of the constraints sequentially using parameters $M_k = 6(2^k-2)$, $S = 2n+1$, and $s_k = n+1-k$. Since $C^-_{n,k}$ and $C^+_{n,k}$ are the same except for the unary on $(k,1)$, we use c for all the weights except for $c^-_{(k,1)}$ (Equation (14)) and $c^+_{(k,1)}$ (Equation (16)):

$$
\begin{aligned}
c_{\{(k,6),(k-1,1)\}} &= && M_k S && (2)\\
c_{(k,6)} &= -(|c_{\{(k,6),(k-1,1)\}}| + S) &&= -(M_k+1)S && (3)\\
c_{\{(k,3),(k,6)\}} &= |c_{(k,6)}| + S &&= (M_k+2)S && (4)\\
c_{\{(k,5),(k,6)\}} &= -|c_{\{(k,3),(k,6)\}}| &&= -(M_k+2)S && (5)\\
c_{(k,3)} &= -(|c_{\{(k,3),(k,6)\}}| + S) &&= -(M_k+3)S && (6)\\
c_{\{(k,2),(k,3)\}} &= |c_{(k,3)}| + S &&= (M_k+4)S && (7)\\
c_{(k,2)} &= -(|c_{\{(k,2),(k,3)\}}| + s_k) &&= -(M_k+4)S - s_k && (8)\\
c_{\{(k,1),(k,2)\}} &= |c_{(k,2)}| + S - s_k &&= (M_k+5)S && (9)\\
c_{(k,5)} &= &&= -S && (10)\\
c_{\{(k,4),(k,5)\}} &= |c_{(k,5)} + c_{\{(k,5),(k,6)\}}| + S &&= (M_k+4)S && (11)\\
c_{(k,4)} &= -(|c_{\{(k,4),(k,5)\}}| + S) &&= -(M_k+5)S && (12)\\
c_{\{(k,1),(k,4)\}} &= |c_{(k,4)}| + s_k &&= (M_k+5)S + s_k && (13)\\
c^-_{(k,1)} &= -(|c_{\{(k,1),(k,2)\}} + c_{\{(k,1),(k,4)\}}| + S - s_k) &&= -(2(M_k+5)+1)S && (14)\\
c_{\{(k+1,6),(k,1)\}} &= |c_{(k,1)}| + S &&= \underbrace{2(M_k+6)}_{M_{k+1}} S && (15)\\
c^+_{(k,1)} &= c^-_{(k,1)} + c_{\{(k+1,6),(k,1)\}} &&= S && (16)
\end{aligned}
$$

The above weights are shown on the constraint graph in Figure 1. This structure of weights has three important features that ensure the VCSP is oriented, that let us recurse on smaller k, and that help us control the steepest ascent. We have set the above weights in such a way that the following three properties hold in $\mathcal{C}^-_{n,k}$:

(a) all the unaries are negative,

(b) the magnitude of each variable's unary is greater than the binary constraint from that variable to a variable in the gadget with higher second index (or than the sum of the two binaries in the case of $|c^-_{(k,1)}| > c_{\{(k,1),(k,2)\}} + c_{\{(k,1),(k,4)\}}$), and

(c) the sum of the weights of any subset of "incoming" binaries on a variable (i.e., binary constraint to that variable from a variable in the gadget with lower second index) is either non-positive or greater than the magnitude of the unary (or the sum of the unary and any negative "outgoing" binaries in the case of $|c_{\{(k,4),(k,5)\}}| > |c_{(k,5)}| + |c_{\{(k,5),(k,6)\}}|$).

These three properties ensure that the resulting VCSP is oriented:

▶ **Proposition 4.** *$\mathcal{C}^{\pm}_{n,k}$ and $\mathcal{C}^{\pm}_{n,\leq m}$ are oriented.*

Proof. First we prove that $\mathcal{C}^{\pm}_{n,k}$ are oriented as in Figure 1. Note that $\mathcal{C}^{\pm}_{n,k}$ are cycles and every vertex has degree 2. We will denote by (k,i) and (k,j) the two neighbors of an arbitrary variable $(k,h) \in V_k$. Table 1 shows the fitness change $\nabla^{\pm}_{(k,h)}\mathcal{C}(x_{(k,i)}x_{(k,j)})$ for all possible assignments to $x_{(k,i)}$ and $x_{(k,j)}$. The cells of Table 1 are colored according to the sign of $\nabla^{\pm}_{(k,h)}\mathcal{C}$. It suffices to check Table 1 against Definition 2: The sign of $\nabla^{+}_{(k,1)}$ is always positive, and the sign of $\nabla^{-}_{(k,1)}$ is always negative, regardless of the assignment to $(k,2)$ and $(k,4)$, so $(k,1)$ does not sign depend on either $(k,2)$ or $(k,4)$ in $\nabla^{\pm}_{(k,1)}$.

On the other hand, for the rows where $h = 2,3,4$ and 5 the two columns where $x_{(k,i)} = 0$ are negative whilst the two columns corresponding to $x_{(k,i)} = 1$ are positive. This means that $(k,2),(k,3),(k,4)$ and $(k,5)$ sign-depend, respectively, on $(k,1),(k,2),(k,1)$ and $(k,4)$;

Table 1 Fitness change $\nabla_{(k,h)}\mathcal{C}^{\pm}_{n,k}$ incurred by flipping the assignment of variable (k,h) from $x_{(k,h)}=0$ to $x_{(k,h)}=1$ given the assignments of its two neighbors $x_{(k,i)}$ and $x_{(k,j)}$ in $\mathcal{C}^{\pm}_{n,k}$. We abbreviate $\nabla_{(k,h)}\mathcal{C}^{\pm}_{n,k}$ as $\nabla^{\pm}_{(k,h)}$. The dark gray cells highlight negative changes, and light gray cells highlight positive changes.

				$(\ x_{(k,i)}\ ,\ x_{(k,j)}\ )$			
h	i	j	$\nabla^{\pm}_{(k,h)}$	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
1	4	2	$\nabla^{+}_{(k,1)}$	S	$(M_k{+}6)S{+}s_k$	$(M_k{+}6)S$	$(2M_k{+}11)S{+}s_k$
			$\nabla^{-}_{(k,1)}$	$-(2(M_k{+}5){+}1)S$	$-(M_k{+}6)S{+}s_k$	$-(M_k{+}6)S$	$-S{+}s_k$
2	1	3	$\nabla^{\pm}_{(k,2)}$	$-(M_k{+}4)S{-}s_k$	$-s_k$	$S{-}s_k$	$(M_k{+}5)S{-}s_k$
3	2	6	$\nabla^{\pm}_{(k,3)}$	$-(M_k{+}3)S$	$-S$	S	$(M_k{+}3)S$
4	1	5	$\nabla^{\pm}_{(k,4)}$	$-(M_k{+}5)S$	$-S$	s_k	$(M_k{+}4)S{+}s_k$
5	4	6	$\nabla^{\pm}_{(k,5)}$	$-S$	$-(M_k{+}3)S$	$(M_k{+}3)S$	S
6	3	5	$\nabla^{\pm}_{(k,6)}$	$-(M_k{+}1)S$	S	$-(2M_k{+}3)S$	$-(M_k{+}1)S$

but do not sign-depend, respectively, on $(k,3),(k,6),(k,5)$ and $(k,6)$. Finally, from the fact that the row with $h=6$ has different signs on the two columns where $x_{(k,3)}=0$, and also different signs on the two columns where $x_{(k,5)}=1$ we can see that $(k,6)$ sign-depends on $(k,3)$ and $(k,5)$ in $\nabla^{\pm}_{(k,1)}$.

Second, to show that $\mathcal{C}^{\pm}_{n,\leq m}$ are oriented, it suffices to show that for $1\leq k<m$ the preferred assignment to $x_{(k+1,6)}$ is independent of the assignment to $x_{(k,1)}$. This is equivalent to showing that the sign of the last row in Table 1 does not change if we add M_kS to each column, which is obviously true. This is sufficient to show the statement of the proposition.

To show the additional property that the orientation of the edge $\{x_{(k+1,6)},x_{(k,1)}\}$ is as it appears in Figure 1, we must show that $(k,1)$ sign-depends on the assignment to $(k+1,6)$. But this can be seen from the fact that the first row of our table is equivalent to having $x_{(k+1,6)}=1$ in the background, and the second row is equivalent to having $x_{(k+1,6)}=0$. Since the first and second rows have different signs the sign-dependence follows. ◀

The weight of $c^{+}_{(k,1)}$ in Equation (16) is set so that we have the next two properties:

(d) If we fix $x_{(k,6)}=0$ then the sublandscape spanned by $V_{\leq k-1}$ is the same as the landscape implemented by $\mathcal{C}^{-}_{n,\leq k-1}$, and

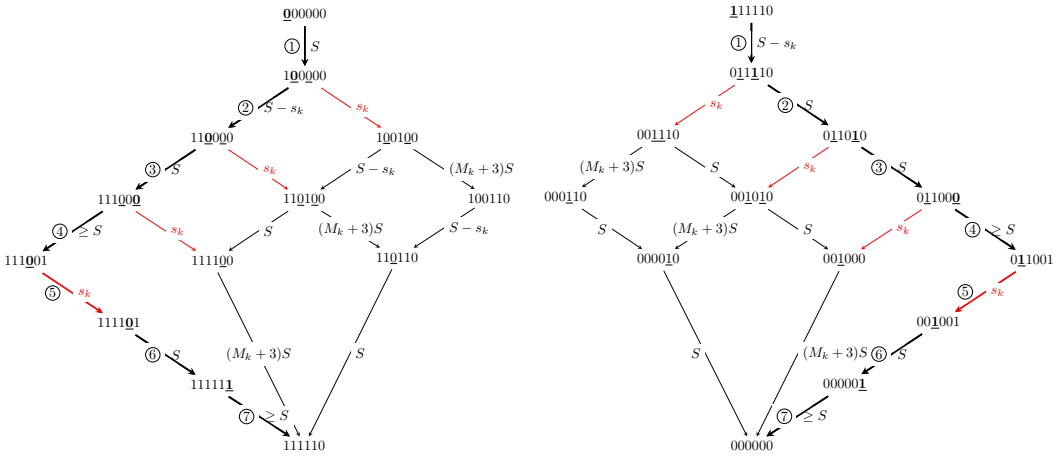
(e) if we fix $x_{(k,6)}=1$ then the sublandscape spanned by $V_{\leq k-1}$ is the same as the landscape implemented by $\mathcal{C}^{+}_{n,\leq k-1}$.

This allows us to analyze ascents on $\mathcal{C}^{\pm}_{n,\leq m}$ inductively because when $x_{(m,6)}$ is fixed to 1 or 0 we can use the analysis from our inductive hypothesis of $\mathcal{C}^{+}_{n,\leq m-1}$ or $\mathcal{C}^{-}_{n,\leq m-1}$, respectively. It also makes it very easy to check for the peaks of our landscapes:

▶ **Proposition 5.** *The semismooth fitness landscapes of $\mathcal{C}^{-}_{n,k}$, $\mathcal{C}^{-}_{n,\leq m}$, $\mathcal{C}^{+}_{n,k}$, and $\mathcal{C}^{+}_{n,\leq m}$ have their unique fitness peaks at $x^*(\mathcal{C}^{-}_{n,k})=000000$, $x^*(\mathcal{C}^{-}_{n,\leq m})=0^{6m}$, $x^*(\mathcal{C}^{+}_{n,k})=111110$, and $x^*(\mathcal{C}^{+}_{n,\leq m})=x^*(\mathcal{C}^{+}_{n,m})0^{6(m-1)}=1111100^{6(m-1)}$, respectively.*[4]

Proof. By property (a), all the unaries in $\mathcal{C}^{-}_{n,k}$ and $\mathcal{C}^{-}_{n,\leq m}$ are negative, so the all zero assignment is a local peak. Since the landscapes are semismooth from Proposition 4, this is also the unique global peak.

[4] For convenience, when we specify an assignment x the variables are ordered from left to right by decreasing first index and, to break ties, by increasing second index. For example, for $\mathcal{C}^{+}_{n,\leq m}$ the assignment $x=0100010^{6(m-1)}$ sets $x_{(m,2)}=1, x_{(m,6)}=1$ and all other variables to 0.



(a) Partial fitness landscape of $\mathcal{C}^+_{n,k}$. **(b)** Partial fitness landscape of $\mathcal{C}^-_{n,k}$.

Figure 2 All ascents from 000000 in the fitness landscape of $\mathcal{C}^+_{n,k}$ **(a)** and from 111110 in the fitness landscape of $\mathcal{C}^-_{n,k}$ **(b)**. The increase in fitness of each step is shown on the edges, with small increments (s_k) highlighted in red. The steepest ascent is shown in bold. For emphasis, bits that lead to a fitness increase when flipped are underlined, and the bit flipped by steepest ascent is bolded and numbered in the same way as in Equations (19)–(22) and Equations (23)–(26).

For $\mathcal{C}^+_{n,k}$, $x_{(k,1)}$ is conditionally-independent of all other variables and has $c^+_{(k,1)} = S > 0$, so the preferred assignment is $x_{(k,1)} = 1$. With $x_{(k,1)}$ set to 1, the preferred assignments for $x_{(k,2)}$ and $x_{(k,4)}$ are also 1; and based on those, the preferred assignments for $x_{(k,3)}$ and $x_{(k,5)}$ are also 1. This leaves $x_{(k,6)}$ which, conditional on $x_{(k,3)} = x_{(k,5)} = 1$, prefers $x_{(k,6)} = 0$. Overall, this gives $x^*(\mathcal{C}^+_{n,k}) = 111110$.

Since $x_{(k,6)} = 0$, property (d) implies that $x^*(\mathcal{C}^+_{n,\leq m}) = x^*(\mathcal{C}^+_{n,m})x^*(\mathcal{C}^-_{n,\leq m-1})$. ◀

Finally, the difference in increase of Equations (8) and (13) from the other weights ensures that:

(f) steps from assignments $0\underline{1}1x_4x_5x_6$ to $0\mathbf{0}1x_4x_5x_6$ and from $1x_2x_3\underline{0}0x_6$ to $1x_2x_3\mathbf{1}0x_6$ increase fitness by exactly $s_k \leq n$, and

(g) all other fitness increasing steps increase fitness by at least $S - s_k > n$.

As we will see in the next section, these "small" steps allow us to control in what order each block V_k of variables appears in the steps of the steepest ascent. Combined with properties (d) and (e) this lets us show the exponential steepest ascent by induction on m.

4 Exponential steepest ascent in the landscape of $\mathcal{C}^{\pm}_{n,\leq m}$

We can now show that it takes a large number of steps to go from the assignment $x^*(\mathcal{C}^-_{n,\leq m}) = 0^{6m}$ to the peak $x^*(\mathcal{C}^+_{n,\leq m}) = 1111100^{6(m-1)}$ in the semismooth fitness landscape implemented by $\mathcal{C}^+_{n,\leq m}$ (and vice versa for the landscape implemented by $\mathcal{C}^-_{n,\leq m}$):

▶ **Theorem 6.** *Both the steepest ascents starting from $x^*(\mathcal{C}^-_{n,\leq m}) = 0^{6m}$ in the fitness landscape of $\mathcal{C}^+_{n,\leq m}$ and from $x^*(\mathcal{C}^+_{n,\leq m}) = 1111100^{6(m-1)}$ in the fitness landscape of $\mathcal{C}^-_{n,\leq m}$ have length $7(2^m - 1)$, where each step increases fitness by at least s_m.*

Proof. Our proof is by induction on the number of gadgets m. We will show that by adding the gadget $\mathcal{C}^{\pm}_{n,m}$, steepest ascents of length T_{m-1} in the landscape of $\mathcal{C}^{\pm}_{n,\leq m-1}$ will convert to steepest ascents of length $T_m = 7 + 2T_{m-1}$ in the landscape of $\mathcal{C}^{\pm}_{n,\leq m}$. To do this, we will

look at all the ascents that take us from $x^*(\mathcal{C}^-_{n,m}) = 000000$ to $x^*(\mathcal{C}^+_{n,m}) = 111110$ in the landscape of the gadget $\mathcal{C}^+_{n,m}$ and vice-versa for the gadget $\mathcal{C}^+_{n,m}$. All of these ascents are in Figure 2a for $\mathcal{C}^+_{n,m}$ and in Figure 2b for $\mathcal{C}^-_{n,m}$. Each arrow in Figure 2 is labeled by the fitness increase from flipping the appropriate bit. The steepest ascent is bolded.

As we can see from the figures, although there exists a minimal ascent of length five between these two assignments that does not flip the $x_{(m,6)}$ variable, this is not the steepest ascent. Instead, the steepest ascent from $x^*(\mathcal{C}^-_{n,m})$ to $x^*(\mathcal{C}^+_{n,m})$ in the fitness landscape of $\mathcal{C}^+_{n,m}$ (and vice-versa for $\mathcal{C}^-_{n,m}$) takes seven steps and flips $x_{(m,6)}$ twice (at step ④ and ⑦).

This double flip of $x_{(m,6)}$ is what creates the recursion in $\mathcal{C}^\pm_{n,\leq m}$ that forces the steepest ascent in $\mathcal{C}^\pm_m$ to trigger twice as many ascents in $\mathcal{C}^\pm_{m-1}$. Specifically, although the first four steps of the steepest ascent in the landscape of $\mathcal{C}^\pm_{n,m}$ increase fitness by a large amount $\geq S - s_m > n$, step ⑤ increases fitness by only s_m. Thus, the steepest ascent in the sublandscape spanned by V_m "pauses" after step ④ and lets the steepest ascents in V_{m-1} take over with steps that increase fitness by an amount $\geq s_{m-1} > s_m$.

With this intuition in mind, look at the steepest ascent in the fitness landscape of $\mathcal{C}^+_{n,\leq m}$ starting from 0^{6m}. For brevity, define the (partial) assignments $x^*_\pm$ as the peaks of $\mathcal{C}^\pm_{n,\leq m-1}$:

$$x^*_- := x^*(\mathcal{C}^-_{n,\leq m-1}) = 0^{6(m-1)}, \text{ and} \tag{17}$$

$$x^*_+ := x^*(\mathcal{C}^+_{n,\leq m-1}) = 1111100^{6(m-2)}. \tag{18}$$

Now we can rewrite our starting assignment $x^*(\mathcal{C}^-_{n,\leq m}) = 0^{6m}$ as $000000x^*_-$ and note that the first four flips are entirely in V_m:

$$\overbrace{\underline{\mathbf{0}}00000x^*_-}^{x^*(\mathcal{C}^-_{n,\leq m})} \xrightarrow{①} 1\underline{\mathbf{0}}0000x^*_- \xrightarrow{②} 11\underline{\mathbf{0}}000x^*_- \xrightarrow{③} 111\underline{00}\underline{\mathbf{0}}x^*_- \xrightarrow{④} 111\underline{00}1\underline{\mathbf{x}^*_-} \tag{19}$$

where the variables that can flip are underlined and the variable that will be flipped by steepest ascent is bolded. For step ① there is only one choice to flip. On the subsequent two steps (②,③) the first of the two 0s is chosen by steepest ascent because they increase fitness by $S - s_m$ and S (which are both $> n$) while flipping the second 0 to a 1 would increase fitness by only $s_m \leq n$.

Step ④ is the most interesting. It flips $x_{(m,6)}$ since that increases fitness by $S > n \geq s_m$. In so doing, this flip transforms the landscape for variables on $V_{\leq m-1}$ from the one implemented by $\mathcal{C}^-_{n,\leq m-1}$ to the one implemented by $\mathcal{C}^+_{n,\leq m-1}$. In the landscape of $\mathcal{C}^+_{n,\leq m-1}$, x^*_- is no longer the peak, and hence it is underlined. Furthermore x^*_- is bolded because, by construction, all fitness increasing flips of variables in $V_{\leq m-1}$ increase fitness by $\geq s_{m-1} > s_m$ and so steepest ascent will flip variables in $V_{\leq m-1}$ instead of the $x_{(m,3)}$ flip that only increases fitness by s_m:

$$111\underline{00}1\underline{\mathbf{x}^*_-} \xrightarrow{T_{m-1} \text{ steps of } V_{\leq m-1} \text{ in landscape of } \mathcal{C}^+_{n,\leq m-1}} 111\underline{\mathbf{0}}01x^*_+ \tag{20}$$

Once all the steps in $V_{\leq m-1}$ are taken, steepest ascent can return to V_m, where the only remaining fitness-increasing step is the small step at $x_{(m,4)}$ that increases fitness by only s_m. This step subsequently opens two more steps in V_m:

$$111\underline{\mathbf{0}}01x^*_+ \xrightarrow{⑤} 1111\underline{\mathbf{0}}1x^*_+ \xrightarrow{⑥} 11111\underline{\mathbf{1}}x^*_+ \xrightarrow{⑦} 111110\underline{\mathbf{x}^*_+} \tag{21}$$

As with the first four steps in Equation (19), the most interesting step is the final step (⑦). It flips $x_{(m,6)}$ from 1 to $x_{(m,6)} = 0$. In so doing, this flip transforms the landscape for variables on $V_{\leq m-1}$ from the one implemented by $\mathcal{C}^+_{n,\leq m-1}$ to the one implemented by

$\mathcal{C}^-_{n,\leq m-1}$. Thus, "undoing" step ④. In the landscape of $\mathcal{C}^-_{n,\leq m-1}$, x^*_+ is no longer the peak, and hence underlined and bolded. Steepest ascent finishes with all remaining steps in $V_{\leq m-1}$:

$$111110\underline{\mathbf{x}^*_+} \xrightarrow{T_{m-1} \text{ steps of } V_{\leq m-1} \text{ in landscape of } \mathcal{C}^-_{n,\leq m-1}} \underbrace{111110x^*_-}_{x^*(\mathcal{C}^+_{n,\leq m})} \tag{22}$$

Similar to the steepest ascent in Equations (19)–(22), the steepest ascent in the fitness landscape of $\mathcal{C}^-_{n,\leq m}$ starting from the assignment $x^*(\mathcal{C}^+_{n,\leq m})$ has the following steps:

$$\overbrace{\underline{\mathbf{1}}11110x^*_-}^{x^*(\mathcal{C}^+_{n,\leq m})} \xrightarrow{①} 0\underline{\mathbf{1}}\underline{\mathbf{1}}10x^*_- \xrightarrow{②} 0\underline{\mathbf{1}}10\underline{\mathbf{1}}0x^*_- \xrightarrow{③} 0\underline{\mathbf{1}}100\underline{\mathbf{0}}x^*_- \xrightarrow{④} 0\underline{\mathbf{1}}1001\underline{\mathbf{x}^*_-} \tag{23}$$

$$0\underline{\mathbf{1}}1001\underline{\mathbf{x}^*_-} \xrightarrow{T_{m-1} \text{ steps of } V_{\leq m-1} \text{ in landscape of } \mathcal{C}^+_{n,\leq m-1}} 0\underline{\mathbf{1}}1001x^*_+ \tag{24}$$

$$0\underline{\mathbf{1}}1001x^*_+ \xrightarrow{⑤} 00\underline{\mathbf{1}}001x^*_+ \xrightarrow{⑥} 00000\underline{\mathbf{1}}x^*_+ \xrightarrow{⑦} 000000\underline{\mathbf{x}^*_+} \tag{25}$$

$$000000\underline{\mathbf{x}^*_+} \xrightarrow{T_{m-1} \text{ steps of } V_{\leq m-1} \text{ in landscape of } \mathcal{C}^-_{n,\leq m-1}} \underbrace{000000x^*_-}_{x^*(\mathcal{C}^-_{n,\leq m})} \tag{26}$$

Finally, both the steepest ascent in the fitness landscape of $\mathcal{C}^+_{n,\leq m}$ starting from the assignment $x^*(\mathcal{C}^-_{n,\leq m})$ (Equations (19)–(22)) and the steepest ascent in the fitness landscape of $\mathcal{C}^-_{n,\leq m}$ starting from the assignment $x^*(\mathcal{C}^+_{n,\leq m})$ (Equations (23)–(26)) have lengths of $T_m = 7+2T_{m-1}$ steps. The recurrence of $T_1 = 7$ and $T_m = 7 + 2T_{m-1}$ is solved by $T_m = 7(2^m - 1)$. ◀

Combining Theorem 6 with that greedy local search follows steepest ascents, and using the facts that our instances are very sparse by Proposition 3, and that they are oriented by Proposition 4, we conclude that greed is slow on sparse graphs of oriented valued constraints:

▶ **Theorem 7.** *Greedy local search can take* $7(2^n - 1)$ *steps to find the unique local optimum in oriented binary Boolean* VCSP*-instances* $\mathcal{C}^{\pm}_{n,\leq n}$ *on* $6n$ *variables with* $6n$ *unary constraints and* $7n - 1$ *binary constraints, maximum degree* 3 *and pathwidth* 2.

5 Discussion

There are two important structural features of our construction: that it is sparse (Proposition 3) and that is is oriented (Proposition 4). Sparseness resolves in the negative the two open questions on the efficiency of greedy local search for VCSP-instances of degree 3 and treewidth 2. Given that all ascents are short for VCSPs of degree 2 [13] and treewidth 1 [14], the $\mathcal{C}^{\pm}_{n,\leq n}$ family of instances that are hard for greedy local search belong to the simplest class of instances where *some* local search has the possibility to fail. That $\mathcal{C}^{\pm}_{n,\leq n}$ is an oriented VCSP (Proposition 4), reminds us that this failure of greedy local search is not due to the popular suspect of "bad" local peaks blocking the way to a hard to find global peak. Oriented VCSPs only produce semismooth fitness landscapes that are single peaked on each sublandscape [16]: there are no "bad" local peaks in this family of instances, there is the single global peak. Further, in semismooth fitness landscapes, there is always a short ascent of Hamming distance from any initial assignments to unique peak [10, 12]. A short ascents is always available in $\mathcal{C}^{\pm}_{n,\leq n}$, but greed stops us from find it.

Many other local search heuristics do not lose their way on oriented VCSPs. In contrast to Theorem 6, consider how a local search method that chooses improving steps at random – known as random ascent – behaves on $\mathcal{C}^{-}_{n,\leq n}$ starting from any assignment. Since there are $6n$ variables, any improving bit flip has a probability of at least $1/6n$ of being chosen as the next step. Thus, if $x_{(n,1)} \neq 0 (= x^*_{(n,1)})$ then after an expected number of at most $6n$ steps, it will be flipped to 0. Given the oriented constraints, it cannot be flipped back for the rest of the run. Next, if – at this point in the run – the current assignment has $x_{(n,2)} \neq 0$ or $x_{(n,4)} \neq 0$ then in an expected number of about $2 \cdot 6n$ steps, $x_{(n,2)}$ and $x_{(n,4)}$ will also be flipped to 0. Given the oriented constraints and that the only in-edge to these variables is from $x_{(n,1)}$ which is now fixed to 0, $x_{(n,2)}$ and $x_{(n,4)}$ will not be flipped back for the rest of the run. This logic continues for $x_{(n,3)}$ and $x_{(n,5)}$, then $x_{(n,6)}$, then $x_{(n-1,1)}$ and all other remaining variables following the arrows of the oriented constraints (i.e., in the topological order of the oriented VCSP). This simple argument gives us a bound of $(6n)^2$ on the expected number of steps for random ascent to find the peak.

The behavior of inadvertently fixing variables to their optimal state in the topological order of an oriented VCSP is a feature of many local search methods, not just random ascent. Kaznatcheev and Vazquez Alferez [16] give a more careful analysis of random ascent on general oriented binary Boolean VCSPs. Applying their bound yields an expected number of at most $32n^2 - 18n$ steps to solve $\mathcal{C}^{\pm}_{n,\leq n}$. For the task of solving general oriented VCSPs they give quadratic bounds on the number of steps taken by, random ascent, simulated annealing, Zadeh's simplex rule, the Kerninghan-Lin heuristic, and various other local search methods. Thus, not only does greedy local search require an exponential number of steps on oriented VCSPs of maximum degree 3 and pathwidth 2, but lots of other local search methods can solve these instances in quadratic time. We conclude that, among local search methods, greed is slow.

References

1 Emile Aarts and Jan Karel Lenstra, editors. *Local Search in Combinatorial Optimization*. Princeton University Press, Princeton, NJ, USA, 2003.

2 Umberto Bertelè and Francesco Brioschi. On non-serial dynamic programming. *Journal of Combinatorial Theory, Series A*, 14(2):137–148, 1973. doi:10.1016/0097-3165(73)90016-2.

3 Michaela Borzechowski. The complexity class polynomial local search (pls) and pls-complete problems. Bachelor's thesis, Freie Universitat Berlin, 2016. URL: https://www.mi.fu-berlin.de/inf/groups/ag-ti/theses/download/Borzechowski16.pdf.

4 Clément Carbonnel, Miguel Romero, and Stanislav Živný. The complexity of general-valued constraint satisfaction problems seen from the other side. *SIAM Journal on Computing*, 51(1):19–69, 2022. doi:10.1137/19M1250121.

5 David A Cohen, Martin C Cooper, Artem Kaznatcheev, and Mark Wallace. Steepest ascent can be exponential in bounded treewidth problems. *Operations Research Letters*, 48:217–224, 2020. doi:10.1016/j.orl.2020.02.010.

6 Marek Cygan, Fedor V Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*. Springer International Publishing, 2015. doi:10.1007/978-3-319-21275-3.

7 Reinhard Diestel. *Graph Theory*. Springer Publishing Company, Incorporated, 5th edition, 2017.

8 Robert Elsässer and Tobias Tscheuschner. Settling the complexity of local max-cut (almost) completely. In *International Colloquium on Automata, Languages, and Programming*, pages 171–182. Springer, 2011. doi:10.1007/978-3-642-22006-7_15.

9 Armin Haken and Michael Luby. Steepest descent can take exponential time for symmetric connection networks. *Complex Syst.*, 2(2):191–196, April 1988. URL: `http://www.complex-systems.com/abstracts/v02_i02_a03.html`.

10 Peter L Hammer, Bruno Simeone, Th M Liebling, and Dominique de Werra. From linear separability to unimodality: A hierarchy of pseudo-boolean functions. *SIAM Journal on Discrete Mathematics*, 1(2):174–184, 1988. `doi:10.1137/0401019`.

11 David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37(1):79–100, 1988. `doi:10.1016/0022-0000(88)90046-3`.

12 Artem Kaznatcheev. Computational complexity as an ultimate constraint on evolution. *Genetics*, 212(1):245–265, 2019. `doi:10.1534/genetics.119.302000`.

13 Artem Kaznatcheev. *Algorithmic Biology of Evolution and Ecology*. PhD thesis, University of Oxford, 2020.

14 Artem Kaznatcheev, David A Cohen, and Peter Jeavons. Representing fitness landscapes by valued constraints to understand the complexity of local search. *Journal of Artificial Intelligence Research*, 69:1077–1102, 2020. `doi:10.1613/jair.1.12156`.

15 Artem Kaznatcheev and Melle van Marle. Exponential steepest ascent from valued constraint graphs of pathwidth four. In *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, pages 17:1–17:16, 2024. `doi:10.4230/LIPIcs.CP.2024.17`.

16 Artem Kaznatcheev and Sofia Vazquez Alferez. When is local search both effective and efficient?, 2024. `doi:10.48550/arXiv.2410.02634`.

17 Burkhard Monien and Tobias Tscheuschner. On the power of nodes of degree four in the local max-cut problem. In *Algorithms and Complexity: 7th International Conference, CIAC 2010, Rome, Italy, May 26-28, 2010. Proceedings 7*, pages 264–275. Springer, 2010. `doi:10.1007/978-3-642-13073-1_24`.

18 Svatopluk Poljak. Integer linear programs and local search for max-cut. *SIAM Journal on Computing*, 24(4):822–839, 1995. `doi:10.1137/S0097539793245350`.

19 Ingo A Schurr. *Unique sink orientations of cubes*. PhD thesis, ETH Zurich, 2004.

20 Tibor Szabó and Emo Welzl. Unique sink orientations of cubes. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 547–555. IEEE, 2001. `doi:10.1109/SFCS.2001.959931`.

21 Melle van Marle. Complexity of greedy local search for constraint satisfaction. Master's thesis, Utrecht University, 2025.

22 David P. Williamson and David B. Shmoys. *Greedy Algorithms and Local Search*, pages 27–56. Cambridge University Press, 2011.

Analyzing Self-Stabilization of Synchronous Unison via Propositional Satisfiability

Asma Khoualdia
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Sami Cherif
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Stéphane Devismes
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Léo Robert
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Abstract

Synchronous unison is a classical clock synchronization problem in distributed computing, and especially in self-stabilization. This paper explores the self-stabilization of a synchronous unison algorithm proposed by Arora *et al.* using a propositional satisfiability-based approach. We give a logical formulation of the algorithm. This formulation includes the uniqueness of clock values at each node, the updates of clocks based on the minimum clock value in the neighborhood, and the detection of convergence or divergence. To optimize the models, additional constraints are introduced to reduce redundant cases of initial configurations to be analyzed. Our approach not only verifies the algorithm's behaviour but also offers insights into enhancing its robustness and applicability to broader distributed systems.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Constraint and logic programming; Theory of computation → Logic and verification

Keywords and phrases Self-stabilization, Synchronous Unison, Satisfiability

Digital Object Identifier 10.4230/LIPIcs.CP.2025.19

Supplementary Material
Software (Source Code and Benchmarks): `https://github.com/asmakhoualdia98/SU_SAT_Exec` [26]
archived at `swh:1:dir:4bc78d189155024f85110cb2dc9ae4aeb470d8bf`

Funding This work is supported by the project ANR-24-CE23-6126 (BforSAT), funded by the French National Research Agency. This work was also granted access to HPC resources of "Plateforme MatriCS" within University of Picardie Jules Verne, co-financed by the European Union with the European Regional Development Fund (FEDER) and the Hauts-De-France Regional Council among others.

1 Introduction

The notion of self-stabilization has been introduced in 1974 by Dijkstra [18] in the context of distributed systems. A distributed system is a set of computational entities, usually referred to as *processes*, that are both autonomous and interconnected. The goal of these processes is to cooperate to solve a task global to the system. An algorithm is said to be *self-stabilizing* if, regardless of the initial configuration of the system on which it is deployed, it guarantees to recover within finite time, and without external intervention, a so-called *legitimate configuration* from which the system specification is satisfied. Although the notion of failure is not explicit in the definition of self-stabilizing systems, the primary motivation of this approach is nevertheless fault tolerance. Indeed, after a finite number of transient

© Asma Khoualdia, Sami Cherif, Stéphane Devismes, and Léo Robert;
licensed under Creative Commons License CC-BY 4.0

failures[1], the configuration of a distributed system may be arbitrary and thus no longer satisfy its specification. A self-stabilizing algorithm then guarantees that the system recovers from such faults within finite time.

Designing self-stabilizing algorithms may seem complex and subtle at first glance since processes only have a partial view of the global configuration of the system. Yet, many of these algorithms turn out to be surprisingly elegant and sometimes even simpler than their non-stabilizing counterparts. However, formally demonstrating the self-stabilization of an algorithm is often difficult and error-prone. This is mainly due to the combinatorial nature of the problem: convergence to a legitimate configuration must be demonstrated from *every* possible initial configuration. In the same spirit, our work aims at bringing formal reasoning techniques to the design and verification of self-stabilizing algorithms. In [19], the authors propose a formal approach based on propositional Satisfiability that is complementary to ours. First, as in the present paper, they aim at bringing formal reasoning techniques to the design and verification of self-stabilizing algorithms dedicated to synchronous distributed systems. Yet, their focus is on more severe failure patterns as they specifically consider both transient and Byzantine faults[2]. However, their analysis is restricted to a fully connected network topology while our work addresses several classes of loosely connected topologies, including rings, chains and stars, where each node only accesses a very restrictive view of the system configuration.

With this in mind, we focus here on a simple problem: *Synchronous Unison*. We assume a synchronous system where processes are anonymous and whose topology is arbitrarily connected. Each process has a local clock whose integer values vary between 0 and $m-1$, where m is called the *period*. Starting from a configuration where each process's clock has a set value in $\{0,\ldots,m-1\}$, the goal is to make the system converge to a configuration for which all clocks are identical while allowing increment actions (modulo m) at each step. Arora and al have proposed a simple solution to this problem [4]. The self-stabilization of their algorithm is demonstrated for all $m \geq \max(2, 2\mathcal{D}-1)$ where $\mathcal{D}$ is the diameter of the network [3, 4]. However, the tightness of this bound remains an open question. Actually, this question is partially answered in [3] where *divergence configurations*, *i.e.*, configurations from which executions never stabilize, are exhibited for all *even* periods strictly smaller than the bound: the odd cases remaining unresolved to date.

We propose therefore to study how propositional satisfiability can help to address this question, and more generally how it can help to verify the self-stabilization of distributed algorithms. Indeed, Propositional Satisfiability (SAT) is a fundamental problem in logic and symbolic artificial intelligence, which consists in determining whether a given propositional formula in Conjunctive Normal Form (CNF) can be satisfied by an assignment of the variables. This problem, which was the first proven NP-complete [15], is attracting growing interest particularly due to the remarkable efficiency of modern SAT solvers in handling large instances despite its established difficulty. In addition to its key role in software verification, artificial intelligence, and cryptography, SAT lies at the intersection of several disciplines such as logic, graph theory, computer science, and artificial intelligence [6].

In this paper, we introduce a formal SAT-based approach to rigorously analyze the behavior of Arora and al's synchronous unison algorithm [4]. Clock update rules are translated into logical constraints in CNF, allowing the use of SAT solvers to verify whether synchronization

[1] A transient fault occurs at an unpredictable time, but does not result in a permanent hardware damage. Moreover, as opposed to intermittent faults, the frequency of transient faults is considered to be low.
[2] A Byzantine fault refers to an arbitrary behavior of a node that may no longer follow its local algorithm.

can be achieved. This methodology provides a systematic framework for studying the stabilization of the algorithm with different topologies and initial configurations, and with different periods. To optimize this analysis, specific techniques are applied to reduce redundancies among initial configurations, thereby accelerating model efficiency. Our approach, although focused on unison, provides formal guarantees regarding synchronization scenarios, while opening up optimization opportunities in more complex distributed algorithms.

The remainder of this paper is organized as follows. In Section 2, we present the synchronous unison algorithm and introduce propositional satisfiability. Section 3 is devoted to the formal modeling of the synchronous unison algorithm of Arora and al [4], where we detail the variables and notations, as well as the formulation of clock uniqueness, update, and detection of self-stabilization or its absence. This section also presents additional constraints that aim at accelerating the solving of the problem. Finally, Section 5 presents our experimental evaluation on different graph topologies. Finally, we conclude and discuss future work in Section 6.

2 Preliminaries

2.1 Synchronous Unison

We consider a distributed system of n interconnected processes. The interconnection network is modeled by an undirected connected graph $G = (V, E)$ where V is a set of vertices representing the processes and E a set of edges representing bidirectional communication links between processes. The set of neighbors of a process p will be denoted $N(p)$. Arora and al's synchronous unison [4] is showcased in Algorithm 1. It is written in the *atomic-state model*, a locally shared memory model of computation where each node can directly read its state and those of its neighbors, but can only modify its own state.

Given an integer parameter m common to all processes and called *period*, each process p holds a single variable $p.c \in \{0, \ldots, m-1\}$, which represents its *clock*. The state of a process is then its clock value. The *configuration* of the system is modeled by the vector associating each process with its state (its clock value). The set of possible configurations of the system is denoted by $\Gamma_{n,m} = \{0, \ldots, m-1\}^n$. The local algorithm of each process p is made of a single rule of the form $guard \mapsto action$. The guard of the rule of p is a predicate on the state of p and those of its neighbors. The action part modifies the state of p and, in this case, is thus an assignment of the clock of p. In a given configuration, a process's rule is said to be *enabled* if its guard is true. A configuration where no rule can be enabled is said to be *terminal*. The system is assumed to be synchronous: as long as the system is not in a terminal configuration, iterations are executed as follows. All processes whose rule can be enabled in the current configuration γ_i simultaneously execute their rule's action, thus generating a new configuration γ_{i+1}, and so on. An execution of the algorithm is therefore a sequence of configurations $\gamma_0, \ldots, \gamma_i, \ldots$ such that for all $i > 0$, γ_i is obtained from γ_{i-1} in one synchronous iteration of the algorithm.

The goal of the unison algorithm is to reach a configuration where all clocks are synchronized to an identical value. To this end, at each iteration, each process p computes the minimum clock value $c_{\min}$ in its closed neighborhood (*i.e.*, $N(p) \cup \{p\}$), and then modifies, if necessary, its clock $p.c$ to $(c_{\min} + 1) \bmod m$ (see Algorithm 1). Gradually, the smallest values propagate in the network. This progress can only be disturbed by the appearance of 0 at some processes after their entire closed neighborhood has reached the maximum value $m-1$. However, Arora *et al.* have shown that when m is chosen sufficiently large (*i.e.*, $m \geq \max(2.2\mathcal{D} - 1)$ where $\mathcal{D}$ is the diameter of G), these local resets cannot prevent the

system from converging: regardless of the initial configuration γ_0, there necessarily exists a value $t \geq 0$ such that all clocks have the same value in γ_t. Conversely, if m is chosen too small, the execution may never converge. In this case, there exists a cycle between *illegitimate* configurations (*i.e.*, configurations containing at least two different clock values).

Since the total number of possible configurations is m^n, if the system is still in an illegitimate configuration after $m^n - 1$ iterations, we can claim that the execution will never converge. Thus, it is sufficient to observe the first $t_f = m^n$ configurations of an execution in order to decide whether the execution converges or diverges. This number is very large and can lead to exponential modeling. We can therefore choose a more reasonable value for t_f, even if it means not being able to decide in some cases.

▶ **Example 1.** Consider a process network G=$(\{p_0, p_1, .., p_5\}, E)$ organized as a ring topology. Let $m = 5$ and assume the initial configuration γ_0 given in Figure 1. From γ_0, we obtain the execution prefix $\gamma_0, \gamma_1\gamma_2, \gamma_3$ given in Table 1. The unison algorithm therefore reaches a legitimate configuration after 3 iterations.

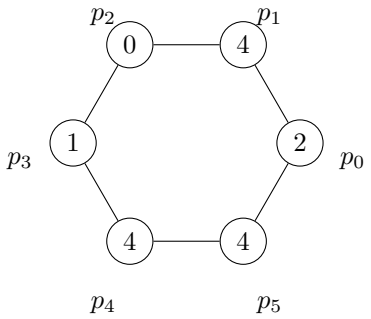


Figure 1 Initial configuration γ_0 in a ring of 6 nodes with $m = 5$.

Table 1 Prefix of 4 configurations obtained after 3 iterations from γ_0.

	p_0	p_1	p_2	p_3	p_4	p_5
γ_0	2	4	0	1	4	4
γ_1	3	1	1	1	2	3
γ_2	2	2	2	2	2	3
γ_3	3	3	3	3	3	3

In the previous example, a legitimate configuration, where the clocks are synchronized, is eventually reached by the algorithm. However, when m is chosen too small, there may exist configurations from which the clocks never synchronize. In this case, we will say that the system *diverges*. The convergence and divergence properties, formally defined below, highlight the importance of studying the behavior of distributed algorithms, and particularly synchronous unison, according to their parameter values (in this case, the period) [16, 20].

Algorithm 1 Synchronous unison for a process p.

Inputs :

$N(p)$: set of the neighbors of p

m : a positive integer, the period

Variable:

$p.c \in \{0, .., m-1\}$, the clock of p

Macro :

$New(p) = (\min(\{q.c : q \in N(p)\} \cup \{p.c\}) + 1) \bmod m$

Rule :

$p.c \neq New(p) \mapsto p.c := New(p)$

▶ **Definition 2** (Convergence). *Given a period m, the synchronous unison algorithm executed on a graph G of n vertices converges when, for every initial configuration $\gamma_0 \in \Gamma_{n,m}$, the system reaches, within a finite number of iterations, a configuration $\gamma_t \in \Gamma_{n,m}$ where all clocks have the same value.*

▶ **Definition 3** (Divergence). *The synchronous unison algorithm diverges for a given period m in a graph G of n vertices when there exists an initial configuration $\gamma_0 \in \Gamma_{n,m}$ from which the clocks never synchronize, i.e., there are infinitely many reached configurations containing at least two distinct clock values.*

▶ **Example 4.** Consider a network of $n = 3$ processes whose topology is a chain. Let $m = 2$ and consider the initial configuration γ_0 given in Figure 2. From this initial configuration, we obtain the execution prefix described in Table 2. Divergence is observable since the the configuration γ_2 is identical to the initial one, γ_0, and both are illegitimate.

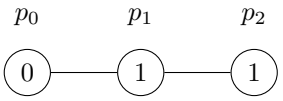


Figure 2 Initial configuration γ_0 in a chain of 3 nodes, where $m = 2$.

Table 2 Two iterations of the synchronous unison in a chain of three nodes, where $m = 2$.

	p_0	p_1	p_2
γ_0	0	1	1
γ_1	1	1	0
γ_2	0	1	1

Our analysis will focus on the impact of graph topology and the period m on synchronization. The following theorem provides an interesting general bound on the maximum number of iterations in a connected graph.

▶ **Theorem 5** ([3]). *The synchronous unison algorithm is self-stabilizing in every connected graph of diameter $\mathcal{D}$ for $m \geq \max\{2, 2\mathcal{D} - 1\}$. Its stabilization time, in this case, is at most $3\mathcal{D} - 2$ iterations.*

Finally, we stress out that we have carefully selected the synchronous unison as our case study with the goal of generalization to a broader class of distributed algorithms in mind. More precisely, we deal with the atomic-state model because it is the most commonly used model in the self-stabilizing literature and it is also close to other important models widely used in distributed computing, e.g., the local and congest models [23, 29]. The atomic-state model is a locally shared memory model so our approach can be straightforwardly generalized to handle algorithms working in systems where communication is made using communication registers. Furthermore, Synchronous Unison is representative of classical techniques used in self-stabilization. For example, the way clocks are updated follows the so-called "$min + 1$" rule that is used in many self-stabilizing algorithms such as the Breadth-First Search (BFS) spanning tree construction [22] and the asynchronous version of the unison [16]. More generally, we believe that our guidelines to model the transition relations are very general while issues related to symmetries that we tackle in the paper are also pretty usual in self-stabilization.

2.2 Propositional Satisfiability

Let X be a set of propositional variables. A literal is a variable $x \in X$ or its negation $\overline{x}$. A clause is a disjunction of literals. A formula in Conjunctive Normal Form (CNF) is a conjunction of clauses. An assignment $I : X \to \{\text{True}, \text{False}\}$ associates each variable with a

Boolean value and can be represented as a set of literals. A literal is satisfied by an assignment I if $l \in I$, otherwise it is falsified by I. A clause is satisfied by an assignment I if at least one of its literals is satisfied by I, otherwise it is falsified by I. A CNF formula is satisfiable if there exists an assignment that satisfies all its clauses, otherwise it is unsatisfiable. The satisfiability problem (SAT) thus consists in determining whether a given CNF formula is satisfiable. Although SAT is NP-complete [15], modern solvers based on the Conflict Driven Clause Learning (CDCL) [30] algorithm are efficient and can solve instances involving a large number of variables and clauses. Indeed, beyond clause learning and non-chronological backtracking, these solvers integrate powerful mechanisms such as lazy structures, dedicated branching heuristics, and restarts [10]. Moreover, the SAT problem is widely used to model and solve many complex combinatorial problems from various fields, notably in formal verification [8, 14], but also in cryptography, bioinformatics and planning, among many others [10].

Logic-based model checking has been a cornerstone in the formal verification of algorithms and systems. In particular, Bounded Model Checking (BMC) [7], which verifies whether a finite-state model satisfies a given specification within a bounded execution length, has gained significant traction over the past decade, especially when encoded as satisfiability (SAT) or satisfiability modulo theories (SMT) problems. Early efforts in the formal verification of distributed self-stabilizing algorithms were led by Lakhnech and Siegel [27,33], who introduced theoretical frameworks for computer-aided verification, although no practical toolsets emerged from their work. Subsequent research explored symbolic BMC for distributed algorithms such as the work of Chen and Kulkarni who applied SMT-based BMC to assess the stabilization properties of distributed algorithms, particularly Dijkstra's token ring, Ghosh's Mutual Exclusion and a version tree-based mutual exclusion [13]. However, while SMT solvers offer expressive modeling capabilities, their higher-level abstraction can hinder fine-grained analysis and scalability. More recently, a SAT-based formal approach was proposed in [19] to detect Byzantine faults in synchronous systems with complete network topologies. This propositional encoding demonstrates the viability of SAT-based reasoning for distributed fault-tolerant systems and opens the door to more scalable and precise verification frameworks for self-stabilizing algorithms.

In the following sections, we thus focus on modeling the behavior of the synchronous unison algorithm, applied to an arbitrarily connected graph. The objective is to apply bounded model checking through SAT to characterize the conditions of convergence or divergence of the algorithm. This involves modeling the states of the processes using logical formulas, which must be expressed in such a way that they can be effectively solved by a SAT solver. Finally, we mention that previous literature introduces cardinality constraints having the form $\sum_i l_i = 1$ where each l_i is a literal, and which can be efficiently encoded into CNF form [32].

3 Formal Modeling of Synchronous Unison

In this section, we focus on the formal modeling of the behavior of the synchronous unison of Arora *et al.* through propositional satisfiability. We begin by introducing the model variables. Then, we define the constraints that govern the execution of the algorithm. These constraints must faithfully represent the evolution of the system, and also allow us to detect its convergence or divergence. Furthermore, we will introduce specific constraints that allow us to optimize the model by eliminating redundant initial configurations.

3.1 Variables and Notations

We recall that the network is modeled by an undirected connected graph $G = (V, E)$ where $V = \{0, \ldots, n-1\}$ is a set of n vertices representing the processes and E is a set of edges representing bidirectional communication links between pairs of processes. $M = \{0, \ldots, m-1\}$ denotes the set of possible clock values, where m is the period. We are interested in the first t_f configurations of any execution of the algorithm on G. The configurations will thus be indexed from 0 to $t_f - 1$ and the set of indices will be denoted $T = \{0, \ldots, t_f - 1\}$. To model the execution of the algorithm, it is also necessary to reason on the state of the process clocks in the different configurations. To do this, we introduce the clock variables $h_{p,t,v}$, for all $(p, t, v) \in V \times T \times M$. Each variable $h_{p,t,v}$ will indicate whether the clock of process p is equal to the value v in configuration t. This is sufficient to model execution and detect convergence. However, modeling divergence requires other variables, in particular to represent the presence of a cycle between illegitimate configurations during execution and to simplify the transformation of constraints into conjunctive normal form. The set of variables we will use is detailed below:

- **Clock variables:** $h_{p,t,v}$ where $(p, t, v) \in V \times T \times M$. These variables are assigned to $True$ if process p has clock value v in configuration t, $False$ otherwise.
- **Cycle variables:** c_t where $t \in T$. These variables are assigned to $True$ if the configuration $t \in T$ and the initial configuration are identical, *i.e.*, if the clock value of each process is identical in both configurations.
- **Clock-Similarity variables:** $s_{p,t,v}$ where $(p, t, v) \in V \times T \times M$. These variables are assigned to True if process p has the same clock value in configuration $t \in T$ and the initial configuration.

3.2 Modeling Unison Execution

First, we focus on formalizing the constraints modeling a valid execution of synchronous unison. More specifically, we define clock uniqueness constraints as well as update rules ensuring the correct computation of clock values at each iteration.

3.2.1 Clock Uniqueness

To guarantee a valid execution of the algorithm, it is necessary to maintain the uniqueness of each process's clock at each iteration. Specifically, it is necessary to ensure that each process has one and only one clock value at each configuration of the algorithm. This constraint can be modeled using the following cardinality constraint:

$$\sum_{v \in M} h_{p,t,v} = 1 \quad \forall p \in V, \forall t \in T \tag{1}$$

Note that this constraint can be easily rewritten in CNF form, using a pairwise encoding, without addition of auxiliary variables as shown below. The constraint (1a) ensures that a process cannot simultaneously hold two different clock values in a given configuration, while the constraint (1b) ensures that each process holds at least one valid clock value in each configuration. The number of clauses induced by these constraints is thus $O(n * t_f * m^2)$ but more efficient encodings can be used if we allow the use of auxiliary variables [31].

$$\bigwedge_{\substack{(v,v')\in M^2 \\ v<v'}} (\overline{h_{i,t,v}} \vee \overline{h_{p,t,v'}}) \quad \forall p \in V, \forall t \in T \tag{1a}$$

$$\bigvee_{v\in M} h_{p,t,v} \quad \forall p \in V, \forall t \in T \tag{1b}$$

3.2.2 Clock Update

The process clocks are updated, at each iteration, based on the clock values of their closed neighborhood. More precisely, each process p computes the minimum values among its own clock value and those of its neighbors. This value will then be incremented by 1 modulo m. Finally, the obtained value v is assigned to $p.c$ if $p.c \neq v$. To simplify the notations, we denote $\tilde{N}(p) = N(p) \cup \{p\}$ the closed neighborhood of the process $p \in V$ and $d_p = |\tilde{N}(p)| = |N(p)|+1$ its size. The clock update constraint is then modeled as follows:

$$\bigwedge_{\substack{(v_{p'})_{p'\in\tilde{N}(p)}\in M^{d_p} \\ v'=(\min_{p'\in\tilde{N}(p)} v_{p'}+1) \bmod m}} \left(\bigvee_{p'\in\tilde{N}(p)} \overline{h_{p',t,v_{p'}}} \right) \vee h_{p,t+1,v'}$$

$$\forall t \in T \setminus \{t_f - 1\}, \forall p \in V \tag{2}$$

Indeed, for each possible assignment $(v_{p'})_{p'\in\tilde{N}(p)}$ of the clock values of the closed neighborhood of a process p, we want to ensure that the minimum value incremented by 1 modulo m, denoted v', is propagated to the process p at the following configuration, *i.e.*, $\bigwedge_{p'\in\tilde{N}(p)} h_{p',t,v_{p'}} \rightarrow h_{p,t+1,v'}$. This last implication perfectly highlights the clock update mechanism in the context of synchronous unison and is rewritten in conjunctive normal form in the constraint (2) by adding the necessary iterations on the possible assignments and the computation of the new value of the clock at the conjunction level. To provide a refined analysis of the encoding complexity, we denote $d_{\max} = \max_{p\in V}(d_p - 1)$ the maximum degree of the graph. Thus, it is clear that the number of clauses is bounded by $O(n * t_f * \times m^{d_{\max}+1})$. Therefore, it seems that the complexity of our model is strongly correlated to the maximum degree in the graph and thus it is more suited to graphs in which this value remains relatively small. In particular, for ring or chain graphs, the complexity will be $O(n * t_f * \times m^3)$ since, for these topologies, we have $d_{\max} = 2$.

3.3 Modeling Convergence

Convergence occurs when the system has reached a legitimate configuration, where all clock values are identical. An algorithm is said to be convergent if, starting from any initial configuration, such a legitimate configuration is always reached within a finite number of iterations. To demonstrate this property, we will reason by contradiction by assuming that convergence is never reached from some initial configuration. In other words, there exists (at least) one initial configuration from which no legitimate configuration can be obtained. Since a legitimate configuration necessarily generates another legitimate configuration after an iteration of the algorithm, it suffices to show that the last configuration $t_f - 1$ is illegitimate for some initial configuration. This property is thus translated by the following constraint:

$$\bigwedge_{v\in M} \bigvee_{p\in V} \overline{h_{p,t_f-1,v}} \tag{3}$$

Thus, it is clear that the CNF formula ϕ obtained thanks to the constraints (1), (2), and (3) is unsatisfiable if and only if the synchronous unison algorithm converges. Indeed, if ϕ is satisfiable, the provided model allows to exhibit an initial configuration $\lambda_0 \in \Gamma_{n,m}$ deduced from the clock variables $h_{p,0,v}$ which are set to true for $p \in V$ and $v \in M$, from which the algorithm does not converge. More specifically, in the valid execution of the unison algorithm from λ_0 (ensured by the constraints (1) and (2)), we exhibit, at the level of the last configuration λ_{t_f-1} and for any possible clock value $v \in M$, a process $p \in V$ whose clock does not hold the value v in λ_{t_f-1} (ensured by constraint (3)). In terms of complexity, since constraint (3) requires only $O(m)$ clauses, our model for convergence requires only $O(n \times t_f \times m)$ variables with a number of clauses bounded by $O(n * t_f * m^{d_{\max}+1})$ where $d_{\max}$ is the maximum degree of the graph. We also note that the maximum clause size of the model is $\max(m, n+2)$.

Additionally, we can notice that constraint (3) can be further imposed on any subset of configurations in $T' \subseteq T \setminus \{t_f - 1\}$ as showcased in (4). Adding these redundant constraints does not alter the overall complexity of the model but may accelerate the detection of convergence. Specifically, these constraints can enable the solver to identify a valid configuration without requiring all necessary propagations to simulate the system execution until the final iteration from a given initial configuration. Consequently, if we consider the set $X = T \setminus \{t_f - 1\}$, the model will ensure that the presence of a legitimate configuration is verified at each iteration of the algorithm whereas the set $P = \{0 \leq t < t_f - 1 \mid \exists q \in \mathbb{N} \text{ s.t } t = t_f - 1 - q.m\} \cup \{0\}$ guarantees that this verification occurs every m iterations, while necessarily including the initial configuration. We will thus refer to the following constraint as IC (for iterative convergence) and we denote $IC_{T'}$ the constraint with the corresponding set $T' \in \{P, X\}$.

$$\bigwedge_{v \in M} \bigvee_{p \in V} \overline{h_{p,t,v}} \quad \forall t \in T' \tag{4}$$

3.4 Modeling Divergence

Divergence implies the existence of two identical illegitimate configurations in the execution. Indeed, the set of legitimate configurations is trivially closed, and since the algorithm is synchronous and deterministic, such a cycle repeats itself infinitely. Let two configurations $(t_1, t_2) \in T$ such that $t_1 < t_2$, where the clock values of the processes at t_2 are identical to those at t_1. We will call such an occurrence a *cycle* at the level of configurations (t_1, t_2). To establish the divergence, we must therefore model the existence of a cycle at the level of two possible illegitimate configurations reached by the algorithm. To simplify the modeling, we consider the existence of a cycle with respect to the initial configuration, *i.e.*, $t_1 = 0$. Indeed, even if $t_1 > 0$, it is always possible to reduce the analysis to the initial configuration by taking the configuration in t_1 as the initial configuration of the system (*n.b.*, a self-stabilizing algorithm must converge from any initial configuration). Furthermore, to ensure that the cycle configurations are illegitimate, it is sufficient to check that the initial configuration is illegitimate. We recall that the variables c_t where $t \in T$, defined previously, represent the existence of a cycle at configuration t. Thus, the existence of a cycle between illegitimate configurations can be enforced by the following constraints:

$$\bigvee_{\substack{t \in T \\ 0 < t}} c_t \tag{5a}$$

$$\bigwedge_{v \in M} \bigvee_{p \in V} \overline{h_{p,0,v}} \tag{5b}$$

Clearly, the constraint (5a) models the existence of a cycle $(0, t)$ where $t \in T$; while the constraint (5b) guarantees that configuration 0, and therefore the configurations of the cycle, are illegitimate. To establish the semantic meaning of the cycle variables, it is necessary to guarantee that, if these variables are set to true, then there is a cycle $(0, t)$, *i.e.*, the clock values in configuration t are identical to those in the initial configuration. In this context, the clock similarity variables $s_{p,t,v}$ where $(p, t, v) \in V \times T \times M$ can play the intermediate role of guaranteeing the same clock value in configuration $t \in T$ and the initial configuration. Thus, we add the following constraints:

$$\overline{c_t} \vee \bigvee_{v \in M} s_{p,t,v} \quad \forall t \in T, \forall p \in V \tag{6a}$$

$$(\overline{s_{p,t,v}} \vee h_{p,0,v}) \wedge (\overline{s_{p,t,v}} \vee h_{p,t,v}) \quad \forall t \in T, \forall p \in V, \forall v \in M \tag{6b}$$

The constraint (6a) thus ensures that the corresponding configuration $t \in T$ has the same clock values as configuration 0 or, more formally, that $c_t \rightarrow \bigwedge_{p \in V} \bigvee_{v \in M} s_{p,t,v}$. The normal form presented in the constraint (6a) can be simply obtained by the clausal rewriting of the implication as well as the distributivity of disjunctions over logical conjunctions. The constraint (6b) is necessary to establish the semantic meaning of the variables $s_{p,t,v}$ which are set to $True$ when the process $p \in V$ has the same clock value $v \in V$ at configurations 0 and t. More formally, we have $s_{p,t,v} \rightarrow h_{p,0,v} \wedge h_{p,t,v}$ and its normal form presented in the constraint (6b) can also be deduced by rewriting the implication and applying the distributivity of disjunction.

It is clear that the CNF formula ψ obtained by the constraints (1), (2), (5) and (6) is satisfiable if and only if the unison algorithm diverges on the given input network with period m. Indeed, a model of ψ allows to exhibit an initial configuration $\lambda_0 \in \Gamma_{n,m}$ from which a valid execution of the synchronous unison generates a cycle $(0, t)$ of illegitimate configurations, given by a cycle variable assigned to $True$ at the level of the clausal constraint (5a). Moreover, in terms of encoding complexity for divergence, it is clear that the constraints (5) and (6) introduce a bounded number of clauses in $O(n \times t_f \times m)$ and, consequently, the model has a complexity similar to the convergence case, mainly governed by the constraint (2), *i.e.*, in $O(n * t_f * m^{d_{\max}+1})$ where $d_{\max}$ is the maximum degree of the graph. However, the model requires a larger number of variables, but which remains in $O(n \times t_f \times m)$.

3.5 Elimination of Initial Configurations

In this section, we aim at reducing the number of initial configurations to be processed by removing redundant configurations. In other words, the goal is to eliminate configurations for which the algorithm's behavior is identical to that of other configurations we already consider in the analysis. This approach optimizes the algorithm's execution by avoiding processing situations that, although different in their representation, do not lead to different results.

3.5.1 Lexicographic Order

Within the framework of the unison algorithm, the evolution of process clock values depends exclusively on their local neighborhood and, more specifically, on their closed neighborhood. In particular, two processes with the same set of neighbors in the studied topology evolve identically, up to any permutation of their initial clock values. This observation makes it possible to exploit a structural symmetry to reduce the space of configurations to be considered by establishing a specific lexicographic order on the initial clock values for processes

with identical neighborhoods in the topology. To take advantage of this symmetry and avoid examining multiple equivalent configurations, we introduce constraints imposing a lexicographic order on the configurations. More precisely, only the minimal configurations according to this order are considered, which eliminates redundant configurations and reduces the search space. In our case, we will introduce constraints prohibiting lexicographically larger configurations on local neighborhoods identical to those selected. This constraint, which we will denote LO, can be formulated as follows:

$$\bigwedge_{\substack{(p,p')\in V^2 \\ p<p', \\ N(p)=N(p')}} \bigwedge_{\substack{(v,v')\in M^2 \\ v'<v}} (\overline{x_{p,0,v}} \vee \overline{x_{p',0,v'}}) \tag{7}$$

Thus, if two processes $(p, p') \in V^2$ such that $p < p'$ have an identical neighborhood, the constraint (7) guarantees that p' cannot take an initial clock value smaller than that of p. This constraint introduces a number of clauses bounded in $O(n^2 \times m^2)$ and can only be applied in topologies where some processes have an identical neighborhood, such as in the star topology, where a central process (hub) is connected to several peripheral processes (see Figure 5), or in complete graphs.

3.5.2 Rotation Elimination

As previously discussed, when a transformation of an initial configuration does not modify the overall behavior of the algorithm, it is possible to eliminate redundant configurations to optimize the search and analysis of relevant cases. A particularly interesting case concerns graphs, such as rings, where the same configuration can be expressed in several equivalent forms via a simple rotation of the process indices. In these structures, any initial configuration can be transformed into another configuration by uniformly shifting the clock values. Since the rotation does not modify either the neighbor relationships or the algorithm dynamics, the study can be restricted to a representative subset of possible configurations.

To enforce this restriction, a constraint can be introduced guaranteeing that, among all equivalent configurations, only those where a specific process has the lowest clock value are considered. In particular, in a ring graph of n processes, we consider a configuration where the first process has the lowest clock value among the clock values of the other processes in the same configuration. Since the rotation, whether in the direct or indirect direction, preserves the same behavior of the synchronous unison algorithm, we can eliminate any other initial configuration that is an image of this configuration in order to avoid the analysis of redundant cases, as proven in Proposition 6, with $\lambda[p]$ denoting the clock value of $p \in V$ in the configuration $\lambda \in \Gamma_{n,m}$. This constraint, which we will denote RE, can therefore be expressed as follows on any ring topology:

$$\bigwedge_{\substack{(v,v')\in M^2 \\ v<v'}} \bigwedge_{p\in V\setminus\{0\}} (\overline{x_{0,0,v'}} \vee \overline{x_{p,0,v}}) \tag{8}$$

▶ **Proposition 6.** *Let $G = (V, E)$ be a ring graph composed of n processes and m a period. Let $ER = \{\lambda \in \Gamma_{n,m} \mid \exists p \in V \setminus \{0\}, \lambda[p] < \lambda[0]\}$. Then, for any initial configuration $\lambda_0 \in ER$, there exists a configuration $\lambda'_0 \in \Gamma_{n,m} \setminus ER$ from which a synchronous unison execution produces the same convergence or divergence behavior.*

Proof. Without loss of generality, we assume that $V = \{0, \ldots, n-1\}$. Let $\lambda_0 \in ER$ and $p_{\min} = \mathsf{argmin}_{p \in V} \lambda_0[p]$. We define the configuration λ'_0 such that:

$$\lambda'_0[p] = \begin{cases} \lambda[p - p_{\min}] & \text{for } p_{\min} \leq p \leq n-1, \\ \lambda[n - p_{\min} + p] & \text{for } 0 \leq p < p_{\min}. \end{cases}$$

Clearly, λ'_0 is obtained by a rotation of λ_0 that brings the smallest value back to process 0. Therefore, λ'_0 has the same behavior as λ, up to a rotation. Indeed, following an execution of the synchronous unison algorithm in a ring graph, the relations between neighbors are preserved under rotation and each process applies exactly the same evolution rules as its representative in the original configuration. Moreover, since $\lambda'_0[0] = \lambda_0[p_{\min}] = \min_{p \in V} \lambda[p]$, there is no process $p \in V \setminus \{0\}$ such that $\lambda[p] < \lambda[0]$ and, therefore, λ'_0 belongs to the kernel $\Gamma_{n,m} \setminus ER$. ◀

The constraint (8) introduces a number of clauses bounded in $O(n \times m^2)$ and can be applied to graphs whose topology allows for specific rotations while preserving neighbor relationships such as rings, or star topologies by applying the constraint only to the peripheral nodes.

4 Experimental Evaluation

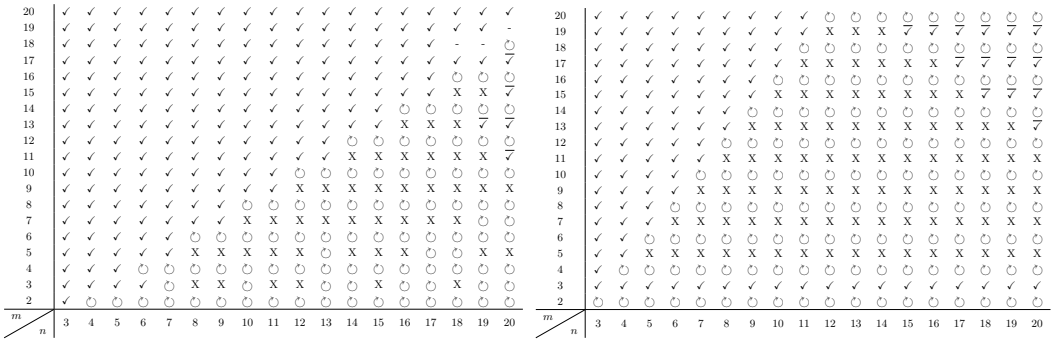
4.1 Experimental Protocol

We consider the following classical network topologies: chains, rings, and stars (see Figures 1, 2, and 5 respectively) [34]. The properties of the considered graphs and periods are summarized in Table 3. We encoded our models in Python using the PySAT library [25] [3] and solved these instances using the state-of-the-art solvers Cadical [9] and MapleSAT [28] [4]. Tests were performed on a machine equipped with an Intel Core i7 processor clocked at 3.80 GHz, running under Ubuntu 22.04. A timeout of 7200 seconds was set for each instance. We have generated the instances associated with all our models, totaling 4968 instances. For each type of topology among rings and chains, we took graph sizes n ranging from 3 to 20 nodes, and we varied the period m from 2 to 20. For the star topology, we took graph sizes n ranging from 3 to 10 nodes, and we varied the period m from 2 to 10. In particular, we generated 756 instances (342 for chains and rings and 72 for stars) for the initial model of convergence (INI_{CNV}) which is described by the constraints (1), (2) and (3). Similarly we generated the same number of instances for the divergence model (INI_{DIV}) described through the constraints (1), (2), (5) and (6).

Furthermore, for topologies where initial configuration elimination constraints can be applied, we also generate the corresponding instances. Thus, we generated 828 instances of convergence and divergence for ring and star topologies with the rotation elimination (RE) constraint. For the lexicographic order (LO) constraint, we generated 144 instances of convergence and divergence for the star topology. Moreover, we generated 756 instances of convergence for ring, chain, and star topologies for each constraint IC_P and IC_X, added on top of the initial models. 414 instances of convergence were generated for ring and star topologies with each combinations $RE + IC_P$ and $RE + IC_X$. Finally, we generated 72

[3] Our code and benchmark are available in the following **GitHub repository**

[4] For lack of space, we present the results with Cadical and we provide a comparison on the initial models of convergence and divergence in the appendix (refer to Tables 10, 9 and 8)



(a) Rings. **(b)** Chains.

Figure 3 Convergence and divergence results of synchronous unison on rings and chains, for different graph sizes n and periods m. The signs (✓), (↻), ($\overline{✓}$), ($\overline{↻}$), (X) and (−) respectively indicate that the algorithm converges, diverges, does not converge in the allocated number of configurations, does not diverge in the allocated number of configurations, neither converge nor diverge in the allocated number of configurations, or that both convergence and divergence instances were not solved within the allocated time.

instances of convergence for star topology for each combination $LO + IC_P$ and $LO + IC_X$. The number of allowed configurations is set to $t_f = 3\mathcal{D}$ for all topologies where $\mathcal{D}$ is the diameter of the considered graph. We also note that the Cardinality Network encoding [5] was used to rewrite constraint (1) in CNF form.

4.2 Analysis of Convergence and Divergence

The results presented in Figures 3(a), 3(b), and 4 illustrate the behavior of the synchronous unison in terms of convergence and divergence on the different topologies (rings, chains, and stars) as a function of size n and period m. For the studied graph topologies, we clearly notice that the results are coherent with Theorem 5 when $m \geq \max\{2, 2\mathcal{D} - 1\}$. revealing a convergence for the corresponding instances. Notably, the generated results demonstrate unison behavior on many instances across different topologies for which it was previously unknown. In particular, we were able to detect the convergence of the algorithm for the three topologies for values of $m < \max\{2, 2\mathcal{D} - 1\}$. However, some instances remain unsolved, most likely due to the chosen upper limit on the number of allowed steps (*i.e.*, $3\mathcal{D}$). It is also interesting to note that our results show that the bound proposed in the theorem 5 is tight in the case of stars. Indeed, this theorem states that convergence is guaranteed in stars (of diameter 2) from a period m greater than or equal to 3 ($\max(2, 2 \times 2 - 1) = 3$). Our results exhibit a case of divergence with $m = 2$ in star graphs as shown in Figure 4, independently of their sizes. Thus, the synchronous unison algorithm converges in a star if and only if $m \geq 3$. We demonstrate this property in following proposition.

Table 3 Properties of the considered graphs. For each topology, we report the considered number of nodes n, the chosen periods m, the diameter $\mathcal{D}$, and the maximal degree $d_{\max}$.

Topology	n	m	$\mathcal{D}$	$d_{\max}$
Chain	$\{3, \ldots, 20\}$	$\{2, \ldots, 20\}$	$n - 1$	2
Ring	$\{3, \ldots, 20\}$	$\{2, \ldots, 20\}$	$\lfloor n/2 \rfloor$	2
Star	$\{3, \ldots, 10\}$	$\{2, \ldots, 10\}$	2	$n - 1$

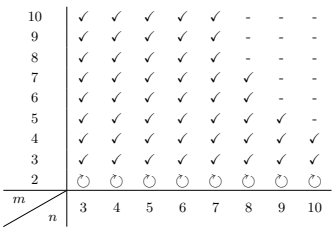


Figure 4 Convergence and divergence results of synchronous unison on stars, for different graph sizes n and periods m.

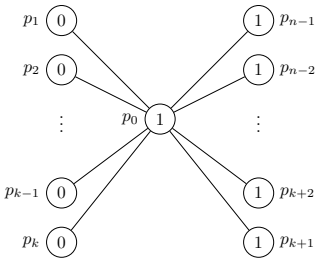


Figure 5 Initial configuration ensuring divergence on a star graph with n nodes and for period to $m = 2$.

Table 4 Two iterations of the synchronous unison in a chain of three nodes, where $m = 2$.

	p_0	p_1	...	p_k	p_{k+1}	...	p_{n-1}
γ_0	1	0	...	0	1	...	1
γ_1	0	1	...	1	0	...	0
γ_2	1	0	...	0	1	...	1

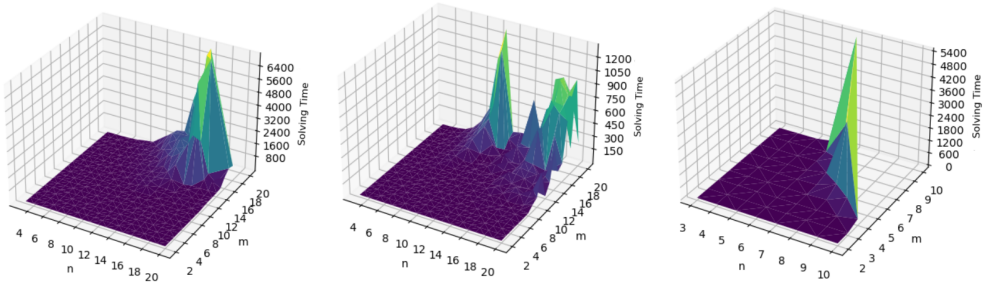
▶ **Proposition 7.** *The synchronous unison algorithm diverges for any star graph, regardless of its number of nodes $n \geq 3$, with period $m = 2$.*

Proof. We can always exhibit an initial configuration $\gamma_0 \in \Gamma_{n,m}$ that causes the synchronous unison to diverge. Such a configuration is illustrated in Figure 1, where the initial clock value of the central node is set to 1, while the clock values of the peripheral nodes take at least two distinct initial values. The prefix of execution until a cycle is detected is represented in Table 4 and therefore the algorithm diverges on star graphs when $m = 2$. ◀

4.3 Performance Analysis

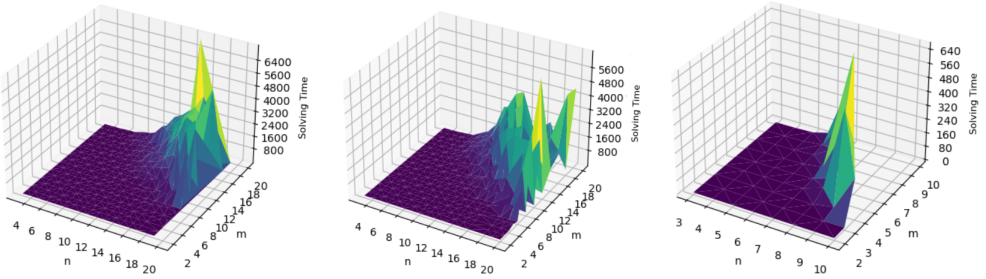
In this section, we analyze the performance of the models on the different topologies in terms of number of solved instances and average solving time per graph size. The results obtained for ring and star graphs are presented respectively in Tables 5 and 7, with respect to the graph sizes, the analyzed property, and the considered model. Due to the lack of space, the results obtained for chain graphs have been moved to the appendix (refer to Table 6). The results indicate that the initial model manages to solve most of the instances but it clearly encounters increasing difficulties with increasing graph sizes, particularly for the star and ring topologies. It thus manages to solve 329 (resp. 321) instances out of 342 for the convergence (resp. divergence) check on rings and 58 (resp. 59) instances out of 72 for the convergence (resp. divergence) check on stars. For chains, all instances were solved for convergence with the initial model whereas 328 out of 342 were solved for divergence. This behavior is not surprising especially for stars, since the star topology has a maximum degree $d_{\max} = n - 1$, which leads to a combinatorial explosion in the number of clauses generated by the constraint (2). Indeed, it increases with a complexity of the order $O(n \times t_f \times m^n)$, which explains the model's difficulty in handling larger instances. Figures 6 (resp. 7) illustrates the evolution of the solving time of instances generated for convergence (resp. divergence) verification for the initial model, as a function of graph size n and period m, for ring, chain and star topologies. These curves highlight a progressive increase in resolution time as these two parameters increase. We observe in particular that, in the case of star graphs, this increase is much more pronounced for relatively smaller values of n and m compared to the other topologies. This trend, which remains similar both for convergence and divergence corroborates our previous observations and confirms the increased complexity of the model for topologies whose maximal node degree is high.

Next, we focus on comparing the initial convergence model (INI_{CNV}) with its counterparts augmented with the IC constraint, specifically IC_P and IC_X, across ring, star, and chain topologies. In the ring topology (Table 5), IC_X consistently outperforms IC_P and INI_{CNV}



(a) Rings. (b) Chains. (c) Stars.

Figure 6 Solving time in seconds for convergence (INI_{CNV}) w.r.t. the nodes n and the period m.



(a) Rings. (b) Chains. (c) Stars.

Figure 7 Solving time in seconds for divergence (INI_{DIV}) w.r.t. the nodes n and the period m.

Table 5 Average solving times for detecting convergence and divergence over all periods for a given graph size (n). For a given graph size n with instances that have reached the time limit, the number of solved instances is indicated in "[]". The best results among the convergence and divergence models are marked in bold.

n	CONVERGENCE						DIVERGENCE	
	INI_{CNV}	RE	IC_P	IC_X	ER + IC_P	ER + IC_X	INI_{DIV}	RE
3	0.025	0.023	0.025	**0.022**	0.025	0.024	0.05	**0.02**
4	0.69	0.48	0.66	0.62	0.42	**0.29**	1.48	**0.33**
5	1.97	**1.59**	1.85	1.88	1.73	1.79	2.33	**0.48**
6	7.34	**1.13**	6.90	6.93	7.21	6.74	9.46	**2.71**
7	11.58	**2.07**	11.75	10.87	11.08	11.59	15.00	**4.10**
8	36.25	29.72	31.62	31.38	26.57	**25.66**	48.89	**12.15**
9	55.56	47.01	49.06	46.70	46.40	**40.84**	68.76	**18.87**
10	109.10	111.82	100.48	105.55	92.27	**88.04**	141.30	**54.98**
11	187.26	161.45	151.87	151.41	149.89	**135.45**	231.78	**82.91**
12	331.62	288.23	312.34	275.70	273.22	**263.15**	427.35	**148.67**
13	420.22	384.39	387.10	348.26	**341.03**	360.64	503.95	**258.33**
14	658.15	629.41	573.85	583.77	749.07	**494.72**	844.47	**417.39**
15	858.95	836.12	794.91	871.29	1045.89	**743.99**	1172.70	**532.67**
16	1500.84	1324.06	1279.63	1292.98	**1216.69**	1250.76	852.05 [16]	**779.99**
17	819.73 [16]	1654.83	1757.84	1446,94	1506.43 [18]	**1400.94**	1196.05 [16]	**996.15**
18	73.47 [15]	518.55 [16]	79.97 [15]	58.06 [15]	61.10[15]	**504.26 [16]**	560.75 [15]	**846.22 [17]**
19	**44.35 [15]**	116.42 [15]	77.05 [15]	97.72 [15]	82.12[15]	55.32[15]	1164.60 [14]	**862.39 [17]**
20	166.89 [17]	310.59 [17]	**111.46 [17]**	125.39 [17]	283.68[17]	449.56 [17]	502.73 [13]	**995.53 [17]**

Table 6 Average solving times for detecting convergence and divergence over all periods for a given graph size (n). For a given size n with instances that have reached the time limit, the number of solved instances is indicated in "[]". The best results among the convergence and divergence models are marked in bold.

n	CONVERGENCE									DIVERGENCE		
	INI_{CNV}	RE	LO	IC_P	IC_X	RE + IC_P	RE + IC_X	LO + IC_P	LO + IC_X	INI_{DIV}	RE	LO
3	0.007	0.006	**0.002**	0.004	0.004	0.004	0.003	0.004	0.004	**0.01**	**0.01**	0.02
4	0.04	0.03	**0.02**	0.045	0.04	0.034	0.03	**0.02**	**0.02**	0.09	**0.04**	0.05
5	1.00	0.86	0.31	0.83	0.95	0.94	0.59	0.31	**0.20**	0.90	0.49	**0.33**
6	27.38	19.72	**4.19**	25.06	28.37	17.41	13.75	7.10	7.14	6.97	4.48	**3.43**
7	862.27	746.24	**97.84**	868.23	828.41	634.05	730.64	102.46	696.16	76.61	55.18	**28.88**
8	553.70 [6]	**936.88 [7]**	32.60 [6]	652.89 [6]	547.38 [6]	473.89 [6]	171.55 [6]	24.19 [6]	93.25 [6]	146.91[6]	96.12 [6]	**23.75 [6]**
9	119.25 [4]	542.24 [5]	43.60 [5]	212.08[4]	139.36 [4]	185.13 [4]	184.93 [4]	42.01 [5]	**33.88 [5]**	166.68[5]	67.94 [5]	**16,79 [5]**
10	67.71 [3]	487.43 [4]	40.21 [4]	86.60 [3]	75.33 [3]	22.51[3]	598.80 [4]	**39.14 [4]**	68.56 [4]	76.16[3]	432.92 [4]	**52.76 [4]**

in larger instances, achieving the lowest solving times (-10,21% w.r.t INI_{CNV}) and highest number of solved instances (+3 w.r.t INI_{CNV}). For the star topology (Table 6), although convergence times are higher overall, IC_X still shows better performance than IC_P in mid-sized to larger instances. In the chain topology (Table 7), IC_X also stands out as it achieves the lowest convergence times (-5,97% w.r.t INI_{CNV}). Overall, IC_X added on top of the initial model emerges as the most effective in convergence detection, balancing low solving times with a higher number of solved instances.

Finally, we study the formulations augmented by the constraints eliminating the initial configurations. As illustrated in Tables 5 and 6, the comparison with the initial models highlights the relevance of introducing these constraints, particularly for the divergence analysis. The results show a significant reduction in solving time for ring and star topologies thanks to rotation elimination (RE) and lexicographic order (LO), particularly when they are combined with Iterative Convergence (IC) constraints. More specifically, for ring graphs, the ER and IC_X combination model achieves the best overall performance, with an average solving time of 324.10s for convergence. For divergence, RE achieves better performance with respect to the initial model with an average solving time of 596.52s, across all graph sizes tested. Compared to the initial model, ER + IC_X thus reduces the total solving time by 10,5% with a higher number of solved instances (+4 w.r.t INI_{CNV}) for convergence and ER reduces it by 55,14% for divergence while solving 15 additional instance with respect to INI_{DIV}. These results demonstrate the positive impact of the symmetry-breaking constraints on the analysis of synchronous unison behavior. The star graph reveals a better dynamic with LO which demonstrates a significant improvement in the divergence time with an average of 15,80s, a reduction of 81.5% compared to the initial model (59,29s) with a higher number of solved instances (+1 w.r.t INI_{DIV}). On the other hand, LO + IC_P presents a significant reduction in solving time for convergence with a total of 215,23s, a reduction of 90,71% compared to the initial model (1631,35s) and a higher number of solved instances (+2 w.r.t INI_{CNV}). These observations are also confirmed by Table 6 and highlight the relevance of this optimization in structures where interactions are highly centralized around a central node.

5 Conclusion

In this paper, we presented a formal approach based on propositional satisfiability to analyze the behavior of the self-stabilizing synchronous unison algorithm proposed by Arora *et al.* [4]. Specifically, we focused on detecting cases where this algorithm converges or diverges. By representing the problem states and its execution as logical constraints, we showed how SAT solvers can be used to prove the self-stabilization of a distributed system or, conversely, to produce a counterexample proving its divergence. This framework allowed us to rigorously study the self-stabilization of the algorithm in various topologies such as chains, rings and stars, while taking into account different periods. Furthermore, we optimized the analysis by proposing specific constraints to eliminate redundant initial configurations, which accelerated model evaluation and ensured more efficient results. Our models also exhibited specific behaviors, notably divergence in the case of stars when $m = 2$.

Several avenues of research remain open. First, extending our analysis to other types of graphs, such as grids, tori, or binary trees, would allow for a more comprehensive and detailed examination of the impact of the topology on the algorithm's self-stabilization. It could be also relevant to consider refining the choice of the maximum number of allowed configurations relative to the topology studied. Another interesting perspective involves considering symmetries for the detection of divergent cycles. These perspectives pave the way

for a better understanding of synchronization dynamics in complex distributed systems and for the development of new, more efficient and optimized methods to address a wider range of application cases. Our work could thus facilitate not only the analysis of synchronous unison, but more generally that of self-stabilizing algorithms in broader and more diverse contexts including the asynchronous case [16,17], variants of self-stabilization [11,12], reactive tasks [1,24] and the possibility of topological change [2,21].

References

1 Karine Altisen, Stéphane Devismes, and Anaïs Durand. Concurrency in snap-stabilizing local resource allocation. *Journal Of Parallel and Distributed Computing*, 102:42–56, 2017. doi:10.1016/J.JPDC.2016.11.004.

2 Karine Altisen, Stéphane Devismes, Anaïs Durand, Colette Johnen, and Franck Petit. Self-stabilizing systems in spite of high dynamics. *Theor. Comput. Sci.*, 964:113966, 2023. doi:10.1016/J.TCS.2023.113966.

3 Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. *Introduction to Distributed Self-Stabilizing Algorithms*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2019.

4 Anish Arora, Shlomi Dolev, and Mohamed G. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1:11–18, 1991. doi:10.1007/bfb0022438.

5 Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing – SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 – July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 167–180. Springer, 2009. doi:10.1007/978-3-642-02777-2_18.

6 Armin Biere. Handbook of satisfiability. In *Frontiers in Artificial Intelligence and Applications*, pages 75–98. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-75.

7 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.

8 Armin Biere et al. Symbolic model checking without bdds. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 193–207, 1999.

9 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.

10 Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of Satisfiability: Second Edition*. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021. URL: https://books.google.fr/books?id=dUAvEAAAQBAJ.

11 James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993. doi:10.1007/BF02278854.

12 Fabienne Carrier, Ajoy Kumar Datta, Stéphane Devismes, Lawrence L. Larmore, and Yvan Rivierre. Self-stabilizing (f, g)-alliances with safe convergence. *J. Parallel Distributed Comput.*, 81-82:11–23, 2015. doi:10.1016/J.JPDC.2015.02.001.

13 Jingshu Chen and Sandeep S. Kulkarni. Smt-based model checking for stabilizing programs' . In Davide Frey, Michel Raynal, Saswati Sarkar, Rudrapatna K. Shyamasundar, and Prasun Sinha, editors, *Distributed Computing and Networking, 14th International Conference, ICDCN 2013, Mumbai, India, January 3-6, 2013. Proceedings*, volume 7730 of *Lecture Notes in Computer Science*, pages 393–407. Springer, 2013. doi:10.1007/978-3-642-35668-1_27.

14 Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981. doi:10.1007/BFB0025774.

15 Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158, 1971. doi:10.1145/800157.805047.

16 J.-M. Couvreur, N. Francez, and M. G. Gouda. Asynchronous unison (extended abstract). In *12th International Conference on Distributed Computing Systems, (ICDCS'92)*, pages 486–493. IEEE Computer Society, 1992. doi:10.1109/ICDCS.1992.235005.

17 Stéphane Devismes, David Ilcinkas, Colette Johnen, and Frédéric Mazoit. Being efficient in time, space, and workload: a self-stabilizing unison and its consequences. In Olaf Beyersdorff, Michal Pilipczuk, Elaine Pimentel, and Kim Thang Nguyen, editors, *42nd International Symposium on Theoretical Aspects of Computer Science, STACS 2025, March 4-7, 2025, Jena, Germany*, volume 327 of *LIPIcs*, pages 30:1–30:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.STACS.2025.30.

18 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974. doi:10.1145/361179.361202.

19 Danny Dolev, Keijo Heljanko, Matti Järvisalo, Janne H. Korhonen, Christoph Lenzen, Joel Rybicki, Jukka Suomela, and Siert Wieringa. Synchronous counting and computational algorithm design. *J. Comput. Syst. Sci.*, 82(2):310–332, 2016. doi:10.1016/J.JCSS.2015.09.002.

20 Danny Dolev and Dahlia Malkhi. Consensus: Perspectives and challenges. In *Proceedings of the Tenth International Workshop on Distributed Algorithms (WDAG)*, pages 1–12, 1995. doi:10.1007/3-540-60220-8_1.

21 Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1995.

22 Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distrib. Comput.*, 7(1):3–16, November 1993. doi:10.1007/BF02278851.

23 Stéphane Grumbach and Zhilin Wu. Logical locality entails frugal distributed computation over graphs (extended abstract). In Christophe Paul and Michel Habib, editors, *WG*, volume 5911 of *Lecture Notes in Computer Science*, pages 154–165, 2009. doi:10.1007/978-3-642-11409-0_14.

24 Rachid Hadid. A stabilising optimal k-out-of- resources allocation algorithm. *Int. J. Knowl. Eng. Data Min.*, 5(3):173–194, 2018. doi:10.1504/IJKEDM.2018.10014813.

25 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, 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.

26 Asma Khoualdia, Sami Cherif, Stéphane Devismes, and Léo Robert. SAT_for_UNISON. Software, swhId: swh:1:dir:4bc78d189155024f85110cb2dc9ae4aeb470d8bf (visited on 2025-07-16). URL: https://github.com/asmakhoualdia98/SU_SAT_Exec, doi:10.4230/artifacts.23375.

27 Yassine Lakhnech and Michael Siegel. Deductive verification of stabilizing systems. In Sukumar Ghosh and Ted Herman, editors, *3rd Workshop on Self-stabilizing Systems, Santa Barbara, California, USA, August, 1997, Proceedings*, pages 201–216. Carleton University Press, 1997.

28 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. 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 123–140. Springer, 2016. doi:10.1007/978-3-319-40970-2_9.

29 Nathan Linial. Distributive graph algorithms-global solutions from local data. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 331–335. IEEE Computer Society, 1987. doi:10.1109/SFCS.1987.20.

30 João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999. doi:10.1109/12.769433.

31 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.

32 Olivier Roussel and Vasco Manquinho. Pseudo-boolean and cardinality constraints. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1087–1129. IOS Press, 2021. doi:10.3233/FAIA201012.

33 Michael Siegel. Formal verification of stabilizing systems. In Anders P. Ravn and Hans Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 5th International Symposium, FTRTFT'98, Lyngby, Denmark, September 14-18, 1998, Proceedings*, volume 1486 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1998. doi:10.1007/BFB0055345.

34 Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2000.

A Further Experimental Results

Table 7 Solving times in seconds for chains. The average solving times for detecting convergence (CONV) and divergence (DIV) over all periods for a given graph size (n) are reported. The results of other optimized models are also reported allowing additional constraints, and the best results among the models are marked in bold. For a given size n with instances that have reached the time limit, the number of solved instances is indicated in "[]".

n	CONVERGENCE			DIVERGENCE
	INI_{CNV}	IC_{P}	IC_{X}	INI_{DIV}
3	0.04	0.043	**0.030**	0.13
4	0.50	0.49	**0.43**	1.58
5	2.14	**2.00**	2.04	5.26
6	6.26	6.32	**5.88**	15.45
7	15.73	14.84	**14.27**	29.99
8	36.34	39.42	**34.33**	77.61
9	68.83	65.79	**62.28**	136.50
10	**110.36**	110.56	117.65	203.64
11	140.07	125.27	**123.88**	332.38
12	22.22	31.38	**20.07**	374.40
13	**25.44**	48.88	26.86	579.32
14	48.96	**28.80**	29.68	693.87
15	67.21	89.78	**66.52**	733.70 [18]
16	**73.89**	91.88	79.18	961.18 [18]
17	189.63	109.90	**89.17**	895.57 [17]
18	199.90	96.98	**86.50**	646.86 [16]
19	214.00	142.18	**105.24**	1292.26 [16]
20	185.48	209.33	**138.50**	1180.25 [15]

Table 8 Comparison of Cadical and MapleSAT solving times in seconds on chain topologies for the initial models of convergence (INI_{CNV}) and divergence (INI_{DIV}). The average solving times for detecting convergence (CONV) and divergence (DIV) over all periods for a given graph size (n) are reported, with the best results marked in bold. For a given size n with instances that have reached the time limit, the number of solved instances is indicated in "[]".

n	CONV		DIV	
	Cadical	MapleSAT	Cadical	MapleSAT
3	**0.042**	0.045	**0.13**	0.15
4	**0.50**	0.58	1.58	**1.44**
5	**2.14**	2.35	5.26	**4.65**
6	**6.26**	7.11	15.45	**13.96**
7	**15.73**	20.93	**29.99**	56.89
8	**36.34**	49.46	**77.61**	178.70
9	**68.83**	109.96	**136.50**	274.05[18]
10	**110.36**	234.97	**203.64**	993.92
11	**140.07**	377.84	**332.38**	584.39 [17]
12	**22.22**	405.42	**374.40**	668.67 [17]
13	**25.44**	369.56	**579.32**	1265.57 [17]
14	**48.96**	958.41	**693.87**	1145.67 [15]
15	**67.21**	319.76 [16]	**733.70 [18]**	682.19 [13]
16	**73.89**	688.95	**961.18 [18]**	1185.94 [13]
17	**189.63**	432.39 [16]	**895.57 [17]**	1193.92 [12]
18	**199.90**	826.68 [16]	**646.86 [16]**	1036.81 [11]
19	**214.00**	379.40 [14]	**1292.26 [16]**	737.62 [10]
20	**185.48**	459.89 [15]	**1180.25 [15]**	1001.62 [10]

Table 9 Comparison of Cadical and MapleSAT solving times in seconds on ring topologies for the initial models of convergence (INI_{CNV}) and divergence (INI_{DIV}). The average solving times for detecting convergence (CONV) and divergence (DIV) over all periods for a given graph size (n) are reported, with the best results marked in bold. For a given size n with instances that have reached the time limit, the number of solved instances is indicated in "[]".

n	CONV		DIV	
	Cadical	MapleSat	Cadical	MapleSat
3	0.025	**0.018**	**0.05**	0.06
4	**0.69**	0.71	**1.48**	1.78
5	1.97	**1.74**	**2.33**	3.18
6	**7.34**	8.31	**9.46**	16.58
7	**11.58**	22.32	**15.00**	44.11
8	**36.25**	75.15	**48.89**	220.96
9	**55.56**	222.82	**68.76**	554.39
10	**109.10**	527.96	**141.30**	810.69 [18]
11	**187.26**	592.69 [18]	**231.78**	703.10 [16]
12	**331.62**	601.80 [17]	**427.35**	605.34 [15]
13	**420.22**	1240.20 [17]	**503.95**	901.28 [15]
14	**658.15**	691.24 [15]	**844.47**	636.14 [13]
15	**858.95**	647.02 [14]	**1172.70**	1005.36 [13]
16	**1500.84**	183.36 [13]	**852.05 [16]**	988.76 [13]
17	**819.73 [16]**	177.51 [13]	**1196.05 [16]**	1096.93 [12]
18	**73.47 [15]**	661.29 [15]	**560.75 [15]**	1491.43 [13]
19	**44.35 [15]**	731.77 [15]	**1164.60 [14]**	952.23 [12]
20	**166.89 [17]**	833.91 [15]	**502.73 [13]**	711.09 [10]

Table 10 Comparison of Cadical and MapleSAT solving times in seconds on star topologies for the initial models of convergence (INI_{CNV}) and divergence (INI_{DIV}). The average solving times for detecting convergence (CONV) and divergence (DIV) over all periods for a given graph size (n) are reported, with the best results marked in bold. For a given size n with instances that have reached the time limit, the number of solved instances is indicated in "[]".

n	**CONV**		**DIV**	
	Cadical	MapleSat	Cadical	MapleSat
3	0.007	**0.003**	**0.01**	0.0129
4	**0.04**	0.05	**0.09**	0.10
5	1.00	**0.59**	0.90	**0.40**
6	27.38	**17.82**	6.97	**5.50**
7	862.27	**701.20**	76.61	**75.41**
8	553.70 [6]	**321.57 [6]**	146.91 [6]	**67.39 [6]**
9	119.25 [4]	**1302.39 [5]**	**166.68 [5]**	205.43 [5]
10	**67.71 [3]**	94.48 [3]	76.16 [3]	**1537.67 [4]**

Dependency-Curated Large Neighbourhood Search

Frej Knutar Lewander[1]
Department of Information Technology, Uppsala University, Sweden

Pierre Flener
Department of Information Technology, Uppsala University, Sweden

Justin Pearson
Department of Information Technology, Uppsala University, Sweden

Abstract

In large neighbourhood search (LNS), an incumbent initial solution is incrementally improved by selecting a subset of the variables, called the *freeze set*, and fixing them to their values in the incumbent solution, while a value for each remaining variable is found and assigned via solving (such as constraint programming-style propagation and search). Much research has been performed on finding generic and problem-specific LNS selection heuristics that select freeze sets that lead to high-quality solutions. In constraint-based local search (CBLS), the relations between the variables via the constraints are fundamental and well-studied, as they capture dependencies of the variables. In this paper, we apply these ideas from CBLS to the LNS context, presenting the novel dependency curation scheme, which exploits them to find a low-cardinality set of variables that the freeze set of any selection heuristic should be a subset of. The scheme often improves the overall performance of generic selection heuristics. Even when the scheme is used with a naïve generic selection heuristic that selects random freeze sets, the performance is competitive with more elaborate generic selection heuristics.

2012 ACM Subject Classification Computing methodologies → Heuristic function construction

Keywords and phrases Combinatorial Optimisation, Large Neighbourhood Search (LNS), Constraint-Based Local Search (CBLS)

Digital Object Identifier 10.4230/LIPIcs.CP.2025.20

Supplementary Material *Software (Solver)*: `https://github.com/astra-uu-se/gecode-lns` archived at `swh:1:dir:6415a1abc5e2361c3994f6e8a17cfc4716edc89f`
Software (Results, Experiments, and Scheme Generator): `https://github.com/astra-uu-se/gecode-lns-experiments`, archived at `swh:1:dir:099aba93950a85b9ccbb07a750c31877cb21f019`

Funding Supported by grant 2018-04813 of the Swedish Research Council (VR).

Acknowledgements We thank Mikael Zayenz Lagerkvist and Dexter Leander for their help with the Gecode implementation.

1 Introduction

Large neighbourhood search (LNS) [23, 19] is a method that combines systematic search with local search to improve the scalability of the former on constrained optimisation problems using heuristics of the latter. LNS starts from an incumbent solution that is iteratively improved by fixing a subset of the variables to their values in the incumbent solution and a value for each remaining variable is found and assigned via solving (such as constraint programming-style propagation and search). This method has been very successful on a wide variety of problems, such as vehicle routing [23, 2], bin packing [28], and scheduling [8, 24].

[1] Corresponding author

© Frej Knutar Lewander, Pierre Flener, and Justin Pearson;
licensed under Creative Commons License CC-BY 4.0

Traditionally, a modeller has to construct a problem-specific selection heuristic for the determination of the subset of variables to fix [8, 22]. However, there are now many variants of LNS that automatically determine that subset with good search performance, such as (but not limited to) (reverse) propagation guided LNS [18], cost impact guided LNS [12], explanation-based LNS [20], self-adaptive LNS [26], and variable-relationship guided LNS [24].

In most combinatorial optimisation solvers, such as for mixed integer linear programming, constraint programming (CP), Boolean satisfiability (SAT), and constraint-based local search (CBLS), when some variable x becomes fixed, the values of some other variables can be found and assigned via search-free unique solving (via inference such as CP propagation, SAT unit propagation, and CBLS invariant propagation). Therefore, the relations between x and the assigned variables are functional dependencies. In CBLS, these functional dependencies are fundamental and heavily studied [15, 16, 27, 10]. We have not found generic LNS selection heuristics or CP branching heuristics that automatically exploit these functional dependencies that are exploited in CBLS. However, they are often exploited manually by the modeller when for example creating a problem-specific CP branching heuristic that guides search.

Our contributions are:

- applying from CBLS to a CP context (and hence to CP-based LNS) the idea of a directed possibly cyclic graph that is induced by the functional dependencies between variables via the constraints of the model;
- designing a scheme that exploits the induced directed graph to remove variables from the LNS space that are functionally defined by others;
- describing how our scheme can be automated;
- showing that state-of-the-art generic LNS selection heuristics are easily extended to make use of our scheme;
- showing that our scheme often improves the overall performance when used with state-of-the-art generic selection heuristics; and
- showing that our scheme, when used with a naïve generic LNS selection heuristic that fixes a set of variables selected at random inside each LNS iteration, is competitive with more elaborate state-of-the-art generic LNS selection heuristics, even when they also use our scheme.

We first give related work from CBLS in Section 2, before we present how to apply it to a CP context (and hence to CP-based LNS) in Section 3. We then give an example of functional dependencies for a CP model in Section 4. We describe generic LNS selection heuristics from the literature in Section 5. We present our scheme in Section 6. We give our experiment setup in Section 7 and discuss the results in Section 8. Finally, we conclude in Section 9.

2 Constraint-Based Local Search

In constraint-based local search [15, 16, 27, 10] (CBLS), each constraint of the problem is either made to always implicitly hold via the initialisation and neighbourhood, or replaced by one or more invariants, where an *invariant* (also called a *one-way constraint*) defines a subset of the constrained variables, called its *output variables*, of the replaced constraint by the remaining variables, called its *input variables*. Invariants that replace a constraint of the problem are called violation invariants, where a *violation invariant* introduces and defines a *violation variable*, absent in the problem, which takes the value 0 if the replaced constraint holds, else a positive value, usually proportional to the amount of violation. The replaced constraint is therefore softened, but only an assignment of the variables where all violation variables take the value 0 is a solution, as it satisfies all the constraints of the problem.

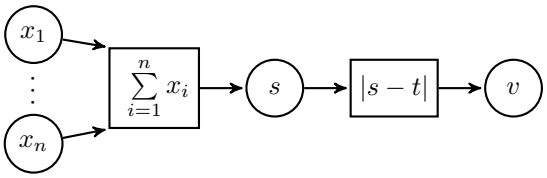


Figure 1 An invariant graph for the subset-sum constraint satisfaction problem of size n with $n+2$ integer variables and two invariants. Note that t is not a variable.

The invariants and variables induce a directed graph, called the *invariant graph*, with the invariants and variables as vertices and the definitions of variables via invariants as arcs, where each variable has an outgoing arc to each invariant that it is an input variable to and an incoming arc from the invariant that defines it (if any). Note that in any CBLS model, any variable is defined by at most one invariant. All the possibly multiple (if not zero) source vertices and possibly multiple (if not zero) sink vertices are variables, where the source variables transitively define the remaining variables via the invariants.

As an example of an invariant graph, consider the subset-sum constraint satisfaction problem, where given a set of n integers v_i and an integer t, a subset of the set is to be found that sums to exactly t. A corresponding invariant graph contains n integer variables, where each variable x_i takes the value v_i if v_i is in the subset and 0 otherwise; one invariant with as inputs the variables x_i and as output the integer variable s, which takes as value the sum of the integers in the subset; and one violation invariant with as input s and as output the violation variable v, which is to be minimised and takes as value the absolute difference between s and t. This amounts to softening the constraint $\sum_{i=1}^{n} x_i = t$. The source variables are the variables x_i, and they transitively define the other variables via the two invariants. Any assignment of the variables x_i where s takes the value t is a solution, as $v = 0$ then, which is minimal. The invariant graph is shown in Figure 1, where circled vertices are variables and boxed vertices are invariants.

During a CBLS-style search, only the values of the source variables are changed, while the values of the remaining variables are found via CBLS-style invariant propagation.

In CBLS, the variables and constraints induce the invariant graph of definitions of variables, whether this is done by the CBLS modeller or by automated analysis of, say, a MiniZinc [17] model [4]. In the sequel, we apply the idea of the induced invariant graph to CP models to find functional definitions of variables, which can be exploited during LNS.

3 The Dependency Graph of a CP Model

In a CP model, some constraints are dependency constraints, where a *dependency constraint* c on the variables $\mathcal{X} = \mathcal{I} \cup \mathcal{O}$ determines the values of the variables in $\mathcal{O}$ when the variables in $\mathcal{I}$ become fixed. We can view c as a function that determines the values of output variables $\mathcal{O}$ when the input variables $\mathcal{I}$ become fixed. We say that c is a *dependency constraint* and that $\mathcal{I}$ *functionally defines* $\mathcal{O}$ via c, denoted by $\mathcal{I} \overset{c}{\Longrightarrow} \mathcal{O}$, or simply that $\mathcal{I}$ *functionally defines* $\mathcal{O}$.

For example, consider the integer variables y and i, the array $\mathcal{P}$ of integers, and the constraint Element$(\mathcal{P}, i, y)$, which constrains y to be equal to the integer in $\mathcal{P}$ at index i. When i becomes fixed to some value v, the value of y becomes assigned to the integer in $\mathcal{P}$ at index v. The constraint is a dependency constraint, via which i functionally defines y.

As another example, consider the Boolean variable b, the array of Boolean variables $\mathcal{B}$, and the constraint Exists$(\mathcal{B}, b)$, which constrains b to take the value **True** if any of the variables in $\mathcal{B}$ takes the value **True**, else b takes the value **False**. The constraint is a dependency constraint, via which $\mathcal{B}$ functionally defines b.

Some constraints are not dependency constraints. For example, consider the integer variables x and y, and the constraint LessEqual(x, y), which enforces that x is smaller than or equal to y. This constraint is not a dependency constraint as neither x nor y can be defined via it by the other variable.

For a CP model, the dependency constraints and the variables induce a directed graph, called the *dependency graph* [13], where for each set $\mathcal{I}$ of variables that functionally defines a set $\mathcal{O}$ of variables via some dependency constraint c, there is a vertex d, an arc $x \to d$ for each vertex $x \in \mathcal{I}$, and an arc $d \to y$ for each vertex $y \in \mathcal{O}$. Note that the dependency graph is possibly cyclic: we will see an example of an acyclic dependency graph in Figure 2 in Section 4, and an example of a cyclic dependency graph in Figure 3 in Section 7.2.3. Also note that all the variables and all the dependency constraints are in the dependency graph, while the other (non-dependency) constraints are not.

A dependency graph differs from an invariant graph as follows:

- in a dependency graph, a variable can have any number of incoming arcs, while it can have at most one in an invariant graph;
- a dependency graph does not contain non-dependency constraints, while in an invariant graph, all constraints that are not made to implicitly hold are encoded (as invariants); and
- an invariant graph is required when performing CBLS-style invariant propagation, while a dependency graph is not required when performing CP-style solving (neither in search nor in CP-style propagation).

For any acyclic dependency graph, the source variables transitively functionally define all remaining variables. Therefore, for any acyclic dependency graph, the minimum-cardinality set of such variables is the set of source variables. However, this does not always hold for a cyclic dependency graph (as we will see for the dependency graph shown in Figure 3 in Section 7.2.3). In Section 6, we present a greedy scheme that, given a CP model (inducing a cyclic or acyclic dependency graph), finds a low-cardinality set of such variables, which can be exploited to guide LNS.

4 Example: Relaxed Car Sequencing (RCS)

We now give an example to show how the dependency graph is constructed from a CP model. The car sequencing (constraint satisfaction) problem [5] has a set of car classes, each class with a set of mandatory features (called options in [5]), each feature with a capacity on how many cars of that feature can be produced in a subsequence of given size, and an order stating how many cars of each class to produce. The problem is to find a production sequence for all ordered cars, such that the capacity restrictions on features are satisfied and all ordered cars are produced. Since LNS can only be performed on constrained optimisation problems, we transform the problem into one by relaxing it into the relaxed car sequencing (RCS) problem by (i) including an additional dummy class of car that has no features; (ii) relaxing the constraint that all ordered cars are to be produced by replacing it by a constraint that each car is to be produced at most once; and (iii) introducing an objective variable that is to be minimised and is the number of dummy cars that are produced. Note that for any solution to the relaxed problem, if the objective variable takes the value 0, then the solution satisfies the constraints of the original problem, as all cars are produced.

A model for RCS in the solver-independent MiniZinc language [17] is in Listing 1. Lines 1–7 declare the set of cars to be produced, the set of features, the set of car classes, the array of the sizes of the feature subsequences, the array of the maximum capacities for

Listing 1 A MiniZinc model for the relaxed car sequencing (RCS) problem.

```
1  set of int: Cars;
2  set of int: Features;
3  set of int: Classes;
4  array[Features] of int: sequenceSize;
5  array[Features] of int: sequenceCapacity;
6  array[Classes] of int: numOrdered;
7  array[Classes, Features] of bool: classHasFeature;
8  int: dummyClass = max(Classes) + 1;
9  set of int: ClassesAndDummy = Classes union {dummyClass};
10 array[ClassesAndDummy, Features] of bool: classHasFeatureExt =
11   array2d(ClassesAndDummy, Features,
12     array1d(classHasFeature) ++ [false | _ in Features]);
13 array[Cars] of var ClassesAndDummy: class;
14 array[Cars, Features] of var bool: carHasFeature = array2d(
15   Cars, Features, [classHasFeatureExt[class[car], f]
16                    | car in Cars, f in Features]);
17 array[ClassesAndDummy] of var int: numProduced =
18   global_cardinality_closed(class, ClassesAndDummy);
19 constraint forall(c in Classes)(numProduced[c]<=numOrdered[c]);
20 constraint forall(f in Features)(
21   sliding_sum(0, sequenceCapacity[f],
22               sequenceSize[f], carHasFeature[.., f]));
23 solve minimize numProduced[dummyClass];
```

the feature subsequences, the array of the numbers of ordered cars for each class, and the two-dimensional matrix of the features that the car classes use respectively. The sets of cars to be produced, features, and car classes are required to be intervals. Lines 8–12 declare the dummy car class, the set of car classes including the dummy car class, and the two-dimensional matrix of the features that the car classes and dummy class use, respectively. On line 13, the array `class` of variables is declared, where `class[c]` denotes the class of car `c`. On lines 14–16, the two-dimensional matrix `carHasFeature` of variables is declared and is functionally defined by `class`, where `carHasFeature[c,f]` denotes if feature `f` is to be installed on car `c`. On lines 17–18, the array `numProduced` of variables is declared and is functionally defined by `class`, where `numProduced[c]` denotes the number of cars of class `c` that are actually produced. Line 19 enforces that no car class is overproduced. Lines 20–22 enforce that the feature capacity is not violated for any subsequence. Finally, Line 23 declares that the number of dummy cars is to be minimised.

This model was retrieved from the MiniZinc Benchmark repository and relaxed from a constraint satisfaction model into a constrained optimisation model as well as rewritten by renaming parameters and variables and replacing each constraint predicate with the corresponding constraint function where possible.[2]

The acyclic dependency graph for RCS with three cars to produce, two features, and two car classes induced by the MiniZinc model in Listing 1 is shown in Figure 2, where circled vertices are variables and boxed vertices are dependency constraints.

[2] The MiniZinc Benchmark repository: `https://github.com/MiniZinc/minizinc-benchmarks`

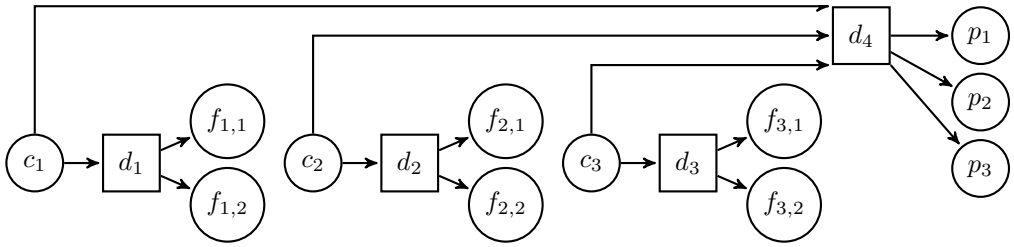


Figure 2 The acyclic dependency graph induced by the RCS MiniZinc model of Listing 1 with 3 cars, 2 features, and 2 non-dummy car classes. Integer variable c_i denotes `class[`i`]`, Boolean variable $f_{i,j}$ denotes `carHasFeature[`i`,`j`]`, and integer variable p_ℓ denotes `numProduced[`ℓ`]`. Vertices d_1, d_2, and d_3 correspond to the dependency constraints on lines 14–16 and vertex d_4 corresponds to lines 17–18. The constraints in lines 19–22 are non-dependency constraints and are therefore absent in the dependency graph.

5 Large Neighbourhood Search

LNS [23, 19] is an iterative method for optimisation problems, starting from an incumbent feasible solution that is improved during search. In every iteration, first a subset of the variables, called the *freeze set*, is selected and the values of these variables are kept from the incumbent solution, then search (for example CP-style branch and bound search) is performed until some stopping criterion is met, and finally, if a better solution was found, then the incumbent solution is replaced by it and a constraint that all future solutions must be better than the new incumbent solution is added. The selection of a freeze set is done via a *selection heuristic*, which can be either problem-specific or generic. Typically, problem-specific selection heuristics (such as [8, 22]) cannot easily be reused between problems.

In Section 5.1 we give a naïve generic selection heuristic that selects the freeze set at random, and in Sections 5.2 to 5.4 we give more elaborate generic selection heuristics from the literature.

5.1 Randomised LNS

Randomised LNS is a naïve generic selection heuristic that inside each LNS iteration creates a freeze set by, for each variable x, selecting uniformly from the continuous closed interval $[0, 1]$ a value p: if p is below some threshold ϕ (in our implementation $\phi \in [0.05, 0.95]$), then x is included in the freeze set. The threshold ϕ is typically updated during search (with initial value 0.6 in our implementation), where it is decreased to escape local minima of the LNS space, while it is increased to focus on exploring a smaller part of the LNS space.

5.2 Propagation Guided LNS and Reverse Propagation Guided LNS

Propagation guided LNS [18] (PG-LNS) has a selection heuristic that gradually expands the freeze set inside each LNS iteration, starting from the empty set, by freezing variables with the largest reduction in domain size. The freeze set is expanded until the size of the CP search space becomes smaller than some predefined size (computed from the sum of the logarithms of the sizes of the current domains of the variables).

Reverse propagation guided LNS [18] (RPG-LNS) differs from PGLNS by having a selection heuristic that gradually excludes variables from the freeze set inside each LNS iteration, starting from the set of all variables, by excluding variables with the smallest reduction in domain size. The freeze set is shrunk until the size of the CP search space becomes greater than some predefined size (which is dynamically updated during search).

5.3 Cost Impact Guided LNS

Cost impact guided LNS [12] (CIG-LNS) has an impact collecting procedure and a selection heuristic, where the latter makes use of information gathered by the former.

During impact collection, several dives are performed, where in each *dive*, first a random permutation of the variables is constructed, then each variable in the permutation is fixed to its value in the incumbent solution, and then the change in the lower (upper) bound of the objective variable (often called *cost variable*) for a minimisation (maximisation) problem is retrieved. For each variable and over the dives, the sum of the retrieved changes in the bound of the objective variable is stored and used by the selection heuristic. For each variable, this sum is called the *impact* of that variable.

The selection heuristic gradually shrinks the freeze set inside every LNS iteration, starting from the set of all variables, by excluding the variables with the greatest impacts as gathered by the impact collecting procedure.

Typically, the impact collecting procedure is performed every 10 LNS iterations and 10 dives are performed during it.

5.4 Variable-Relationship Guided LNS

Given a variable x that is not fixed, the *local cost* of x is the change in the lower (upper) bound of the objective variable for a minimisation (maximisation) problem when x is fixed to its value in the incumbent solution.

Variable-relationship guided LNS [24] (VRG-LNS) has a selection heuristic such that inside each LNS iteration the freeze set is gradually shrunk, starting from the set of all variables, by either (i) first selecting a random subset of the variables and then removing one amongst them that has the greatest local cost, or (ii) removing a variable that is constrained by one or more of the same constraints as the previously excluded variable. The selection heuristic shrinks the freeze set by alternating between (i) and (ii), starting with (i). The number of excluded variables is low in initial iterations but changes during search, where it is increased in some iterations to escape local minima of the LNS space, before it is restored to its original size.

The VRG-LNS selection heuristic (ii) selects a variable x that shares constraints with the previously selected variable y, but does not exploit what we call the dependency graph as the shared constraints do not have to be dependency constraints and there is no requirement that x or y functionally define each other.

6 Dependency Curation for LNS

In the selection heuristics of PG-LNS, RPG-LNS, and VRG-LNS, the freeze set is gradually updated inside each LNS iteration to include variables such that many other variables are assigned values via CP-style propagation [18, 24]. For these selection heuristics, such variables are found either experimentally or heuristically *during* search. We now show such variables can also be found by exploiting the dependency graph *before* search starts and how they can be used by any LNS selection heuristic.

We call any set of variables that transitively functionally define all remaining variables a set of *search variables*, as only their values must be found via search.

Consider a CP model with the set $\mathcal{V}$ of variables. Finding a set $\mathcal{S} \subseteq \mathcal{V}$ of search variables is trivial as $\mathcal{V}$ itself is a set of search variables, though of maximum cardinality. Our idea is that the smaller the set $\mathcal{S}$ is, the more CP-style propagation will occur, as the value of each

variable in $\mathcal{V} \setminus \mathcal{S}$ is found and assigned via only propagation (as $\mathcal{S}$ transitively functionally defines $\mathcal{V} \setminus \mathcal{S}$). Additionally, for any (generic or problem-specific) selection heuristic, if the freeze set is forced to become a subset of $\mathcal{S}$, then the LNS space is reduced, no data has to be stored, and no operations have to be performed on any variable in $\mathcal{V} \setminus \mathcal{S}$ by the selection heuristic, potentially improving its memory footprint and running time.

Note that if the dependency graph is acyclic, then the minimum-cardinality set of search variables is the set of source variables. Otherwise, the dependency graph contains at least one strongly connected component (SCC) with two or more vertices, and finding a low-cardinality set of search variables is a subtle issue, as discussed next.

As the variables and dependency constraints are known up-front, a low-cardinality set of search variables can be constructed before search starts. A (generic or problem-specific) selection heuristic can use the constructed set of search variables throughout search by forcing the freeze set to be a subset of that set.

Our scheme, called the *dependency curation scheme* (DCS), finds a low-cardinality set of search variables given the variables, vertices, and arcs of a dependency graph. It is shown in greedy Algorithm 1, where:

- function stronglyConnectedComponents($\mathcal{N}, \mathcal{A}$) returns an ordered list of SCCs for vertices $\mathcal{N}$ and arcs $\mathcal{A}$, such that each SCC is before all other SCCs that are reachable from it, like [25] but where the returned list is reversed (as the graph is explored in depth-first order);
- function in($\mathcal{A}, v$) returns all incoming arcs to vertex v in the set $\mathcal{A}$ of arcs;
- function out($\mathcal{A}, v$) returns all outgoing arcs from vertex v in the set $\mathcal{A}$ of arcs;
- function stack() returns an empty stack;
- function empty($\mathcal{Z}$) returns **True** if the stack $\mathcal{Z}$ contains no elements, else **False**;
- function pop($\mathcal{Z}$) removes the top-most element from the stack $\mathcal{Z}$ and returns it; and
- procedure push($\mathcal{Z}, v$) pushes element v onto stack $\mathcal{Z}$.

First, the sets $\mathcal{W}$ of visited vertices and $\mathcal{S}$ of search variables are initialised, lines 2–3. Each SCC of the dependency graph is iterated over, where SCC $\mathcal{X}$ is iterated over before all other SCCs that are reachable from $\mathcal{X}$ are iterated over, line 4. While there are unvisited variables in $\mathcal{X}$, line 5, a variable x with the lexicographically lowest priority is retrieved from the unvisited variables in $\mathcal{X}$, where the *priority* of a variable y is (i) the in-degree of y and (ii) the opposite of the out-degree of y, line 6. Variable x is added to the set $\mathcal{S}$ of search variables, line 7, and to the set $\mathcal{W}$ of visited vertices, line 8. The empty stack $\mathcal{Z}$ is created, line 9, before x is added as the top-most element to $\mathcal{Z}$, line 10. While $\mathcal{Z}$ is not empty, line 11, its top-most element u is removed, line 12. The outgoing arcs $\mathcal{A}_u$ from u are retrieved, line 13, and removed from $\mathcal{A}$, line 14. For each arc $(u, v) \in \mathcal{A}_u$, where v has not been visited and either (i) is a variable or (ii) has no incoming arcs in $\mathcal{A}$, line 15, vertex v is added to the set $\mathcal{W}$ of visited vertices, line 16; its incoming arcs are removed from $\mathcal{A}$, line 17; and v is added as the top-most element to $\mathcal{Z}$, line 18. Finally, the found set $\mathcal{S}$ of search variables is returned, line 19.

Note that Algorithm 1 is greedy as the selection of a search variable x, line 6, potentially makes the set $\mathcal{S}$ *not* a minimal-cardinality set of search variables.

For example, consider the car sequencing problem given in Section 4 with three cars, two features, and two car classes; its MiniZinc model in Listing 1; and its dependency graph in Figure 2. The dependency graph contains 16 SCCs: one for each vertex. Algorithm 1 first creates the initially empty sets of search variables and visited vertices, lines 2–3. The sets $\{c_1\}$, $\{c_2\}$, and $\{c_3\}$ of vertices are the first three SCCs that are iterated over, as c_1, c_2, and c_3 are the only vertices without incoming arcs, line 4. The remaining lines then explore

Algorithm 1 The dependency curation scheme for finding a low-cardinality set of search variables.

Data: $\mathcal{V}$ is the set of variables, $\mathcal{N} \supseteq \mathcal{V}$ is the set of nodes in the dependency graph, and $\mathcal{A}$ is the set of arcs in the dependency graph.
Result: A set $\mathcal{S} \subseteq \mathcal{V}$ of search variables, which transitively functionally define the remaining variables $\mathcal{V} \setminus \mathcal{S}$ via the constraints.

function dependency-curation($\mathcal{V}, \mathcal{N}, \mathcal{A}$)**:**
  $\mathcal{W} \coloneqq \emptyset$ // The set of visited vertices
  $\mathcal{S} \coloneqq \emptyset$ // The set of search variables
  forall $\mathcal{X} \in$ stronglyConnectedComponents($\mathcal{N}, \mathcal{A}$) **do**
    while $(\mathcal{X} \cap \mathcal{V}) \setminus \mathcal{W} \neq \emptyset$ **do**
      $x \coloneqq \arg\operatorname{lex}\min_{y \in (\mathcal{X} \cap \mathcal{V}) \setminus \mathcal{W}} (\langle |\text{in}(\mathcal{A}, y)|, -|\text{out}(\mathcal{A}, y)| \rangle)$
      $\mathcal{S} \coloneqq \mathcal{S} \cup \{x\}$
      $\mathcal{W} \coloneqq \mathcal{W} \cup \{x\}$
      $\mathcal{Z} \coloneqq$ stack() // The stack used for depth-first search
      push($\mathcal{Z}, x$)
      while ¬empty($\mathcal{Z}$) **do**
        $u \coloneqq$ pop($\mathcal{Z}$)
        $\mathcal{A}_u \coloneqq \text{out}(\mathcal{A}, u)$
        $\mathcal{A} \coloneqq \mathcal{A} \setminus \mathcal{A}_u$
        forall $u \to v \in \mathcal{A}_u$ **where** $v \notin \mathcal{W}$ **and** $(v \in \mathcal{V}$ **or** $\text{in}(\mathcal{A}, v) = \emptyset)$ **do**
          $\mathcal{W} \coloneqq \mathcal{W} \cup \{v\}$
          $\mathcal{A} \coloneqq \mathcal{A} \setminus \text{in}(\mathcal{A}, v)$
          push($\mathcal{Z}, v$)
  return $\mathcal{S}$

the dependency graph in a fashion similar to depth-first search, starting from these source variables, where traversed outgoing arcs are removed and outgoing arcs from any non-variable vertex are only traversed if all its incoming arcs have been traversed and removed. The dependency graph is acyclic, and therefore the variables c_1, c_2, and c_3 transitively functionally define the remaining variables and actually form (in this case) a minimum-cardinality set of search variables.

If each strongly connected component only has a single vertex, then the dependency graph is acyclic and the algorithm can be bypassed, as the minimum-cardinality set of search variables is the set of source variables.

For a CP model, DCS finds a low-cardinality set of search variables that can be used with any (generic or problem-specific) selection heuristic by forcing its freeze set to be a subset of that set of search variables. Note that DCS itself is not a selection heuristic and must be used together with one, such as any of the generic selection heuristics in Sections 5.1 to 5.4.

VRG-LNS (Section 5.4) [24] is similar to DCS as both find the information of relations on variables via constraints before LNS iterations. However, they differ from each other, as VRG-LNS uses *both* dependency and non-dependency constraints that each pair of variables share, while DCS uses the functional definitions of variables via *only* dependency constraints.

7 Experiments

In Section 7.1, we give the details of our extensions to the solver that was used for the experiments. In Section 7.2, we describe how we selected the problems and for each problem, its specification and how its initial incumbent solutions are generated.

7.1 Setup

We extended the Gecode-based information-sharing portfolio solver in [11] – containing amongst other assets the CP solver Gecode [6]; Gecode-based LNS for randomised LNS, PG-LNS, and RPG-LNS; and Gecode-based LNS inspired by CIG-LNS and VRG-LNS – by:

- reimplementing CIG-LNS and VRG-LNS by following [12] and [24] more closely;
- allowing for supplying a set of search variables;
- enforcing the freeze set to be a subset of the set of search variables, if one was supplied;
- allowing for running a single generic LNS selection heuristic without any parallelism instead of simultaneously running multiple communicating assets in parallel (effectively making it a non-portfolio solver); and
- allowing for supplying an initial solution to be used as the first incumbent solution.

Note that we could have chosen any open-source CP solver, such as MiniCP [14], but we chose to extend the solver in [11], as it is very competitive; already has support for randomised LNS, PG-LNS, and RPG-LNS; as well as already has support for a variation each of CIG-LNS and VRG-LNS.

We decided not to implement the generic selection heuristics of explanation-based LNS [20] and self-adaptive LNS [26], as there is no Gecode-based LNS or one inspired thereof for either of them in the portfolio solver [11].

Additionally, since variable objective LNS [21] does not have a generic selection heuristic and therefore must be paired with (a generic or problem-specific) one, we decided not to use the generic selection heuristic that makes use of it [11].

There are problem-specific LNS selection heuristics, for example [8] for the job shop problem and [22] for steel mill slab design. However, we here only compare the performance of generic LNS selection heuristics, both with and without the use of DCS, as DCS can be used with any (generic or problem-specific) LNS selection heuristic. Additionally, we believe that the use of DCS for most problem-specific LNS selection heuristics gives little to no improvement in performance, as for any such LNS selection heuristic, the functional definitions via constraints are typically exploited by it by design.

For each run on a problem instance, we supply the same initial incumbent solution in order to make a fair comparison between selection heuristics on the same instance, as we can then compare the quality of their found solutions.

7.2 Problem Selection

We selected four problems: relaxed car sequencing (Section 4) [5] (RCS), steel mill slab design [9, 22] (SMSD), job shop (JSP), and the travelling salesperson problem with time windows [7] (TSPTW). RCS and SMSD were selected since their models contain many dependency constraints, and therefore our conjecture was that DCS improves performance. Conversely, JSP was selected since its model contains few dependency constraints, and therefore our conjecture was that DCS does not improve performance. Lastly, TSPTW was selected since the induced dependency graph is cyclic (as will be seen in Section 7.2.3), because of cyclic dependency constraints. SMSD was used both in [12] and [24], while a version of RCS was also used in the latter.

We used models in the solver-independent MiniZinc language [17] and instances for RCS and SMSD from the MiniZinc Benchmark repository, where we rewrote each model and instance for ease of readability, by renaming parameters and variables, moving some parameters from the models to the instances, and replacing each constraint predicate with the corresponding constraint function where possible. As the JSP model in the MiniZinc benchmark repository does not make use of suitable global constraints (such as `disjunctive`), we handcrafted a new MiniZinc model for JSP but used MiniZinc instances from the MiniZinc benchmark repository. We handcrafted a MiniZinc model for TSPTW, starting from [3]. We translated published instances for TSPTW into MiniZinc-readable instances.[3]

We created for each problem a DCS generator that, for each problem instance, finds a low-cardinality set of search variables.

We gave the specification for RCS in Section 4. The dependency graph for each RCS instance is acyclic, and the minimum-cardinality set of search variables is the set of classes of the cars, as shown in Section 6. The initial incumbent solution for each RCS instance is where the class of each car is the dummy class, as that fulfils the capacity restrictions on the features.

We give the specifications, the found sets of search variables, and how the initial incumbent solutions are created for the remaining problems in Sections 7.2.1 to 7.2.3.

7.2.1 Steel Mill Slab Design (SMSD)

The steel mill slab design problem is a constrained optimisation problem that has a set of orders, each with a size and colour; a set of slabs, each taking one of a set of capacities; and a colour capacity c. Orders are to be assigned to slabs, such that for each slab, the number of differently coloured orders does not exceed c and the sizes of the fitted orders do not exceed the capacity of the slab. The objective is to minimise the unused capacity (called *slack*) of the slabs (unused slabs have zero slack).

The SMSD MiniZinc model also contains symmetry-breaking constraints that (i) for any pair of slabs s_j and s_k, with $j < k$, required the total size of the fitted orders of s_j to be at most the total size of the fitted orders of s_k and (ii) for any pair of orders o_ℓ and o_m, with $\ell < m$ and where o_ℓ and o_m both have the same colour and size, require order o_ℓ to be placed in the same slab as o_m or a previous slab than o_m.

The dependency graph for each instance is acyclic, and the minimum-cardinality set of search variables is the set of slabs that the orders are placed in.

The initial supplied incumbent solution for each instance is automatically generated by assigning each order to a distinct slab.

7.2.2 Job Shop (JSP)

The job shop problem is a constrained optimisation problem that has a set $\mathcal{M}$ of machines and a set of jobs, where each job consists of a sequence of $|\mathcal{M}|$ tasks, each having a discrete duration, and the tasks of a job each require a distinct machine. For any job j, before the task at index i may be started, all previous tasks $1\,..\,(i-1)$ of j must have been completed. No two tasks requiring the same machine can be performed simultaneously. Once a machine has started working on a task, it must run it until completion. The objective is to minimise the latest completion time of all tasks.

[3] TSPTW instances: `https://myweb.uiowa.edu/bthoa/tsptwbenchmarkdatasets.htm`

The dependency graph for each instance is acyclic, and the minimum-cardinality set of search variables is the set of starting times of the tasks.

The initial supplied incumbent solution is to perform the tasks of each job sequentially, without any two tasks of any two jobs being performed simultaneously.

7.2.3 Travelling Salesperson with Time Windows (TSPTW)

Consider n locations that are to be visited, where one location is called the *depot*, with a travelling duration between each directed pair of locations, and for each location u an earliest visiting time e_u and a latest visiting time ℓ_u. A TSPTW *tour* is a Hamiltonian cycle of the weighted directed graph induced by the n locations as nodes, starting at the depot, visiting each location u exactly once and between times e_u and ℓ_u, with a zero visit duration. If the salesperson arrives at location u before e_u, then they have to wait until e_u before departing. The total duration of the tour is to be minimised.

Note that for other TSPTW problems, the objective function to be minimised is typically the total travel time, such as in [3]. However, and only for realism, we here instead minimise the total duration of the tour, including the wait times.

A MiniZinc model for TSPTW is in Listing 2. Lines 1–6 declare the number of locations, the depot location, the set of locations to be visited, the array of the earliest visiting times of the locations, the array of the latest visiting times of the locations, and the two-dimensional matrix with the travelling durations between the locations, respectively. On line 7, the array `pred` of variables is declared, where `pred[l]` denotes the location that is visited just before location `l`. On lines 8–9, the array `durFromPred` of variables is declared and is functionally defined by `pred`, where `durFromPred[l]` denotes the travelling duration from location `pred[l]` to location `l`. On line 10, the array `arrival` of variables is declared, where `arrival[l]` denotes the arrival time at location `l`. The arrival time back at the depot is the total duration of the tour. On lines 11–12, the array `departure` of parameters and variables is declared and is functionally defined by the arrival times at non-depot locations in `arrival`, where `departure[l]` denotes the departure time from location `l`; it differs from `arrival[l]` if `l` is the depot or the salesperson arrives at `l` before `early[l]` and has to wait there until `early[l]` before departing. Note that `departure[depot]` is the earliest visiting time of the depot, which is a parameter. On lines 13–14, the array `departurePred` of variables is declared and is functionally defined by `pred` and `departure`, where `departurePred[l]` is the departure time from location `pred[l]`. On lines 15–16, the arrival time at each location is functionally defined by `durFromPred` and `departurePred`. Line 17 enforces that the `pred` variables form a Hamiltonian cycle. Line 18 enforces that each location is visited before its latest visiting time. Finally, line 19 declares that `arrival[depot]` is to be minimised, which is the total duration of the tour.

For each location `l` variable `pred[l]` defines variable `durFromPred[l]`. The variables in the array `departure` are transitively functionally defined by themselves and the variables in the array `pred`, and the same is true for the variables in the arrays `departurePred` and `arrival`. Therefore, there is a cycle in the induced dependency graph containing the variables in the arrays `departure`, `departurePred`, and `arrival`. Using DCS, the greedily found low-cardinality set of search variables has the variables in the arrays `pred` and `departure`. The cyclic dependency graph for TSPTW with three locations corresponding to the MiniZinc model in Listing 2 is shown in Figure 3.

Listing 2 A MiniZinc model for the travelling salesperson with time windows (TSPTW) problem.

```
1  int: n;
2  int: depot = 1;
3  set of int: Locs = 1..n;
4  array[Locs] of int: early;
5  array[Locs] of int: late;
6  array[Locs, Locs] of int: duration;
7  array[Locs] of var Locs: pred;
8  array[Locs] of var 0..max(duration): durFromPred = [
9      duration[l, pred[l]] | l in Locs];
10 array[Locs] of var 0..max(late): arrival;
11 array[Locs] of var min(early)..max(late): departure =
12   [early[depot]] ++ [max(early[l], arrival[l]) | l in 2..n];
13 array[Locs] of var 0..sum(duration): departurePred = [
14     departure[pred[l]] | l in Locs];
15 constraint forall (l in Locs) (
16     arrival[l] = departurePred[l] + durFromPred[l]);
17 constraint circuit(pred);
18 constraint forall (l in Locs) (departure[l] <= late[l]);
19 solve minimize arrival[depot];
```

In order to generate initial incumbent solutions for the TSPTW instances, we created a MiniZinc model for a constrained satisfaction problem from Listing 2 by replacing line 19 with `solve satisfy`. We used the Gecode [6] CP solver (without LNS) with a timeout of 10 minutes to find for each instance a solution to be used as the initial incumbent solution. For some instances, no solution was found before timing out, and they are therefore not used.

8 Results

We ran our experiments on a desktop computer with an ASUS PRIME Z590-P motherboard, a 3.5 GHz Intel Core i9 11900K processor, and four 16 GB 3200 MT/s DDR4 memories, running Ubuntu 22.04.4 LTS with GCC (the GNU Compiler Collection) 11.

For each selection heuristic, both with DCS and without DCS, and each problem instance, we ran 10 independent runs, each under a timeout of 3 minutes from the same initial incumbent solution.

We used the scoring function $100 \cdot (o - b) \div i$ proposed in [1] and used in [24], where for each instance, the mean objective value over the 10 independent runs is denoted by o, the best found objective value over any run for the instance is denoted by b, and the objective value of the initial incumbent solution is denoted by i. Therefore, the scores range over the closed continuous interval $[0, 100]$, where a low score is better than a high one.

The results are shown in Table 1 and Figure 4. For all generic selection heuristics, using DCS improves performance for RCS, SMSD, and TSPTW, but worsens performance for JSP. These results support our conjectures that using DCS improves performance for RCS and SMSD, but does not improve performance for JSP. For JSP and each generic selection heuristic except VRG-LNS, using DCS has significantly worse performance than not using it. For JSP and VRG-LNS, using DCS has similar performance to not using it. However, there are a few JSP instances where using DCS with VRG-LNS, PG-LNS, and RPG-LNS

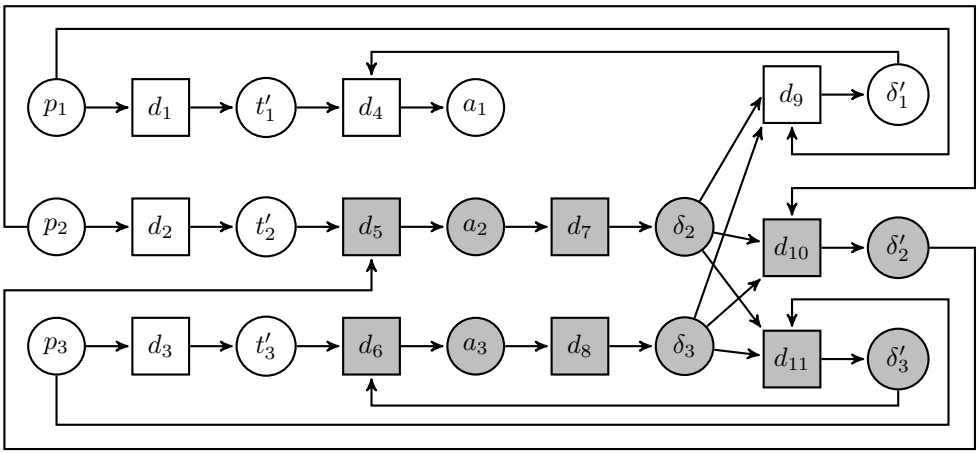


Figure 3 The cyclic dependency graph for the TSPTW MiniZinc model of Listing 2 with 3 locations (for ease of readability). Variable p_ℓ denotes `pred[`ℓ`]`, variable t'_ℓ denotes `durFromPred[`ℓ`]`, variable a_ℓ denotes `arrival[`ℓ`]`, variable δ_ℓ denotes `departure[`ℓ`]`, and variable δ'_ℓ denotes `departurePred[`ℓ`]`. The depot is location 1 and the objective variable is a_1. Vertices d_1, d_2, and d_3 correspond to the dependency constraints on lines 8–9; vertices d_4, d_5, and d_6 correspond to lines 15–16; vertices d_7 and d_8 correspond to lines 11–12; vertices d_9, d_{10}, and d_{11} correspond to lines 13–14; and the vertices with grey backgrounds together form an SCC, while each other SCC has a single vertex. The constraints on lines 17 and 18 are non-dependency constraints and are therefore absent in the graph.

has significantly better performance than when not using it. For RCS and each selection heuristic, the performance of using DCS is significantly better than when not using it. For both SMSD and TSPTW and each selection heuristic, the performance of using DCS is better than when not using it. Additionally, for RCS, SMSD, and TSPTW, using DCS makes the naïve randomised LNS competitive with the remaining more elaborate generic selection heuristics, even when they use DCS.

9 Conclusion and Future Work

We have presented our dependency curation scheme (DCS), which can be used with any (generic or problem-specific) LNS selection heuristic. We have compared the performance of a naïve generic randomised selection heuristic and more elaborate state-of-the-art generic selection heuristics from the literature both with and without DCS, revealing overall improved

Table 1 The mean score for 72 JSP instances, 79 RCS instances, 80 SMSD instances, and 112 TSPTW instances, each over 10 independent runs, both with and without the dependency curation scheme (DCS), for CIG-LNS, PG-LNS, randomised LNS, RPG-LNS, and VRG-LNS, where the "no" and "yes" columns denote if DCS was used or not. Boldface indicates the best performance on each row.

	CIG-LNS		PG-LNS		Randomised LNS		RPG-LNS		VRG-LNS	
DCS	no	yes	no	yes	no	yes	no	yes	no	yes
JSP	**0.13**	7.45	7.30	28.53	0.18	18.37	32.68	18.35	32.20	33.19
RCS	22.11	0.47	9.94	0.48	24.72	**0.34**	9.93	0.43	20.35	0.63
SMSD	1.62	0.87	4.06	**0.67**	1.14	0.85	5.64	0.73	6.45	0.81
TSPTW	4.65	1.82	3.67	**1.36**	4.87	2.18	3.71	1.80	5.41	2.20

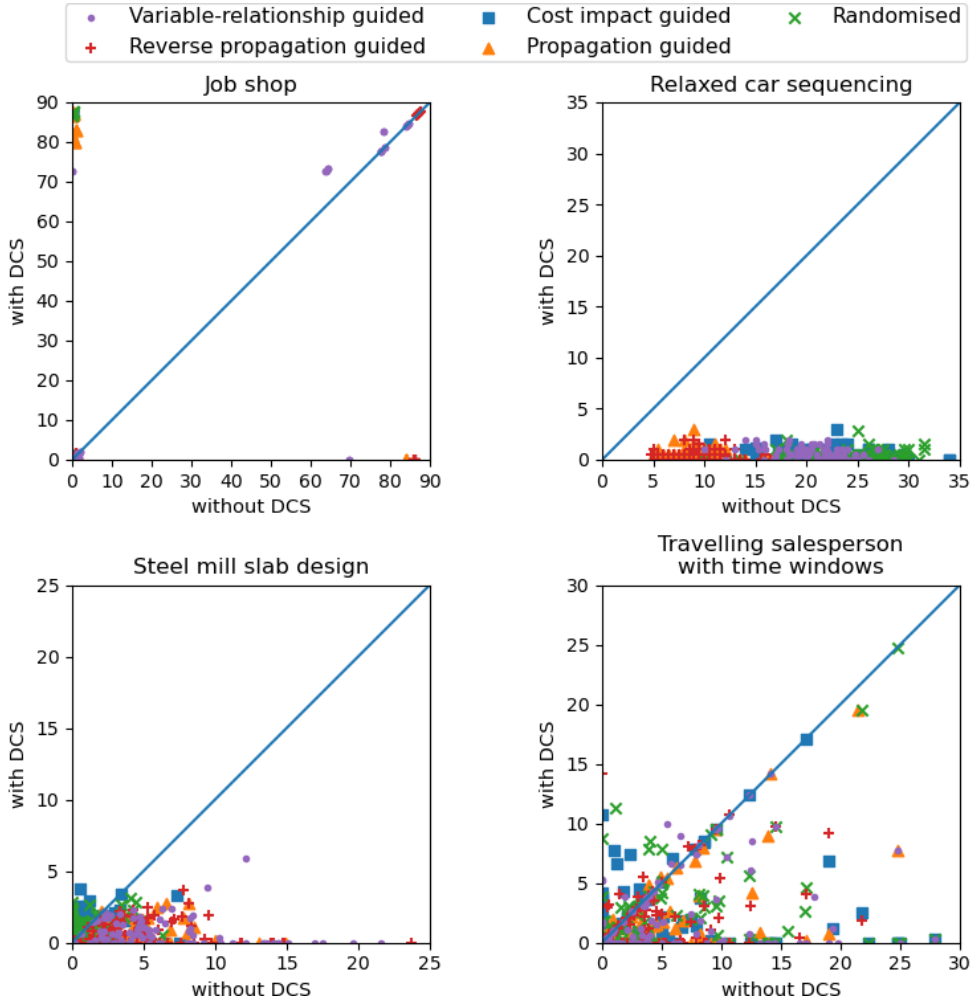


Figure 4 The scores for 72 JSP instances, 79 RCS instances, 80 SMSD instances, and 112 TSPTW instances, each over 10 independent runs, both with and without the dependency curation scheme (DCS), for VRG-LNS, RPG-LNS, CIG-LNS, PG-LNS, and randomised LNS. Each marker is the mean score over 10 independent runs for a selection heuristic, where a low score corresponds to a high-quality solution. Each marker over the diagonal line corresponds to DCS worsening performance. Conversely, each marker under the diagonal line corresponds to DCS improving performance.

performance when using DCS. Our experiments show that the performance of using DCS with the naïve randomised selection heuristic is competitive with the more elaborate state-of-the-art generic selection heuristics, even when they use DCS.

In future work, we intend to implement for our extension of the Gecode-based portfolio solver [11] our dependency curation scheme, so that the solver works without the modeller needing to provide (a generator that finds) a set of search variables. Once our scheme is implemented, we intend to compare the performance of the generic selection heuristics, both with and without DCS, on all optimisation problems in the MiniZinc [17] benchmark repository, to verify the results of this paper and to compare our extended Gecode-based portfolio solver with other state-of-the-art optimisation solvers. Additionally, we intend to extend the (non-portfolio) Gecode [6] CP solver by implementing our scheme and a naïve randomised generic LNS selection heuristic, thereby allowing a modeller to easily use competitive LNS for any MiniZinc constrained optimisation model.

Also, for VRG-LNS, a variable is selected for exclusion from the freeze set if it shares a constraint with a previously excluded variable x. When used with DCS, each variable y that does not share a constraint with x but that transitively functionally defines a variable that does so should be eligible for selection. Our idea is to make this modification to VRG-LNS and empirically test if it improves performance when used with DCS.

Finally, we intend to create a generic branching heuristic for CP-style tree search, where the variables that are selected to be branched over must be a subset of the low-cardinality set of search variables found by DCS.

References

1 H. Murat Afsar, Christian Artigues, Eric Bourreau, and Safia Kedad-Sidhoum. Machine reassignment problem: The ROADEF/EURO Challenge 2012. *Annals of Operations Research*, 242:1–17, 2016. doi:10.1007/s10479-016-2203-7.

2 Russell Bent and Pascal Van Hentenryck. A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows. *Computers and Operations Research*, 33(1):875–893, January 2006. doi:10.1016/j.cor.2004.08.001.

3 Gustav Björdal, Pierre Flener, and Justin Pearson. Generating compound moves in local search by hybridisation with complete search. In Louis-Martin Rousseau and Kostas Stergiou, editors, *CP-AI-OR 2019*, volume 11494 of *LNCS*, pages 95–111. Springer, 2019. doi:10.1007/978-3-030-19212-9_7.

4 Gustav Björdal, Jean-Noël Monette, Pierre Flener, and Justin Pearson. A constraint-based local search backend for MiniZinc. *Constraints*, 20(3):325–345, July 2015. doi:10.1007/s10601-015-9184-z.

5 Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In Yves Kodratoff, editor, *ECAI 1988*, pages 290–295. Pitman, 1988.

6 Gecode Team. Gecode: A generic constraint development environment, 2019. The Gecode solver and its MiniZinc backend are available at https://www.gecode.org.

7 Michel Gendreau, Alain Hertz, Gilbert Laporte, and Mihnea Stan. A generalized insertion heuristic for the traveling salesman problem with time windows. *Operations Research*, 46(3):330–335, 1998. doi:10.1287/opre.46.3.330.

8 Daniel Godard, Philippe Laborie, and Wim Nuijten. Randomized large neighborhood search for cumulative scheduling. In Susanne Biundo, Karen L. Myers, and Kanna Rajan, editors, *ICAPS 2005*, pages 81–89. AAAI Press, 2005. URL: http://www.aaai.org/Library/ICAPS/2005/icaps05-009.php.

9 Jayant Kalagnanam, Milind Dawande, Mark Trumbo, and Ho Lee. Inventory matching problems in the steel industry. *IBM Research Report, Computer Science/Mathematics*, April 1999.

10 Frej Knutar Lewander, Pierre Flener, and Justin Pearson. Invariant graph propagation in constraint-based local search. *Journal of Artificial Intelligence Research*, 2025. Forthcoming.

11 Dexter Leander. Building portfolio search in Gecode for MiniZinc. Master's thesis, Department of Information Technology, Uppsala University, Sweden, September 2024. Available at https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-533041, with the code at https://github.com/DLeander/gecode-dexter.

12 Michele Lombardi and Pierre Schaus. Cost impact guided LNS. In Helmut Simonis, editor, *CP-AI-OR 2014*, volume 8451 of *LNCS*, pages 293–300. Springer, 2014. doi:10.1007/978-3-319-07046-9_21.

13 Toni Mancini and Marco Cadoli. Exploiting functional dependencies in declarative problem specifications. *Artificial Intelligence*, 171(16–17):985–1010, November 2007. doi:10.1016/j.artint.2007.04.017.

14 Laurent Michel, Pierre Schaus, and Pascal Van Hentenryck. MiniCP: A lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1):133–184, 2021. The source code and teaching materials are available at `http://minicp.org`. `doi:10.1007/s12532-020-00190-7`.

15 Laurent Michel and Pascal Van Hentenryck. Localizer: A modeling language for local search. In Gert Smolka, editor, *CP 1997*, volume 1330 of *LNCS*, pages 237–251. Springer, 1997. `doi:10.1007/BFb0017443`.

16 Laurent Michel and Pascal Van Hentenryck. Localizer. *Constraints*, 5(1–2):43–84, 2000. `doi:10.1023/A:1009818401322`.

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 Bessière, editor, *CP 2007*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007. The MiniZinc toolchain is available at `https://www.minizinc.org`. `doi:10.1007/978-3-540-74970-7_38`.

18 Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided large neighborhood search. In Mark Wallace, editor, *CP 2004*, volume 3258 of *LNCS*, pages 468–481. Springer, 2004. `doi:10.1007/978-3-540-30201-8_35`.

19 David Pisinger and Stefan Ropke. Large neighborhood search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 272 of *ORMS*, chapter 4, pages 99–127. Springer, 2019. `doi:10.1007/978-3-319-91086-4_4`.

20 Charles Prud'homme, Xavier Lorca, and Narendra Jussien. Explanation-based large neighborhood search. *Constraints*, 19(4):339–379, October 2014. `doi:10.1007/s10601-014-9166-6`.

21 Pierre Schaus. Variable objective large neighborhood search: A practical approach to solve over-constrained problems. In Alexander Brodsky, editor, *ICTAI 2013*, pages 971–978. IEEE Computer Society, 2013. `doi:10.1109/ICTAI.2013.147`.

22 Pierre Schaus, Pascal Van Hentenryck, Jean-Noël Monette, Carleton Coffrin, Laurent Michel, and Yves Deville. Solving steel mill slab problems with constraint-based techniques: CP, LNS, and CBLS. *Constraints*, 16(2):125–147, April 2011. `doi:10.1007/s10601-010-9100-5`.

23 Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-François Puget, editors, *CP 1998*, volume 1520 of *LNCS*, pages 417–431. Springer, 1998. `doi:10.1007/3-540-49481-2_30`.

24 Filipe Souza, Diarmuid Grimes, and Barry O'Sullivan. An investigation of generic approaches to large neighbourhood search. In Paul Shaw, editor, *CP 2024*, volume 307 of *LIPIcs*, pages 39:1–39:10. Dagstuhl Publishing, 2024. `doi:10.4230/LIPIcs.CP.2024.39`.

25 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. `doi:10.1137/0201010`.

26 Charles Thomas and Pierre Schaus. Revisiting the self-adaptive large neighborhood search. In Willem-Jan van Hoeve, editor, *CP-AI-OR 2018*, volume 10848 of *LNCS*, pages 557–566. Springer, 2018. `doi:10.1007/978-3-319-93031-2_40`.

27 Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

28 Gerhard Wäscher and Thomas Gau. Heuristics for the integer one-dimensional cutting stock problem: A computational study. *OR Spektrum*, 18:131–144, 1996. `doi:10.1007/BF01539705`.

Practically Feasible Proof Logging for Pseudo-Boolean Optimization

Wietze Koops
Lund University, Sweden
University of Copenhagen, Denmark

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

Magnus O. Myreen
Chalmers University of Technology, Gothenburg, Sweden
University of Gothenburg, Sweden

Jakob Nordström
University of Copenhagen, Denmark
Lund University, Sweden

Andy Oertel
Lund University, Sweden
University of Copenhagen, Denmark

Yong Kiam Tan
Institute for Infocomm Research (I^2R), A*STAR, Singapore
Nanyang Technological University, Singapore

Marc Vinyals
University of Auckland, New Zealand

Abstract

Certifying solvers have long been standard for decision problems in Boolean satisfiability (SAT), allowing for proof logging and checking with very limited overhead, but developing similar tools for combinatorial optimization has remained a challenge. A recent promising approach covering a wide range of solving paradigms is pseudo-Boolean proof logging, but this has mostly consisted of proof-of-concept works far from delivering the performance required for real-world deployment.

In this work, we present an efficient toolchain based on *VeriPB* and *CakePB* for formally verified pseudo-Boolean optimization. We implement proof logging for the full range of techniques in the state-of-the-art solvers *RoundingSat* and *Sat4j*, including core-guided search and linear programming integration with Farkas certificates and cut generation. Our experimental evaluation shows that proof logging and checking performance in this much more expressive paradigm is now quite close to the level of SAT solving, and hence is clearly practically feasible.

2012 ACM Subject Classification Theory of computation → Logic and verification; Mathematics of computing → Combinatorial optimization

Keywords and phrases proof logging, certifying algorithms, combinatorial optimization, certification, pseudo-Boolean solving, 0–1 integer linear programming

Digital Object Identifier 10.4230/LIPIcs.CP.2025.21

Supplementary Material *Software (RoundingSat)*: `https://gitlab.com/MIAOresearch/software/roundingsat`, archived at `swh:1:rev:7116cbc7c228335d7a828fc9086639e4321fd5d1`
Software (Sat4j): `http://www.sat4j.org/`
archived at `swh:1:rev:adda8a7c5414b89b6f8ca1b5452c17526296104f`
Software (VeriPB): `https://gitlab.com/MIAOresearch/software/VeriPB` [70]
archived at `swh:1:rev:9dbb658ffb8d88815c3f3d8ff7fcefd6c6fc43db`
Software (CakePB): `https://gitlab.com/MIAOresearch/software/cakepb`
archived at `swh:1:rev:b29b4e967413ee4e8c3dcb1bbc78bd0d508d6817`
Dataset (Experiments): `https://zenodo.org/records/15628604`

Funding *Wietze Koops*: Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.
Daniel Le Berre: BLaSST ANR-21-CE25-0010.
Magnus O. Myreen: Swedish Research Council grant 2021-05165.
Jakob Nordström: Swedish Research Council grants 2016-00782 and 2024-05801 and Independent Research Fund Denmark grant 9040-00389B.

© Wietze Koops, Daniel Le Berre, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Yong Kiam Tan, and Marc Vinyals;
licensed under Creative Commons License CC-BY 4.0

Andy Oertel: Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.
Yong Kiam Tan: Singapore NRF Fellowship Programme NRF-NRFF16-2024-0002
Marc Vinyals: UoA FoS RDF 3729031

Acknowledgements We would like to acknowledge earlier work on *RoundingSat* by Jan Elffers, Jo Devriendt, and Stephan Gocht, and on *Sat4j* by Anthony Blomme and Romain Wallon, including preliminary contributions to proof logging. We are most grateful to Benjamin Bogø for useful feedback and for helping us to fix a bug in the proof logging code developed for this paper. Different subsets of the authors wish to thank the participants of the Dagstuhl workshops 22411 *Theory and Practice of SAT and Combinatorial Solving* and 23261 *SAT Encodings and Beyond* for several stimulating discussions. The first, fourth, and fifth authors also gratefully acknowledge that they have benefitted greatly from being part of the *Basic Algorithms Research Copenhagen (BARC)* environment financed by the Villum Investigator grant 54451. The computational experiments were enabled by resources provided by LUNARC at Lund University.

1 Introduction

Combinatorial search and optimization is a major success story in computer science, and the dramatic improvements in performance of combinatorial solvers over the last couple of decades allow them to be used for solving large-scale real-world problems in model checking [10], cryptanalysis [54], planning [64], kidney allocation for transplants [11, 12], protein design [2, 44], and many other application domains. But modern solvers are highly complex pieces of software, using sophisticated combinations of intelligent inference algorithms driven by intricate heuristics, and it is well documented that even the most mature state-of-the-art combinatorial solvers sometimes return "solutions" that do not satisfy the constraints, or erroneously claim infeasibility or optimality [1, 14, 20, 37]. Such errors could potentially have serious consequences for applications where correctness is a non-negotiable demand, and in more complex settings, where combinatorial solvers are used to solve subproblems, seemingly innocuous off-by-one mistakes can snowball into huge overall errors.

The Boolean satisfiability (SAT) community has pioneered the concept of *certifying solvers* as the most successful approach to date to address this problem. Certifying solvers do not only produce an answer, but also use *proof logging* to generate a machine-verifiable mathematical proof that this answer is correct. Many different SAT proof logging formats such as *RUP* [43], *TraceCheck* [9], *GRIT* [22], and *LRAT* [21] have been developed, with *DRAT* [46, 47, 71] established as the de facto standard; for the past decade, proof logging has been compulsory in the main track of the SAT competitions [63].

A big part of the success of SAT proof logging is that it is at the same time very easy to implement and very efficient to check. In the *DRAT* format, all that is required for proof logging is (essentially) for the solver to print the clauses it learns, meaning that the overhead of proof generation can be kept very small (say, less than 10% of solving time). Yet proof checkers are so fast that the time spent on verifying solver claims is within an order of magnitude of solving time, and can sometimes be made as efficient as solving at the price of adding additional proof details as in *LRAT* proofs [61]. Many SAT proof formats also come with formally verified proof checking backends [21, 22, 51, 66], meaning that the correctness of the final verdict is guaranteed by state-of-the-art formal verification techniques.

Inspired by the success of SAT proof logging, over the years there have been numerous attempts to develop proof logging in other combinatorial solving communities such as constraint programming [28, 60, 69], mixed integer programming [18, 30], satisfiability

modulo theories (SMT) solving [3, 4, 8, 48, 65], model counting [15, 17, 36], and automated planning [33, 34, 58]. However, most of these attempts have run into the problem that it is hard to design simple proof systems that can support the full range of solver reasoning techniques used. And if one instead provides specialized proof rules for different techniques, then proof systems quickly become very complex, making efficient proof checking challenging.

In the last few years, *pseudo-Boolean (PB) reasoning* with 0–1 linear constraints has emerged as a simple and yet expressive proof logging approach for a wide range of solving paradigms and techniques such as SAT-based optimization (MaxSAT) [5, 6, 49], subgraph solving [38, 39, 40], constraint programming [31, 41, 55, 56, 57], automated planning [27], and dynamic programming [23]. Although having a single, unified proof logging method for a large number of seemingly very different combinatorial paradigms appears quite attractive, the actual performance in terms of proof generation and checking has remained at a proof-of-concept stage, far from what would be required for deployment in production-grade software. It seems fair to say that no combinatorial optimization paradigm has been able to deliver proof logging with the efficiency corresponding to what is available for SAT decision problems.

In this work, we present – to the best of our knowledge, for the first time – highly efficient and practical certified solving for a combinatorial optimization problem, thus going beyond the decision problems considered in SAT solving. Using the pseudo-Boolean proof system *VeriPB* together with the formally verified backend *CakePB*, we provide a toolchain for linear pseudo-Boolean optimization with fully formally verified conclusions. We show how to implement proof logging for the state-of-the-art pseudo-Boolean solvers *RoundingSat* [25, 26, 32] and *Sat4j* [52], covering the full range of advanced techniques used in these solvers. For the main bottleneck in previous pseudo-Boolean proof logging works, namely proof checking, we can now provide formally verified conclusions within a factor 20 of the solving time, getting close to the overhead factor of 9 traditionally required in the SAT competitions – this is orders of magnitude faster than even the unverified proof checking in prior work.

To provide some perspective on why our work is a nontrivial contribution, let us briefly explain how SAT proof logging works and why a similar approach is not possible for pseudo-Boolean solving. The workhorse of SAT proof logging is so-called *reverse unit propagation (RUP)* [43, 68]. If SAT conflict analysis produces a clause, say, $C \doteq x_1 \vee x_2 \vee \overline{x}_3$, from the current constraint database $\mathcal{C}$, then in order to verify the correctness of this derivation it is sufficient to assert the negation $\neg C \doteq \overline{x}_1 \wedge \overline{x}_2 \wedge x_3$ and check that so-called *unit propagation* on the constraint database $\mathcal{C}$ leads to contradiction.

The concept of *RUP* can be generalized to pseudo-Boolean constraints [31], where it means achieving integer bounds consistency, but RUP checks are not sufficient to certify pseudo-Boolean solving. At the risk of getting a bit technical, consider the constraints

$$C_1 \doteq 2x + 2y + 2z + u + v \geq 5 \tag{1}$$

$$C_2 \doteq 2\overline{x} + 2\overline{y} + 2\overline{z} + u + v \geq 5 \tag{2}$$

and suppose that a pseudo-Boolean solver decides to set $x = 0$. Then the constraint C_1 propagates $y = z = 1$, which leads to a conflict with C_2. Division-based pseudo-Boolean conflict analysis now weakens away u and v from C_1 to get $2x + 2y + 2z \geq 3$, divides by 2 to obtain $x + y + z \geq 2$, and finally multiplies this constraint by 2 and adds to C_1 to learn

$$C_3 \doteq u + v \geq 3\,. \tag{3}$$

It is easy to see that this learned constraint can never be satisfied for $\{0, 1\}$-valued variables, and so the solver has derived contradiction and can terminate.

The details of the conflict analysis above are not too important, but the crucial observation is that the learned constraint C_3 cannot be verified by RUP, because none of the constraints C_1, C_2, or $\neg C_3 \doteq \overline{u} + \overline{v} \geq 2$ causes any unit propagations. A more expensive check would be to compute if the linear programming relaxation of these constraints yields an empty polytope – this would be another way of showing that C_3 follows from C_1 and C_2. But such a check also fails here, since $x = y = z = \frac{1}{2}$ and $u = v = 1$ is a fractional solution satisfying C_1, C_2, and $\neg C_3$. The point of this technical detour is to illustrate that SAT-style RUP proof logging cannot be expected to work for pseudo-Boolean solvers, but instead we have to provide much more explicit, syntactic, information about how solvers infer constraints.

Implementing proof logging for the state-of-the-art solvers *RoundingSat* and *Sat4j* presents further challenges. In addition to pseudo-Boolean conflict analysis, which maps very naturally to pseudo-Boolean reasoning steps, *RoundingSat* uses a number of more advanced techniques for which it is much less obvious how to express the reasoning in terms of pseudo-Boolean proof rules. In other cases, the natural proof logging approach turns out to lead to performance issues, which have to be circumvented by using more complex solutions. *Sat4j* does not cause as many problems in terms of different reasoning techniques, but has a much larger codebase developed over more than two decades. Adding proof generation to this codebase, and finding all the different places where the solver might, for instance, perform minor simplifications of constraints that do not really change anything in terms of semantics but cause syntactic checks to detect unexplained changes and fail, is highly nontrivial.

Our formally verified proof checking is performed in two stages, as is also commonly done for SAT solving. In the first stage, the unverified checker *VeriPB elaborates* the proof to a more restrictive format for which no search or propagation is required. This elaborated proof is then checked by the formally verified backend *CakePB*. We have made significant efforts to identify bottlenecks in proof elaboration and formal checking, and this work is a big part of the explanation for why the whole verified proof checking workflow is now so efficient.

The rest of this paper is organized as follows. After reviewing the basics of pseudo-Boolean reasoning and proof logging in Section 2, we discuss proof logging for advanced solving techniques in Sections 3 and 4. In Section 5 we report on our work on optimizing the proof logging and checking pipeline. We present our experimental evaluation in Section 6 and provide some concluding remarks in Section 7. Some additional technical details regarding our proof logging derivations are provided in Appendices A and B.

2 Preliminaries

We start with a condensed review of pseudo-Boolean reasoning, referring the reader to [13, 42] for more details on *VeriPB* and to [16] for more on proof systems in general. This is followed by a brief description of the concrete syntax and semantics of *VeriPB* proofs.

2.1 Pseudo-Boolean Reasoning and the Cutting Planes Proof System

We work exclusively with *Boolean variables*, i.e., variables taking values 0 or 1. A literal ℓ over a variable x is either x or its negation $\overline{x} = 1 - x$. By a *pseudo-Boolean (PB) constraint* we mean an inequality of the form $C \doteq \sum_i a_i \ell_i \geq A$, where a_i and A are integers (and where we write $\doteq$ to denote syntactic equality). We can assume without loss of generality that constraints are written in *normalized form* [45] with all *coefficients* a_i and the *degree* A being non-negative, and all literals being over distinct variables. A *pseudo-Boolean formula* is a conjunction $F \doteq \bigwedge_j C_j$ of pseudo-Boolean constraints.

A *substitution* ω is a function from variables to literals or 0 or 1. We extend the domain of substitutions to literals in the natural way by respecting the meaning of negation. For a constraint $C \doteq \sum_i a_i \ell_i \geq A$, we define $C\restriction_\omega \doteq \sum_i a_i \omega(\ell_i) \geq A$ as the constraint where we perform the substitution specified by ω (and simplify the resulting constraint). For a formula F, we define $F\restriction_\omega \doteq \bigwedge_j C_j\restriction_\omega$. We sometimes write $\ell_1 \mapsto \ell_2$ for $\omega(\ell_1) = \ell_2$ when the substitution ω is clear from context or immaterial.

An *assignment* α is a substitution that maps literals to $\{0, 1\}$ only. A *partial assignment* can also leave literals ℓ_i untouched by mapping ℓ_i to itself. A partial assignment ρ *extends* another partial assignment σ if ρ agrees with σ on all literals that σ maps to $\{0, 1\}$, i.e., if $\sigma(\ell) \in \{0, 1\}$ implies $\rho(\ell) = \sigma(\ell)$. A (partial) assignment α *satisfies* a constraint $C \doteq \sum_i a_i \ell_i \geq A$ if $\sum_{\alpha(\ell_i)=1} a_i \geq A$ (note that we assume here that C is written in normalized form). An assignment satisfies a formula $F \doteq \bigwedge_j C_j$ if it satisfies all its constraints, in which case we call it a *solution*. The *decision problem* for the pseudo-Boolean formula F is to determine whether F has a solution. In an *optimization problem*, we are additionally given an integer linear *objective* $f \doteq \sum_i w_i \ell_i$ that we want to minimize over all solutions to F.

We view the *cutting planes proof system* [19] as operating on pseudo-Boolean constraints in normalized form. Our derivation rules, which all preserve solutions, are as follows. We can always introduce the *literal axiom* $\ell_i \geq 0$ for any literal ℓ_i. Given two constraints $C_1 \doteq \sum_i a_i \ell_i \geq A$ and $C_2 \doteq \sum_i b_i \ell_i \geq B$, we can add them to obtain $C_1 + C_2 \doteq \sum_i (a_i + b_i)\ell_i \geq A + B$. For a positive integer c, we can multiply $C_1 \doteq \sum_i a_i \ell_i \geq A$ by c to get $c \cdot C_1 \doteq \sum_i (ca_i)\ell_i \geq cA$, or divide by c and round to get $\sum_i \lceil \frac{a_i}{c} \rceil \ell_i \geq \lceil \frac{A}{c} \rceil$. Finally, we can *saturate* a constraint $\sum_i a_i \ell_i \geq A$ to obtain $\sum_i \min\{a_i, A\}\ell_i \geq A$.

The *slack* of a normalized constraint $C \doteq \sum_i a_i \ell_i \geq A$ with respect to a partial assignment ρ is the sum of the coefficients of all non-falsified literals minus the degree, i.e.,

$$\mathit{slack}\left(\textstyle\sum_i a_i \ell_i \geq A; \rho\right) = \sum_{\rho(\ell_i) \neq 0} a_i - A\,, \tag{4}$$

and measures how close ρ is to falsifying C. If $\mathit{slack}(C; \rho) < 0$, then C is *conflicting* under ρ and no extension of ρ can satisfy C. If ρ does not assign ℓ_i but $0 \leq \mathit{slack}(C; \rho) < a_i$, then we say that C *propagates* ℓ_i under ρ, since any extension of ρ that satisfies C must set $\ell_i = 1$.

A *unit constraint* (or just *unit*) is a constraint $\ell \geq 1$. During *unit propagation* on F under ρ, we iteratively extend ρ by assigning to 1 all literals ℓ_i that are propagated by a constraint in F until either no further literals propagate or some constraint in F is conflicting. In the latter case, we say that F propagates to *conflict* under ρ. The negation of a constraint $C \doteq \sum_i a_i \ell_i \geq A$ is $\sum_i a_i \ell_i \leq A - 1$, which rewritten to normalized form becomes

$$\neg C \doteq \textstyle\sum_i a_i \bar{\ell}_i \geq \sum_i a_i - A + 1\,. \tag{5}$$

It is easy to verify that adding C and $\neg C$ yields the contradictory constraint $0 \geq 1$. We say that F implies C by *reverse unit propagation (RUP)* if $F \wedge \neg C$ propagates to conflict (in which case it is clear that any solution to F must also satisfy C).

2.2 *VeriPB* Syntax and Semantics

We next discuss the syntax and semantics of the *VeriPB* implementation [70] in more detail. The reader should be advised, however, that the presentation is streamlined and simplified to cover precisely the subset of the *VeriPB* system that is needed for this paper.

VeriPB maintains a database of constraints $\mathcal{C}$, where each constraint can be referred to using a *constraint ID* that is a positive integer (assigned consecutively as the derivation proceeds, with the first IDs $1, 2, \ldots, S$ identified with the S pseudo-Boolean constraints in the input formula). In our *VeriPB* code examples we will use displayed math equation labels as constraint IDs for ease of reference, so that the constraint

$$C_6 \doteq 2x_1 + 5x_2 + x_3 \geq 4 \tag{6}$$

is referred to using constraint ID `6`. To make it easier to refer to constraints, they can also be assigned @-labels that refer to their integer IDs when they are derived.

Cutting planes derivations are written in reverse Polish notation prefixed by `pol`. *VeriPB* uses `~` to represent negation, `x`i and `~x`i for the literal axioms $x_i \geq 0$ and $\overline{x}_i \geq 0$, and `+`, `*`, `d`, and `s` for addition, multiplication, division, and saturation, respectively. In addition, *VeriPB* supports literal *weakening* (denoted by `w`), which eliminates a term $a\ell$ from a constraint by adding $a \cdot (\overline{\ell} \geq 0)$ (where we note that ℓ and $\overline{\ell}$ cancel to leave an additive 1 when added since $\ell + \overline{\ell} = 1$ must always hold). To illustrate all of what has been said so far, the line

```
@newconstr pol 6 s ~x2 2 * + x3 w 2 d
```
$$x_1 + x_2 \geq 1 \tag{7}$$

first saturates constraint C_6 (yielding $2x_1 + 4x_2 + x_3 \geq 4$), then takes the literal axiom $\overline{x}_2 \geq 0$, multiplies it by 2 and adds it (yielding $2x_1 + 2x_2 + x_3 \geq 2$), then weakens on x_3 by adding the literal axiom $\overline{x}_3 \geq 0$ (yielding $2x_1 + 2x_2 \geq 1$), and finally divides by 2. The end result is the constraint $x_1 + x_2 \geq 1$, which is stored in the constraint database with the next available ID `7`, and the (optional) label `@newconstr` functions as an alias for this number in the rest of the proof. In our example derivations, we write out the derived constraint on the right for clarity, but stating this constraint is not part of the *VeriPB* syntax for `pol` lines.

Reverse unit propagation (RUP) steps can be *annotated* with information about which constraints propagate to contradiction, as in *LRAT* [21], or stated without such information, as in *DRAT* [71]. Thus, the line

```
rup +1 x2 >= 1 ; 6
```
$$x_2 \geq 1 \tag{8}$$

derives $x_2 \geq 1$ after successful unit propagation to conflict on $\{2x_1 + 5x_2 + x_3 \geq 4, \overline{x}_2 \geq 1\}$, i.e., the constraint `6` together with the negation of $x_2 \geq 1$. Without the annotation `6`, *VeriPB* would instead propagate on the full constraint database and the negation of $x_2 \geq 1$ (which will always work, but might be less efficient). *VeriPB* also supports simple *syntactic implication* `ia`, which checks whether a specified constraint C can be derived from a single constraint in the database by adding literal axioms and saturating. A constraint ID can be added as an annotation to specify which concrete constraint syntactically implies C.

All derivation rules described so far are convenient shorthands for cutting planes derivations, but *VeriPB* also has two *strengthening rules* for deriving non-implied constraints that are guaranteed not to change the satisfiability status or optimal value of the input. We will not need the *dominance-based strengthening rule* in this paper, and so we ignore it and focus instead on the *redundance-based strengthening*, or *redundance* for short. A constraint C is inferred by redundance by specifying a *witness* substitution ω and proving the implication

$$\mathcal{C} \cup \{\neg C\} \vDash (\mathcal{C} \cup \{C\})\restriction_\omega \cup \{f\restriction_\omega \leq f\} \tag{9}$$

by providing cutting planes subproofs of all constraints on the right-hand side (which we refer to as *proof obligations*). In practice, many of these subproofs can be omitted since *VeriPB* will infer them automatically by, e.g, RUP or syntactic implication. As an example, if y_1 is a fresh variable that has not appeared in the derivation before, then we can define y_1 to imply $2x_1 + x_2 + x_3 \geq 3$ by the derivation step

```
red +3 ~y1 +2 x1 +1 x2 +1 x3 >= 3 ; y1 -> 0
```
$$3\overline{y}_1 + 2x_1 + x_2 + x_3 \geq 3 \tag{10}$$

introducing the constraint (10), which has the intended meaning since $y_1 = 1$ clearly forces the inequality $2x_1 + x_2 + x_3 \geq 3$ to hold. This derivation step is valid since the witness $\omega = \{y_1 \mapsto 0\}$ only affects the new constraint $3\overline{y}_1 + 2x_1 + x_2 + x_3 \geq 3$, which it fixes to true, and all other proof obligations are identical to the premises on the left-hand side or vacuous. In general, any subproofs required are provided after the witness specification delimited by `begin` and `end`. In case only one proof obligation needs an explicit proof and this proof only uses one constraint from $\mathcal{C}$, we will refer to that constraint as the *justification*.

When the witness is empty, the redundance rule allows to derive a constraint C using proof by contradiction. Namely, in this case one of the required implications to prove is $\mathcal{C} \cup \{\neg C\} \models C$, which since $C + \neg C \doteq 0 \geq 1$ is equivalent to deriving contradiction.

Solutions to the input formula F can be specified using the `sol` rule, which checks if the specified assignment is a valid solution. For an optimization problem with objective function $f \doteq \sum_i w_i \ell_i$ to be minimized, the `soli` rule also adds a *solution-improving constraint*

$$\sum_i w_i \ell_i \leq -1 + \sum_i w_i \cdot \alpha(\ell_i) \tag{11}$$

to the constraint database (but in normalized form), which enforces that all future solutions will yield a better objective value than α.

A *VeriPB* proof for the decision problem for F should establish either satisfiability by logging a solution to F or else unsatisfiability by deriving contradiction (in the form of a constraint that has negative slack with respect to the empty assignment). For a (feasible) optimization problem, a proof that the optimal value of the objective f lies in the interval $[LB, UB]$ should log a solution α to F achieving $f(\alpha) = UB$ and derive a constraint $f \geq LB$.

3 Proof Logging for Linear Programming Integration

Proof logging for the basic conflict-driven search in the pseudo-Boolean solvers *RoundingSat* and *Sat4j* is relatively straightforward, since both solvers use the cutting planes proof system for their conflict analysis. However, *RoundingSat* is tightly integrated with a linear programming (LP) solver [25], which adds extra complications as explained in this section.

Most aspects of the LP reasoning in *RoundingSat* can be expressed in terms of (positive) linear combinations of constraints and can therefore be logged directly with `pol` lines, and Chvátal-Gomory cut generation is the same as cutting planes division. However, *mixed-integer rounding (MIR)* cuts [53] are not natively supported by the *VeriPB* proof system, and the cut generation procedure turns inequalities into equalities by introducing non-Boolean *integral slack variables*, which cannot be directly expressed in the proof system.

As shown in [25], the MIR cut with divisor $d \in \mathbb{N}^+$ applied to $\sum_i a_i \ell_i \geq A$ yields

$$\sum_i \left(\min\{a_i \bmod d, A \bmod d\} + \left\lfloor \frac{a_i}{d} \right\rfloor (A \bmod d) \right) \ell_i \geq \left\lceil \frac{A}{d} \right\rceil (A \bmod d)\,, \tag{12}$$

which is in general stronger than the constraint

$$\sum_i \left(\left\lceil \frac{a_i}{d} \right\rceil (A \bmod d) \right) \ell_i \geq \left\lceil \frac{A}{d} \right\rceil (A \bmod d) \tag{13}$$

that we obtain by cutting planes division from $\sum_i a_i \ell_i \geq A$ with divisor d and then multiplying by $A \bmod d$, which we call the *multiplier* of the MIR cut. To see this, note that when $0 < a_i \bmod d < A \bmod d$, (12) has a smaller coefficient for the literal ℓ_i than (13).

We illustrate with an example how MIR cuts are generated in *RoundingSat*. As input, consider the two constraints

$$C_{14} \doteq 4x_1 + 7x_2 + 5x_3 + 3x_5 \geq 9\,, \tag{14}$$
$$C_{15} \doteq 3x_1 + 2x_2 + 2x_4 \geq 3\,, \tag{15}$$

and suppose that the multipliers are $\lambda_{14} = 1$ and $\lambda_{15} = 4$, the divisor is $d = 5$, and $P = \{x_1, x_4\}$ is the set of variables on which we *partially weaken* (i.e., add literal axioms to reduce the coefficients to the largest multiple of the divisor d). For purposes of computation, we introduce an integral slack variable $s_j \geq 0$ for each constraint C_j to obtain the equalities

$$C'_{14} \doteq 4x_1 + 7x_2 + 5x_3 + 3x_5 - s_{14} = 9\,, \tag{16}$$
$$C'_{15} \doteq 3x_1 + 2x_2 + 2x_4 - s_{15} = 3\,. \tag{17}$$

We take the linear combination

$$\textstyle\sum_j \lambda_j C'_j \doteq 16x_1 + 15x_2 + 5x_3 + 8x_4 + 3x_5 - s_{14} - 4s_{15} = 21 \tag{18}$$

and continue working on the greater-than-or-equal part of this equality. Partially weakening on x_1 and x_4 by adding $\overline{x}_1 \geq 0$ and $3 \cdot (\overline{x}_4 \geq 0)$ yields

$$15x_1 + 15x_2 + 5x_3 + 5x_4 + 3x_5 - s_{14} - 4s_{15} \geq 17\,. \tag{19}$$

Next, we apply a MIR cut with divisor $d = 5$ to get the inequality

$$6x_1 + 6x_2 + 2x_3 + 2x_4 + 2x_5 - s_{15} \geq 8\,. \tag{20}$$

For later use, we note that the multiplier of this MIR cut is $H = (-3) \bmod 5 = 2$. To obtain the final cut, we *subtract* C'_{15} to cancel the integral slack variable s_{15}, which yields

$$3x_1 + 4x_2 + 2x_3 + 2x_5 \geq 5\,. \tag{21}$$

Our formal *VeriPB* derivation of the MIR cut (21) is a proof by contradiction starting from the negation of this constraint, which is

$$3\overline{x}_1 + 4\overline{x}_2 + 2\overline{x}_3 + 2\overline{x}_5 \geq 7\,. \tag{22}$$

We first add $\lambda_{14} = 1$ times C_{14}, but we postpone adding C_{15}. Intuitively, this is done in order to be able to compensate for the subtraction later. In general, we add $\lambda_j C_j$ for all j such that $\lambda_j \leq H$, where H is the multiplier of the MIR cut. This results in the constraint

$$x_1 + 3x_2 + 3x_3 + x_5 \geq 5\,. \tag{23}$$

Next, we partially weaken on the variables in P, to reduce their coefficients to the largest smaller multiple of $d - H = 3$. In this case, we add $\overline{x}_1 \geq 0$, which yields

$$3x_2 + 3x_3 + x_5 \geq 4\,. \tag{24}$$

Now we divide by $d - H = 3$, add $H = 2$ times this to (22), and finally add C_{15}, which yields the sequence of constraints

$$x_2 + x_3 + x_5 \geq 2 \tag{25}$$
$$3\overline{x}_1 + 2\overline{x}_2 \geq 5 \tag{26}$$
$$2x_4 \geq 3 \tag{27}$$

ending with contradiction, establishing correctness of the cut (21). In general, we add $(d - \lambda_j)C_j$ for all j such that $\lambda_j > H$. The code fragment

```
red +3 x1 +4 x2 +2 x3 +2 x5 >= 5 ; ; begin
  pol 28 14 +
  pol 29 ~x1 + 3 d
  pol 28 30 2 * + 15 +
end
```

$$3\overline{x}_1 + 4\overline{x}_2 + 2\overline{x}_3 + 2\overline{x}_5 \geq 7 \quad (28)$$
$$x_1 + 3x_2 + 3x_3 + x_5 \geq 5 \quad (29)$$
$$x_2 + x_3 + x_5 \geq 2 \quad (30)$$
$$2x_4 \geq 3 \quad (31)$$
$$3x_1 + 4x_2 + 2x_3 + 2x_5 \geq 5 \quad (32)$$

shows how the MIR cut example can be formalized as a valid *VeriPB* derivation.

In Appendix A, we describe how proof logging for MIR cuts is done in full generality.

4 Proof Logging for Pseudo-Boolean Core-Guided Optimization

RoundingSat solves optimization problems using linear search, core-guided search, or a "hybrid" combination of the two [26]. Proof logging for linear search is straightforward, since the solution-improving constraint added by the solver is what is derived by the `soli` rule. For core-guided optimization the proof logging is more complicated, but we explain by example below. The overall approach is similar to core-guided optimization in MaxSAT [6], from which the technique has been imported, but the details differ significantly since the solver deals with general pseudo-Boolean constraints rather than clauses. Pseudo-Boolean constraints in the *VeriPB* code examples of this section are mathematically formatted to improve clarity, which is not correct *VeriPB* syntax.

Consider a minimization problem with objective function $f_o \doteq x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 + 6x_6$. We initially have an implicit lower bound $f_o \geq 0$.

We begin by running the solver with an additional assumption that $f_o = 0$, equivalently $x_1 = \ldots = x_6 = 0$. We either find a solution, which must be optimal, or our assumption is too strong. In that case the solver finds a conflict, say $3x_2 + 2x_3 + x_4 + x_5 \geq 4$, which we derive with some proof line

```
pol (...)
```

$$3x_2 + 2x_3 + x_4 + x_5 \geq 4 \quad (33)$$

as done for conflict analysis. Such a *core constraint* shows that the assumptions are inconsistent with the input formula. We then turn the core constraint (33) into a cardinality constraint, also called a *cardinality core*. The most obvious way to do so is to divide by the largest coefficient in the constraint, but it is possible to obtain stronger cardinality constraints through a more complicated algorithm. All of these algorithms can be logged as a sequence of weakening, saturation, and division steps. In our case, the derivation step

```
pol 33 3 d
```

$$x_2 + x_3 + x_4 + x_5 \geq 2 \quad (34)$$

divides by 3 to get $x_2 + x_3 + x_4 + x_5 \geq 2$. We proceed to rewrite the objective using the cardinality core (34). To do so, we introduce two new slack variables y_3 and y_4 so that we can write $x_2+x_3+x_4+x_5 = 2+y_3+y_4$. Equivalently, writing $\Delta_1 \doteq x_2+x_3+x_4+x_5-(2+y_3+y_4)$, we have $\Delta_1 = 0$. We log the two sides of this equality using the redundance rule. We first log the *at-most* constraint $\Delta_1 \leq 0$ with witness $y_3 = y_4 = 1$ and trivial justification. Then we log the *at-least* constraint $\Delta_1 \geq 0$ in two steps, with witnesses $y_3 = 0$ and $y_4 = 0$, and justifications the cardinality core (34) and the previous step (36):

`red` $x_2 + x_3 + x_4 + x_5 \leq 2 + y_3 + y_4$ `; y3 -> 1 y4 -> 1` (35)

`red` $x_2 + x_3 + x_4 + x_5 \geq 2 + y_3$ `; y3 -> 0` (36)

`red` $x_2 + x_3 + x_4 + x_5 \geq 2 + y_3 + y_4$ `; y4 -> 0` (37)

To speed up search we give slack variables the additional meaning that y_j is true if and only if $x_2 + x_3 + x_4 + x_5 \geq j$. We do so with an ordering constraint of the form $y_3 \geq y_4$, which we derive using the redundance rule and witness swapping y_3 and y_4:

`red` $y_3 \geq y_4$ `; y3 -> y4 y4 -> y3` (38)

We then proceed to rewrite the objective to $f_r \doteq f_o - \Delta$, where $\Delta \doteq 2\Delta_1$ is the difference between the reformulated and original objectives. The multiplier 2 is the largest number that keeps all coefficients of f_r positive. The new objective is $f_r \doteq x_1 + x_3 + 2x_4 + 3x_5 + 6x_6 + 2y_3 + 2y_4 + 4$, and, because $\Delta = 0$, it is equivalent to the original objective. Furthermore, the new objective contains a constant term 4, which gives us a lower bound of 4. We log the *global reformulation constraint* $\Delta \geq 0$ as 2 times (37), then log the *objective lower bound* $f_o \geq 4$ by first logging the trivial constraint $f_r \geq 4$ and then adding $\Delta \geq 0$:

`pol 37 2 *` $2x_2 + 2x_3 + 2x_4 + 2x_5 \geq 4 + 2y_3 + 2y_4$ (39)

`rup` $x_1 + x_3 + 2x_4 + 3x_5 + 6x_6 + 2y_3 + 2y_4 \geq 0$ (40)

`pol 40 39 +` $x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 + 6x_6 \geq 4$ (41)

At this point we run the solver with assumptions $x_i = y_i = 0$ again. Suppose that we now obtain a conflict constraint $x_4 + x_5 + x_6 + y_3 \geq 1$, which is already a cardinality core.

We now have $\Delta_2 \doteq x_4 + x_5 + x_6 + y_3 - (1 + z_2 + z_3 + z_4)$. We introduce at-most, at-least, and ordering constraints:

`red` $x_4 + x_5 + x_6 + y_3 \leq 1 + z_2 + z_3 + z_4$ `; z2 -> 1 z3 -> 1 z4 -> 1` (42)

`red` $x_4 + x_5 + x_6 + y_3 \geq 1 + z_2$ `; z2 -> 0` (43)

`red` $x_4 + x_5 + x_6 + y_3 \geq 1 + z_2 + z_3$ `; z3 -> 0` (44)

`red` $x_4 + x_5 + x_6 + y_3 \geq 1 + z_2 + z_3 + z_4$ `; z4 -> 0` (45)

`red` $z_2 \geq z_3$ `; z2 -> z3 z3 -> z2` (46)

`red` $z_3 \geq z_4$ `; z2 -> z4 z3 -> z2 z4 -> z3` (47)

We rewrite the objective again, with $\Delta \doteq 2\Delta_1 + 2\Delta_2$. The new objective is $f_r \doteq x_1 + x_3 + x_5 + 4x_6 + 2y_4 + 2z_2 + 2z_3 + 2z_4 + 6$. We derive the global reformulation constraint $\Delta \geq 0$ from the previous global reformulation constraint, and then we log the lower bound $f_o \geq 6$:

`pol 39 45 2 * +` $2x_2 + 2x_3 + 4x_4 + 4x_5 + 2x_6 \geq 6 + 2y_4 + 2z_2 + 2z_3 + 2z_4$ (48)

`rup` $x_1 + x_3 + x_5 + 4x_6 + 2y_4 + 2z_2 + 2z_3 + 2z_4 \geq 0$ (49)

`pol 49 48 +` $x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 + 6x_6 \geq 6$ (50)

We run the solver with assumptions $x_i = y_i = z_i = 0$ yet another time. Suppose that these assumptions extend to a solution. We log the solution using `soli` and obtain the objective upper bound $f_o \leq 6$, completing the proof.

Expert readers may note that in practice it is too slow to introduce all the slack variables when we reformulate the objective, and we need to introduce variables lazily instead. This means that when we find the first cardinality core, we only introduce y_3. Therefore, we define $\Delta_1^{(1)} \doteq x_2 + x_3 + x_4 + x_5 - (2 + y_3)$, where the notation $\Delta_i^{(j)}$ refers to the i-th cardinality core after j slack variables have been introduced. We log constraints encoding that y_3 is true if and only if $x_2 + x_3 + x_4 + x_5 \geq 3$ is true. We first log the at-most constraint encoding that $x_2 + x_3 + x_4 + x_5 \geq 3$ implies y_3

`red` $x_2 + x_3 + x_4 + x_5 \leq 2 + 2y_3$ `; y3 -> 1` (51)

with witness $y_3 = 1$ and trivial justification, and then the at-least constraint encoding that y_3 implies $x_1 + x_3 + x_4 + x_5 \geq 3$, or equivalently $\Delta_1^{(1)} \geq 0$,

`red` $x_2 + x_3 + x_4 + x_5 \geq 2 + y_3$ `; y3 -> 0` (52)

with witness $y_3 = 0$ and justification the cardinality core (34).

We rewrite the objective to $f_r \doteq f_o - \Delta$ with $\Delta \doteq 2\Delta_1^{(1)}$. Observe that we no longer have $\Delta = 0$ but only $\Delta \geq 0$, therefore all we can say about the objective is $f_o \leq f_r$. Nevertheless, this is enough to derive a lower bound on the objective.

Suppose that our next core constraint is $x_4 + x_5 + x_6 + y_3 \geq 1$. We now have $\Delta_2^{(1)} \doteq x_4 + x_5 + x_6 + y_3 - (1 + z_2)$. We introduce at-most and at-least constraints:

`red` $x_4 + x_5 + x_6 + y_3 \leq 1 + 3z_2$ `; z2 -> 1` (53)

`red` $x_4 + x_5 + x_6 + y_3 \geq 1 + z_2$ `; z2 -> 0` (54)

Since y_3 appears in this cardinality core and will disappear from the objective, we introduce y_4 and extend $\Delta_1^{(1)}$ to $\Delta_1^{(2)} \doteq x_2 + x_3 + x_4 + x_5 - (2 + y_3 + y_4)$. We first log the at-most constraint with witness $y_4 = y_3$ and justification the previous at-most constraint (51),

`red` $x_2 + x_3 + x_4 + x_5 \leq 2 + y_3 + y_4$ `; y4 -> y3` (55)

and then the at-least constraint with witness $y_4 = 0$ and justification the previous at-least constraint (52).

`red` $x_2 + x_3 + x_4 + x_5 \geq 2 + y_3 + y_4$ `; y4 -> 0` (56)

We derive the ordering constraint from previous constraints instead of by redundance, as there might be constraints containing y_3 which are not obvious to derive under the witness:

`pol 56 51 +` $y_3 \geq y_4$ (57)

We rewrite the objective to $f_r \doteq f_o - \Delta$ with $\Delta \doteq 2\Delta_1^{(2)} + 2\Delta_2^{(1)}$. As $\Delta_1^{(1)}$ has been extended to $\Delta_1^{(2)}$, we cannot obtain the new global reformulation constraint from the previous global reformulation constraint, therefore we rederive it from scratch.

`pol 56 2 * 54 2 * +` $2x_2 + 2x_3 + 4x_4 + 4x_5 + 2x_6 \geq 6 + 2y_4 + 2z_2$ (58)

Assuming that we next find an optimal solution, we complete the proof analogously to the eager case.

We provide proofs of correctness and some further constructions in Appendix B. Most importantly, we touched on eager and lazy sum encodings, but we also support a third reified encoding. Additionally, if we have upper bounds on the objective, we can use that information to derive unit constraints by hardening, and to derive upper bounds on cores, which in turn can be used to derive stronger at-most constraints and stop extending lazy variables earlier.

5 Implementation

In this section, we discuss our work on implementing proof logging in the pseudo-Boolean solvers *Sat4j* and *RoundingSat*, and how we have optimized the proof checkers *VeriPB* and *CakePB* to get close to the levels of SAT proof logging performance.

5.1 Proof Logging in *Sat4j*

The solver *Sat4j* [52] for SAT and pseudo-Boolean problems was designed more than 20 years ago, when proof logging was not a standard feature, but the fine-grained event-driven design allows for precise visualization and remote control of solver behaviour [7]. As such, when *DRUP* proof logging became mandatory in the SAT 2013 competition, including support for this was very fast. In 2024 incremental proof logging with *IDRUP* [35] was added. Such support for incremental solver calls also paved the way for proof logging for optimization.

The *Sat4j* optimization engine uses linear solution-improving search, for which proof logging is straightforward. However, for the conflict analysis the solver has two versions *Sat4j Resolution* with SAT-style, clausal, reasoning and *Sat4j Cutting Planes* with cutting-planes-based reasoning. Proof logging for SAT conflict analysis is easy to implement with RUP statements and is robust to syntactic changes of the underlying constraints, but dealing with cutting planes derivations requires exact control of the syntactic representation of constraints.

Sat4j inherits from *MiniSat* [29] that the constraints are simplified during parsing, so that, e.g., satisfied constraints are ignored and variables with fixed values are removed from constraints. In addition, unit propagation and saturation are applied. RUP proofs can essentially be written as if these simplifications were not performed, but explicit cutting planes proofs quickly fail when there is a mismatch between the original input constraint and the simplified constraint stored in the constraint database. Dealing with this turns out to be a formidable challenge. *Sat4j* has more than 30 different implementations of constraints with individual simplification rules, and any changes to the 15-20-year-old code must be limited so that none of the many software packages that depend on the solver breaks.

The @-labels for constraints discussed in Section 2.2 make book-keeping easier, in that we can represent falsified literals $x_j = 0$ by unit constraints $\overline{x}_j \geq 1$, which we can derive by RUP and give labels `@xj`. Removing a falsified literal then amounts to adding `@xj` to the constraint, whereas satisfied literals can simply be removed by weakening.

To illustrate how this works, suppose that the Mth input constraint read by *Sat4j* is $6x_1 + 2x_2 + x_3 + x_4 \geq 5$ and that $x_4 = 0$ is already known at this point. Then the falsified literal x_4 is removed to derive $6x_1 + 2x_2 + x_3 \geq 5$; saturation is applied to yield $5x_1 + 2x_2 + x_3 \geq 5$; and finally $x_1 = 1$ is propagated. The first two steps yields proof lines

`@x4 rup 1 ~x4 >= 1` $\overline{x}_4 \geq 1$ (59)

`@M pol` M `@x4 + s` $5x_1 + 2x_2 + x_3 \geq 5$ (60)

while the unit $x_1 \geq 1$ is only derived later when needed. Note that because of the intermediate derivations the simplified version of the Mth input constraint will *not* get constraint ID M, but by defining the label `@M` we can avoid having to keep track of the exact number.

Continuing our example, suppose that the solver later reads the Nth input constraint $3\overline{x}_1 + 2x_2 + 2x_3 + \overline{x}_4 + 2x_5 \geq 3$. Then the solver simplifies this to $2x_2 + 2x_3 + 2x_5 \geq 2$, which is semantically equivalent to the clause $x_2 \vee x_3 \vee x_5$, and justifies this with the proof steps

`@x1 rup 1 x1 >= 1` $x_1 \geq 1$ (61)

`@N pol` N `@x1 3 * + x4 w` $2x_2 + 2x_3 + 2x_5 \geq 2$ (62)

Using labels for input constraints and fixed literals make proof generation in *Sat4j* significantly simpler in that we just use a flag to keep track of whether the Nth constraint was simplified or not, and to refer to this constraint by `@N` or N accordingly. Another helpful *VeriPB* feature during implementation was that the proof can be terminated at any time and checked for correctness with respect to the upper and lower bound on the objective known at that time.

5.2 Proof Logging in *RoundingSat*

An important part of making the proof logging in *RoundingSat* efficient is to deal with fixed variable assignments $x \geq 1$ or $x \leq 0$ inferred by the solver (i.e., unit constraints). As in several previous works (e.g., [6, 30, 38, 50]), our work on implementing proof logging helped us find and eliminate bugs which previous extensive testing had failed to detect.

RoundingSat simplifies all derived constraints $C \doteq \sum_i a_i \ell_i \geq A$ by removing unit constraints. Satisfied literals $\ell_i \geq 1$ can just be weakened away, but falsified literals $\ell_j \leq 0$ require a justification. Adding $a_j \cdot (\ell_j \leq 0)$ to C for all falsified literals causes performance issues during proof checking. A better solution turns out to be to derive the fully simplified version of C in one go using a RUP statement annotated with the units $\ell_j \leq 0$ used.

In contrast to constraints learned during conflict analysis, unit constraints are typically discovered during propagation before the solver has made any decisions (i.e., at *decision level* 0). To see why proof logging for this scenario requires care, consider the constraints

$$C_1 \doteq x_2 + x_3 + 2x_4 + 2x_5 + 4x_6 \geq 7 \tag{63}$$

$$C_2 \doteq \overline{x}_1 + \overline{x}_2 + 2\overline{x}_3 + 2\overline{x}_4 + 4\overline{x}_5 + 4\overline{x}_6 \geq 7 \tag{64}$$

unit propagating $x_6 = 1$, $x_5 = 0$, $x_4 = 1$, $x_3 = 0$, $x_2 = 1$, and $x_1 = 0$ in that order. Explicit cutting planes derivations of these units would require adding all previous unit constraints to either C_1 or C_2, yielding a quadratic number of steps overall. And this is not just a theoretical pathological case – we observed instances where constraints propagated many units and where writing proofs for such units dominated the cost of proof logging. What we do instead is to derive such units using annotated RUP, explicitly mentioning which constraints are needed for propagation. While checking such RUP steps can in theory incur a linear overhead for "ping-ponging" propagations as between constraints (63) and (64) above, this does not seem to be a problem in practice.

The fact that cutting planes proof depends on the exact syntactic representation of constraints is not only a source of complications, as discussed in Section 5.1, but can also help detect bugs, including mistakes that do not affect soundness but makes solver reasoning weaker than it should. When generating MIR cuts as in (12), *RoundingSat* computed the coefficient $\left(\frac{a_i}{d} + 1\right)(A \bmod d)$ for ℓ_i when $d \mid a_i$ and $a_i < 0$ instead of the correct value $\frac{a_i}{d}(A \bmod d)$. Since the resulting coefficient is always larger, this is not unsound, but yields a weaker constraint than intended. This was discovered since the constraint derived in the proof did not match what was in the solver constraint database.

Another example, which is related to the discussion at the end of Section 4, is that when an objective upper bound $f \leq UB$ is known and X is a linear form such that $f - wX \geq 0$ holds, then the solver derived $X \leq \lfloor UB/w \rfloor$. However, in this scenario the constraint database in fact contains the solution-improving constraint $f \leq UB - 1$, and so the solver should infer the stronger constraint $X \leq \lfloor (UB - 1)/w \rfloor$ instead. This was again discovered thanks to a mismatch between proof log and constraint database.

5.3 Optimizations in *VeriPB*

We have done substantial work on *VeriPB* to improve performance and robustness of proof checking and elaboration. Much of this concerns technical implementation details that are hard to explain briefly, but we try to give two examples below.

To make it convenient to log solutions with the `sol` and `soli` rules, *VeriPB* is guaranteed to apply unit propagation before checking that the provided assignment satisfies all constraints. Such propagation can cause substantial overhead, especially when solutions are logged

frequently during optimization. Since pseudo-Boolean solvers can easily specify the full assignment to all variables, we modified the default behaviour of *VeriPB* to check first if the provided assignment satisfies all constraints before applying unit propagation.

We have also improved *autoproving*, i.e., the ability of *VeriPB* to automatically infer constraints, by changing the syntactic implication check so that it first applies unit propagation to the premises. Such autoproving is crucial to be able to minimize the size of the generated proofs, especially for strengthening rules, but involves delicate trade-offs with performance.

5.4 Optimizations in *CakePB*

Significant effort has also been invested to speed up *CakePB* and to update the formal verification following our code changes. The verification shows that, upon successful proof checking by *CakePB*, the conclusions of the proof log are sound with respect to the semantics of the input decision or optimization problem. Formally verified compilation [67] extends this guarantee all the way to the *CakePB* machine code (see [39, Section 3.3] for more details).

The *CakePB* updates include implementing and verifying standard code optimizations such as introducing tail recursion and carefully choosing data structures, but below we discuss some more interesting changes specific to pseudo-Boolean proof checking.

1. **RUP with annotations:** In prior work [49], *VeriPB* elaborated RUP steps into cutting planes proofs to be checked by *CakePB*. This meant that *VeriPB* expanded RUP statements to cutting planes derivations, which *CakePB* then checked using its general procedure for cutting planes. A much more efficient approach is for *CakePB* to support annotated RUP steps, which include a list of constraints in the order they propagate. The verified RUP implementation in *CakePB* tracks these propagations with a mutable bitset, which allows units to be calculated in linear passes over the annotated constraints.
2. **Simplifications for cutting planes proofs:** *CakePB* now makes a syntactic simplification pass on `pol` lines, where adjacent literal weakening steps are merged into a single simultaneous step internally, and adjacent additions of literal axioms (possibly multiplied by constants) are also merged. In this way *CakePB* can use a single pass over the resulting merged step to carry out all these operations.
3. **Space-efficient occurrence mapping:** *CakePB* tracks a mapping from variables to the IDs of all constraints containing them [49]; this is used as an efficient heuristic for skipping proof obligations in strengthening rules. However, such a mapping is space-inefficient as many of the variables are never looked up. Our new heuristic instead tracks only a fixed number of constraints per variable. We also changed the mapping to use an in-place array-based representation as opposed to the previous, purely functional implementation.

6 Experiments

We have run experiments on the 397 decision instances and 478 optimization instances in the Pseudo-Boolean Competition 2024 [62] on hardware with i5-1145G7 CPUs, 14GB of available RAM, a 100GB solid state drive as storage, and Rocky Linux 8.10 as operating system. We first ran the solvers without proof logging to establish a baseline, and only kept instances solved within a one-hour time limit. We then ran solvers with proof logging, had *VeriPB* elaborate the proofs, and fed them to the formally verified checker *CakePB*. Noise due to very small running times was smoothed by computing all ratios as $(1 + x)/(1 + y)$.

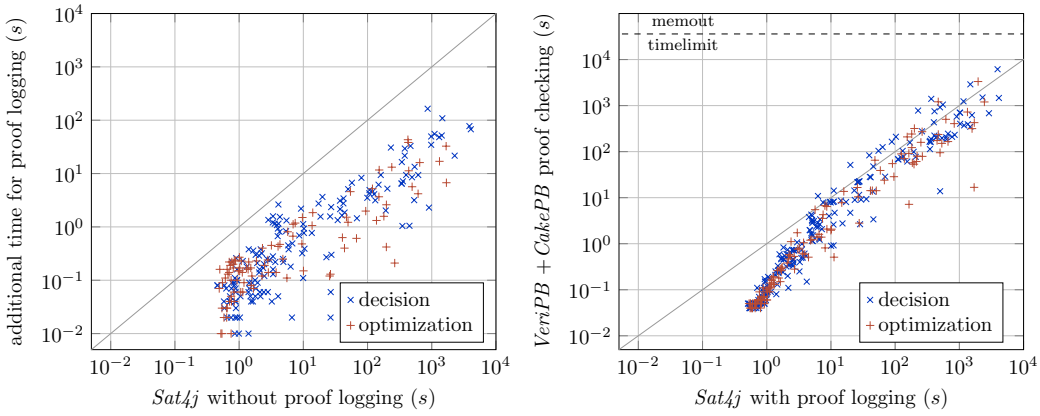


Figure 1 *Sat4j Cutting Planes* overhead for proof logging and logging versus checking time.

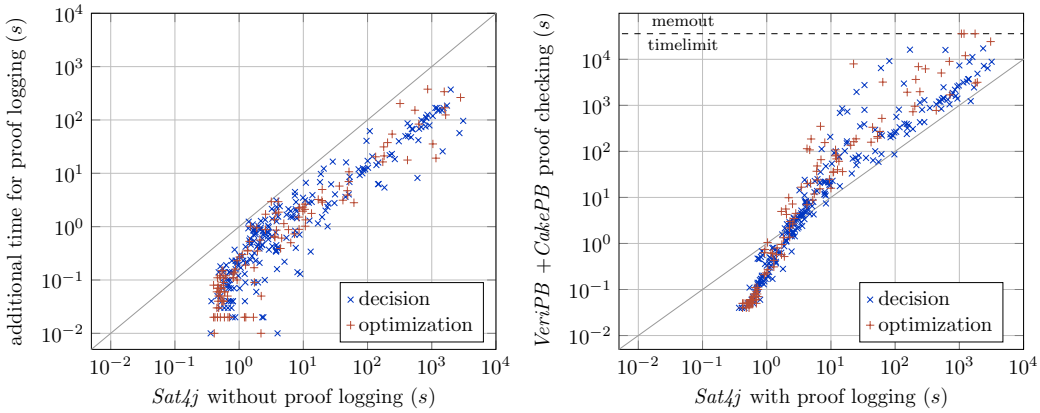


Figure 2 *Sat4j Resolution* overhead for proof logging and logging versus checking time.

Sat4j Cutting Planes (Figure 1) solves 199 decision and 123 optimization instances. The worst overhead for proof generation is 51.7%; 95% of the instances have an overhead below 14.2%; and the median is 2.6%. All instances are successfully checked within 6,158s. The worst ratio of checking to solving time is 3.83; 95% of the instances are checked within a factor 1.60, and the median is a factor 0.54, which is even faster than solving.

Sat4j Resolution (Figure 2) solves 243 decision and 123 optimization instances. The worst proof logging overhead is 71%; 95% of the instances have an overhead below 42%; and the median is 9.7%. For 3 instances, *VeriPB* times out after 10h. The worst ratio of proof checking time compared to solving time is a whopping 338. However, 95% of the instances are checked within a factor 20.9; and the median is a factor 1.33. The main reason for the much larger overheads for checking is that *Sat4j Resolution* essentially writes SAT-style *DRAT* proofs using RUP steps without annotations. This means that *VeriPB* has to perform propagation for every clause learned, but pseudo-Boolean propagation is more expensive than clausal propagation and *VeriPB* has no special handling of clausal proofs.

RoundingSat (Figure 3) solves 297 decision and 258 optimization instances. The worst proof logging overhead is 46.2%; 95% of the instances have an overhead below 21.1%; and the median is 2.7%. One optimization instance cannot be elaborated by *VeriPB* because the proofs exceed the 100GB of available disk space. On 3 decision and 4 optimization instances

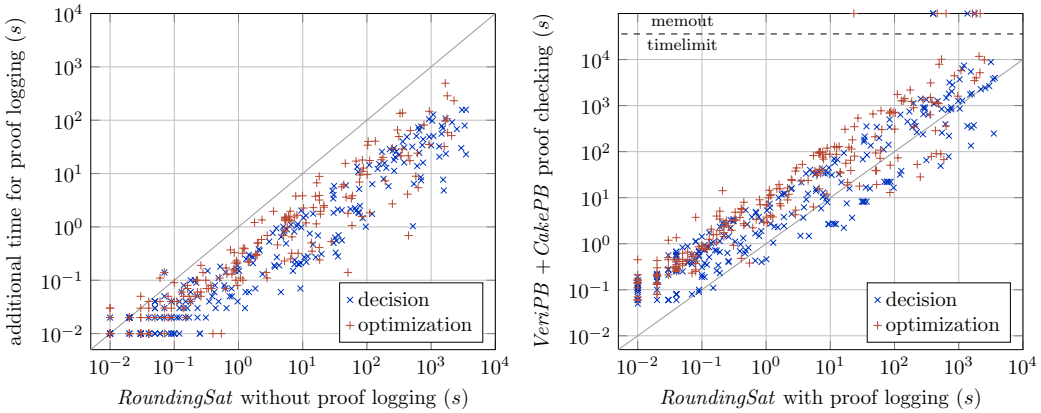


Figure 3 *RoundingSat* overhead for proof logging and logging versus checking time.

VeriPB runs out of memory, and on a further 1 decision and 1 optimization instance *CakePB* does. All remaining instances are checked within 11,730s. The worst ratio of checking to solving time is 19.17; 95% of the instances are checked within a factor 9.22; and the median is a factor 1.43. This is arguably quite close to what would be expected of SAT proof logging.

7 Concluding Remarks

In this work, we present a practically feasible and fully formally verified toolchain based on *VeriPB* and *CakePB* for certified pseudo-Boolean solving and optimization. Our work covers the full range of techniques in the state-of-the-art solvers *RoundingSat* and *Sat4j*, and so should be eminently possible to adapt to other pseudo-Boolean solvers.

From a solver author perspective, however, a key concern for adoption of proof logging is ease of use. One of the lessons learned from our work is that proof logging for basic constraint simplifications such as removing fixed variables is both common and surprisingly tricky to implement correctly and efficiently. Further investigations are warranted into how such simplifications could be handled more smoothly in the proof system. Also, adding a proof rule for mixed integer rounding (MIR) cuts would greatly simplify the proof logging for solvers that employ MIR cut generation techniques from mixed integer programming.

Regarding *VeriPB* performance, a substantial amount of time is spent on writing and parsing files. Introducing a binary file format for pseudo-Boolean formulas and proofs would alleviate this issue. Checking reverse unit propagation (RUP) steps is another bottleneck. For *Sat4j Resolution* the obvious remedy would be to implement RUP with annotations. In general, more sophisticated propagation algorithms [24, 59] would probably be helpful. However, for SAT-like proofs with *DRAT*-style, non-annotated, RUP steps, implementing a dedicated clausal checker along the lines of [71] inside *VeriPB* appears to be the best way to get truly efficient proof checking, but this seems like an engineering rather than a research problem.

For verified proof checking with *CakePB*, working on PB solving has led to several improvements on top of earlier optimizations for subgraph solving [39] and MaxSAT preprocessing [49]. Profiling shows the potential for further efficiency gains in checking strengthening steps and cutting planes derivations. Parallelizing the proof elaboration by *VeriPB* and checking by *CakePB* also seems worth exploring, especially since both tools support streaming proof files.

To the best of our knowledge, we present the first certified solving toolchain for an optimization problem beyond SAT with performance approaching that of SAT proof logging tools. We are hopeful that our work could point the way towards practically feasible certified solving also for stronger paradigms such as constraint programming and mixed integer programming.

References

1 Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP '18)*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, August 2018. doi:10.1007/978-3-319-98334-9_46.

2 Katherine I. Albanese, Sophie Barbe, Shunsuke Tagami, Derek N. Woolfson, and Thomas Schiex. Computational protein design. *Nature Reviews Methods Primers*, 5(13), February 2025.

3 Bruno Andreotti, Hanna Lachnitt, and Haniel Barbosa. Carcara: An efficient proof checker and elaborator for SMT proofs in the alethe format. In *Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '23)*, volume 13993 of *Lecture Notes in Computer Science*, pages 367–386. Springer, April 2023. doi:10.1007/978-3-031-30823-9_19.

4 Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli, and Clark Barrett. Flexible proof production in an industrial-strength SMT solver. In *Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR '22)*, volume 13385 of *Lecture Notes in Computer Science*, pages 15–35. Springer, August 2022. doi:10.1007/978-3-031-10769-6_3.

5 Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesande. Certifying without loss of generality reasoning in solution-improving maximum satisfiability. In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:28, September 2024. doi:10.4230/LIPICS.CP.2024.4.

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. Springer, July 2023. doi:10.1007/978-3-031-38499-8_1.

7 Daniel Le Berre and Stéphanie Roussel. Sat4j 2.3.2: on the fly solver configuration: System description. *Journal on Satisfiability, Boolean Modelling and Computation*, 8(3/4):197–202, August 2012.

8 Frédéric Besson, Pascal Fontaine, and Laurent Théry. A flexible proof format for SMT: a proposal. In *Proceedings of the 1st Workshop on Proof eXchange for Theorem Proving (PxTP '11)*, pages 15–26, August 2011. URL: https://pxtp2011.loria.fr/PxTP2011.pdf#page=19.

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

10 Armin Biere and Daniel Kröning. SAT-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, chapter 10, pages 277–303. Springer, December 2018. doi:10.1007/978-3-319-10575-8_10.

11 Péter Biró, Joris van de Klundert, David F. Manlove, William Pettersson, Tommy Andersson, Lisa Burnapp, Pavel Chromy, Pablo Delgado, Piotr Dworczak, Bernadette Haase, Aline Hemke, Rachel Johnson, Xenia Klimentova, Dirk Kuypers, Alessandro Nanni Costa, Bart Smeulders, Frits C. R. Spieksma, María O. Valentín, and Ana Viana. Modelling and optimisation in European kidney exchange programmes. *European Journal of Operational Research*, 291(2):447–456, June 2021. doi:10.1016/J.EJOR.2019.09.006.

12 Péter Biró, Bernadette Haase-Kromwijk, Tommy Andersson, Eyjólfur Ingi Ásgeirsson, Tatiana Baltesová, Ioannis Boletis, Catarina Bolotinha, Gregor Bond, Georg Böhmig, Lisa Burnapp, Katarína Cechlárová, Paola Di Ciaccio, Jiri Fronek, Karine Hadaya, Aline Hemke, Christian Jacquelinet, Rachel Johnson, Rafal Kieszek, Dirk R. Kuypers, Ruthanne Leishman, Marie-Alice Macher, David Manlove, Georgia Menoudakou, Mikko Salonen, Bart Smeulders, Vito Sparacino, Frits C. R. Spieksma, María Oliva Valentín, Nic Wilson, and Joris van der Klundert. Building kidney exchange programmes in Europe – An overview of exchange practice and activities. *Transplantation*, 103(7):1514–1522, June 2019.

13 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, August 2023. Preliminary version in *AAAI '22*. doi:10.1613/JAIR.1.14296.

14 Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT '09)*, pages 1–5, August 2009.

15 Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Certified knowledge compilation with application to verified model counting. In *Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT '23)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:20, 2023. doi:10.4230/LIPICS.SAT.2023.6.

16 Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 7, pages 233–350. IOS Press, 2nd edition, February 2021. doi:10.3233/FAIA200990.

17 Florent Capelli. Knowledge compilation languages as proof systems. In Mikolás Janota and Inês Lynce, editors, *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*, volume 11628 of *Lecture Notes in Computer Science*, pages 90–99. Springer, 2019. doi:10.1007/978-3-030-24258-9_6.

18 Kevin K. H. Cheung, Ambros M. Gleixner, and Daniel E. Steffy. Verifying integer programming results. In *Proceedings of the 19th International Conference on Integer Programming and Combinatorial Optimization (IPCO '17)*, volume 10328 of *Lecture Notes in Computer Science*, pages 148–160. Springer, June 2017. doi:10.1007/978-3-319-59250-3_13.

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

20 William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013. doi:10.1007/S12532-013-0055-6.

21 Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, August 2017. doi:10.1007/978-3-319-63046-5_14.

22 Luís Cruz-Filipe, João P. Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*, volume 10205 of *Lecture Notes in Computer Science*, pages 118–135. Springer, April 2017. doi:10.1007/978-3-662-54577-5_7.

23 Emir Demirović, Ciaran McCreesh, Matthew McIlree, Jakob Nordström, Andy Oertel, and Konstantin Sidorov. Pseudo-Boolean reasoning about states and transitions to certify dynamic programming and decision diagram algorithms. In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, September 2024. doi:10.4230/LIPICS.CP.2024.9.

24 Jo Devriendt. Watched propagation of 0-1 integer linear constraints. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 160–176. Springer, September 2020. doi:10.1007/978-3-030-58475-7_10.

25 Jo Devriendt, Ambros Gleixner, and Jakob Nordström. Learn to relax: Integrating 0-1 integer linear programming with pseudo-Boolean conflict-driven search. *Constraints*, 26(1–4):26–55, October 2021. Preliminary version in *CPAIOR '20*. doi:10.1007/S10601-020-09318-X.

26 Jo Devriendt, Stephan Gocht, Emir Demirović, Jakob Nordström, and Peter Stuckey. Cutting to the core of pseudo-Boolean optimization: Combining core-guided search with cutting planes reasoning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3750–3758, February 2021. doi:10.1609/AAAI.V35I5.16492.

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

28 Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining alldifferent. In *Proceedings of the 35th Australasian Computer Science Conference (ACSC '12)*, pages 115–124, January 2012. URL: http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV122Downing.html.

29 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03), Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004. doi:10.1007/978-3-540-24605-3_37.

30 Leon Eifler and Ambros Gleixner. A computational status update for exact rational mixed integer programming. *Mathematical Programming*, 197(2):793–812, February 2023. doi:10.1007/S10107-021-01749-5.

31 Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020. doi:10.1609/AAAI.V34I02.5507.

32 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-Boolean solving. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18)*, pages 1291–1299, July 2018. doi:10.24963/IJCAI.2018/180.

33 Salomé Eriksson, Gabriele Röger, and Malte Helmert. Unsolvability certificates for classical planning. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS '17)*, pages 88–97, June 2017. URL: https://aaai.org/ocs/index.php/ICAPS/ICAPS17/paper/view/15734.

34 Salomé Eriksson, Gabriele Röger, and Malte Helmert. A proof system for unsolvable planning tasks. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS '18)*, pages 65–73, June 2018. URL: https://aaai.org/ocs/index.php/ICAPS/ICAPS18/paper/view/17760.

35 Katalin Fazekas, Florian Pollitt, Mathias Fleury, and Armin Biere. Certifying incremental SAT solving. In *Proceedings of the 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2024)*, volume 100 of *EPiC Series in Computing*, pages 321–340, May 2024. doi:10.29007/PDCC.

36 Johannes Klaus Fichte, Markus Hecher, and Valentin Roland. Proofs for propositional model counting. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:24, August 2022. doi:10.4230/LIPICS.SAT.2022.30.

37 Xavier Gillard, Pierre Schaus, and Yves Deville. SolverCheck: Declarative testing of constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, volume 11802 of *Lecture Notes in Computer Science*, pages 565–582. Springer, October 2019. doi:10.1007/978-3-030-30048-7_33.

38 Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020. doi:10.1007/978-3-030-58475-7_20.

39 Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, February 2024. doi:10.1609/AAAI.V38I8.28642.

40 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020. doi:10.24963/IJCAI.2020/158.

41 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, August 2022. doi:10.4230/LIPICS.CP.2022.25.

42 Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021. doi:10.1609/AAAI.V35I5.16494.

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

44 Mark A. Hallen and Bruce Randall Donald. Protein design by provable algorithms. *Communications of the ACM*, 62(10):76–84, October 2019. doi:10.1145/3338124.

45 Peter L. Hammer and Sergiu Rudeanu. *Boolean Methods in Operations Research and Related Areas*. Springer Verlag, New York, 1968.

46 Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, pages 181–188, October 2013. URL: https://ieeexplore.ieee.org/document/6679408/.

47 Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, June 2013. doi:10.1007/978-3-642-38574-2_24.

48 Jochen Hoenicke and Tanja Schindler. A simple proof format for SMT. In *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories (SMT '22)*, volume 3185 of *CEUR Workshop Proceedings*, pages 54–70, August 2022. URL: https://ceur-ws.org/Vol-3185/paper9527.pdf.

49 Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen, and Jakob Nordström. Certified MaxSAT preprocessing. In *Proceedings of the 12th International Joint Conference on Automated Reasoning (IJCAR '24)*, volume 14739 of *Lecture Notes in Computer Science*, pages 396–418. Springer, July 2024. doi:10.1007/978-3-031-63498-7_24.

50 Sonja Kraiczy and Ciaran McCreesh. Solving graph homomorphism and subgraph isomorphism problems faster through clique neighbourhood constraints. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI '21)*, pages 1396–1402, August 2021. doi:10.24963/IJCAI.2021/193.

51 Peter Lammich. Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning*, 64(3):513–532, March 2020. Extended version of paper in *CADE* 2017. doi:10.1007/S10817-019-09525-Z.

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

53 Hugues Marchand and Laurence A. Wolsey. Aggregation and mixed integer rounding to solve MIPs. *Operations Research*, 49(3):325–468, June 2001.

54 Fabio Massacci and Laura Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24:165–203, February 2000. doi:10.1023/A:1006326723002.

55 Matthew McIlree and Ciaran McCreesh. Proof logging for smart extensional constraints. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP '23)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:17, August 2023. doi:10.4230/LIPICS.CP.2023.26.

56 Matthew McIlree and Ciaran McCreesh. Certifying bounds propagation for integer multiplication constraints. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI '25)*, pages 11309–11317, February-March 2025. doi:10.1609/AAAI.V39I11.33230.

57 Matthew McIlree, Ciaran McCreesh, and Jakob Nordström. Proof logging for the circuit constraint. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14743 of *Lecture Notes in Computer Science*, pages 38–55. Springer, May 2024. doi:10.1007/978-3-031-60599-4_3.

58 Esther Mugdan, Remo Christen, and Salomé Eriksson. Optimality certificates for classical planning. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling (ICAPS '23)*, pages 286–294, July 2023. doi:10.1609/ICAPS.V33I1.27206.

59 Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Rui Zhao. Speeding up pseudo-Boolean propagation. In *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT '24)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:18, August 2024. doi:10.4230/LIPICS.SAT.2024.22.

60 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, January 2009. doi:10.1007/S10601-008-9064-X.

61 Florian Pollitt, Mathias Fleury, and Armin Biere. Faster LRAT checking than solving with CaDiCaL. In *Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT '23)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:12, July 2023. doi:10.4230/LIPICS.SAT.2023.21.

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

63 The International SAT Competitions web page. https://satcompetition.github.io.

64 Dominik Schreiber. Lilotane: A lifted SAT-based approach to hierarchical planning. *Journal of Artificial Intelligence Research*, 70:1117–1181, March 2021. doi:10.1613/JAIR.1.12520.

65 Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). In *Proceedings of the 7th Workshop on Proof eXchange for Theorem Proving (PxTP '21)*, volume 336 of *Electronic Proceedings in Theoretical Computer Science*, pages 49–54, July 2021. doi:10.4204/EPTCS.336.6.

66 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake_lpr: Verified propagation redundancy checking in CakeML. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '21)*, volume 12652 of *Lecture Notes in Computer Science*, pages 223–241. Springer, March-April 2021. doi:10.1007/978-3-030-72013-1_12.

67 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2:1–e2:57, February 2019.

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

69 Michael Veksler and Ofer Strichman. A proof-producing CSP solver. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI '10)*, pages 204–209, July 2010. doi:10.1609/AAAI.V24I1.7543.

70 VeriPB: Verifier for pseudo-Boolean proofs. https://gitlab.com/MIAOresearch/software/VeriPB.

71 Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014. doi:10.1007/978-3-319-09284-3_31.

Algorithm 1 MIR cut generation in *RoundingSat*.

input : Set of indices J, list $\vec{C}$ of constraints $C_j \doteq \sum_j a_{ij}x_j \geq A_j$ for $j \in J$, list $\vec{\lambda}$ of multipliers $\lambda_j \in \mathbb{Z}$ for $j \in J$, divisor $d \in \mathbb{N}^+$, variable set P

1 CutGeneration$(J, \vec{C}, \vec{\lambda}, d, P)$:
2 **for** $j \in J$ **do**
3 define $C'_j \doteq \sum_j a_{ij}x_j - s_j = A_j$; `// Introduce slack variables`
4 $D \leftarrow \sum_j \lambda_j C'_j$;
5 partially weaken x_i for all $i \in P$;
6 $B \leftarrow \mathsf{RHS}(D)$;
7 $D \leftarrow \mathsf{MIR}(D, d)$;
8 **for** $j \in J$ **do**
9 add $M(-\lambda_j, d, B \bmod d)$ times C'_j to D ; `// Cancel slack variables`
10 **return** D;

A Proof Logging for MIR Cut Generation

We start by analyzing the cut computed by *RoundingSat*, as summarized in Algorithm 1. Let inequalities $C_j \doteq \sum_i a_{ij}x_i \geq A_j$, multipliers λ_j, and a divisor d be given. We introduce an integer-valued *integral slack variable* $s_j \geq 0$ for each constraint C_j to get the equality $C'_j \doteq \sum_i a_{ij}x_i - s_j = A_j$. The constraint $D \doteq \sum_j \lambda_j C'_j$ can then, after grouping terms by variable, be written as

$$\sum_i \left(\sum_j \lambda_j a_{ij}\right) x_i - \sum_j \lambda_j s_j = \sum_j \lambda_j A_j. \tag{65}$$

Write $b'_i = \sum_j \lambda_j a_{ij}$ and let P be the set of variables on which we *partially weaken* (which means that we add literal axioms to reduce the coefficients of the variables in P to the largest smaller multiple of the divisor d). Partially weakening on the variables in P yields

$$\sum_{i \notin P} b'_i x_i + \sum_{i \in P} d\left\lfloor \frac{b'_i}{d} \right\rfloor x_i + \sum_j (-\lambda_j) s_j = \sum_j \lambda_j A_j - \sum_{i \in P} (b'_i \bmod d). \tag{66}$$

Write $B = \sum_j \lambda_j A_j - \sum_{i \in P}(b'_i \bmod d)$, and let $b_i = b'_i$ for $i \notin P$ and $b_i = d\lfloor b'_i/d \rfloor$ for $i \in P$. Next, we apply a MIR cut with divisor d. Write $H = B \bmod d$, and write $M(a, d, H) = \lfloor a/d \rfloor H + \min(a \bmod d, H)$. This yields:

$$\sum_i M(b_i, d, H) x_i + \sum_j M(-\lambda_j, d, H) s_j \geq \lceil B/d \rceil H. \tag{67}$$

We now cancel the integral slack variables by adding

$$\sum_j M(-\lambda_j, d, H) \left(\sum_i a_{ij}x_i - s_j\right) = \sum_j M(-\lambda_j, d, H) A_j. \tag{68}$$

Write $f_i = \sum_j M(-\lambda_j, d, H) a_{ij}$ and $F = \sum_j M(-\lambda_j, d, H) A_j$. Then this yields

$$\sum_i \big(M(b_i, d, H) + f_i\big) x_i \geq \lceil B/d \rceil H + F, \tag{69}$$

which is the final cut derived by *RoundingSat* and returned by Algorithm 1.

We note that the cut only depends on the values of the multipliers modulo d. Hence, we may assume without loss of generality that $0 \leq \lambda_j \leq d - 1$ for all j. We now show how to proof log the cut. We prove (69) by contradiction. The negation of (69) is given by

$$\sum_i \left(-M(b_i, d, H) - f_i\right) x_i \geq -\lceil B/d \rceil H - F + 1. \tag{70}$$

Write $\delta = d - H$ and $\mu_j = -M(-\lambda_j, d, H)$. We now want to add constraint C_j with multiplier $\lambda_j - \mu_j - \delta \mathbf{1}_{\mu_j > 0}$, which we show to be nonnegative in Lemma 1. Intuitively, the $-\mu_j$ cancels out the f_i and F in (70), while we need the $-\delta \mathbf{1}_{\mu_j>0}$ later on to cancel the f_i again.

▶ **Lemma 1.** *We have* $\lambda_j - \mu_j - \delta \mathbf{1}_{\mu_j>0} \geq 0$.

Proof. If $\lambda_j = 0$, then $\mu_j = 0$ and hence the claim follows.

Otherwise, we have $0 \leq d - \lambda_j \leq d - 1$, and hence $(-\lambda_j) \bmod d = d - \lambda_j$.

If $0 < \lambda_j \leq \delta$, then $d - \lambda_j \geq H$, so $\mu_j = M(-\lambda_j, d, H) = -H + \min(d - \lambda_j, H) = 0$ and hence $\lambda_j - \mu_j - (d - H)\mathbf{1}_{\mu_j>0} = \lambda_j \geq 0$, as required.

If $\lambda_j > \delta$, we have $M(-\lambda_j, d, H) = -H + \min(d - \lambda_j, H) = -H + d - \lambda_j > 0$. Then we have $\lambda_j - \mu_j - \delta \mathbf{1}_{\mu_j>0} = \lambda_j - H + d - \lambda_j - (d - H) = 0$, as required. ◀

Let $T = \{j : \mu_j > 0\}$. Since $\sum_j \lambda_j C_j \doteq \sum_i \Big(\sum_j \lambda_j a_{ij}\Big) x_i \geq \sum_j \lambda_j A_j \doteq \sum_i b'_i x_i \geq \sum_j \lambda_j A_j$ and similarly $-\sum_j \mu_j C_j \doteq \sum_i \Big(\sum_j (-\mu_j) a_{ij}\Big) x_i \geq \sum_j (-\mu_j) A_j \doteq \sum_i f_i x_i \geq F$, we can write the constraint $\sum_j (\lambda_j - \mu_j - \delta \mathbf{1}_{\mu_j>0}) C_j$ as

$$\sum\nolimits_i (b'_i + f_i) x_i - \delta \sum\nolimits_{j \in T} \sum\nolimits_i a_{ij} x_i \geq \sum\nolimits_j \lambda_j A_j + F - \delta \sum\nolimits_{j \in T} A_j. \tag{71}$$

We partially weaken this on variables in P by adding $\sum_{i \in P} (b'_i \bmod d) \overline{x}_i \geq 0$, to obtain

$$\sum\nolimits_i (b_i + f_i) x_i - \delta \sum\nolimits_{j \in T} \sum\nolimits_i a_{ij} x_i \geq B + F - \delta \sum\nolimits_{j \in T} A_j. \tag{72}$$

Hence, adding this to Equation (70) yields

$$\sum\nolimits_i \left(b_i - M(b_i, d, H)\right) x_i - \delta \sum\nolimits_{j \in T} \sum\nolimits_i a_{ij} x_i \geq B - \lceil B/d \rceil H + 1 - \delta \sum\nolimits_{j \in T} A_j. \tag{73}$$

To analyze this, we write $b_i = g_i d + h_i$ with $0 \leq h_i \leq d - 1$. Note that $M(g_i d + h_i, d, H) = g_i H + \min(h_i, H)$. Also, we have $B - \lceil B/d \rceil H = Gd + H - (G + 1)H = G(d - H) = G\delta$. Hence, we can write Equation (73) as

$$\sum\nolimits_i \big(g_i \delta + h_i - \min(h_i, H)\big) x_i - \delta \sum\nolimits_{j \in T} \sum\nolimits_i a_{ij} x_i \geq G\delta + 1 - \delta \sum\nolimits_{j \in T} A_j. \tag{74}$$

We now apply the division rule with divisor $\delta = d - H$ (which is possible since $H < d$). To compute the result, write $S = \{i : h_i > H\}$. If $i \in S$, then

$$g_i \delta < g_i \delta + h_i - \min(h_i, H) < (g_i + 1)\delta,$$

since $h_i < d$. Hence, $\left\lceil \frac{g_i \delta + h_i - \min(h_i, H)}{\delta} \right\rceil = g_i + 1$. On the other hand, if $i \notin S$, then $h_i - \min(h_i, H) = 0$ so $\left\lceil \frac{g_i \delta + h_i - \min(h_i, H)}{\delta} \right\rceil = g_i$. Hence, the result is

$$\sum\nolimits_i g_i x_i + \sum\nolimits_{i \in S} x_i - \sum\nolimits_{j \in T} \sum\nolimits_i a_{ij} x_i \geq G + 1 - \sum\nolimits_{j \in T} A_j. \tag{75}$$

Here, we use that all coefficients in $-\delta \sum_{j \in T} \sum_i a_{ij} x_i$ are divisible by δ, and hence do not affect the division of the x_i, even though the variables overlap. However, this division potentially not in normalized form. Nevertheless, any division which is not in normalized form can be simulated in *VeriPB* by applying partial weakening before dividing.

We now proceed by showing how we can combine Equation (70) with Equation (75) to derive a contradiction. In the newly introduced notation, we can write Equation (70) as

$$-\sum_i \left(g_i H + \min(h_i, H)\right) x_i - \sum_i f_i x_i \geq -(G+1)H - F + 1, \tag{76}$$

where we have also rewritten the terms involving the f_i in variable form.

Multiplying Equation (75) with H and adding it to Equation (76) yields

$$-\sum_{i \notin S} h_i x_i + \sum_i \left(-f_i - H \sum_{j \in T} a_{ij}\right) x_i \geq -F - H \sum_{j \in T} A_j + 1. \tag{77}$$

Note that we can write $f_i = -\sum_j a_{ij}\mu_j$ and $F = -\sum_j A_j \mu_j$, which yields

$$-\sum_{i \notin S} h_i x_i + \sum_i \sum_{j \in T} (\mu_j - H) a_{ij} x_i \geq \sum_{j \in T} (\mu_j - H) A_j + 1. \tag{78}$$

Since $\mu_j \leq H$, we have $H - \mu_j \geq 0$. Hence, we can add C_j with multiplier $H - \mu_j \geq 0$ for all j with $\mu_j > 0$ to Equation (78), which yields $-\sum_{i \notin S} h_i x_i \geq 1$. Since $h_i \geq 0$, this yields a contradiction by adding literal axioms $x_i \geq 0$.

B Core-Guided PB Logging

In this section use X_S as a shortcut to denote $\sum_S x_i$. We do not normalise constraints. We assume without loss of generality that all weights and literals in the objective are positive, so we can write $f_o \doteq \sum w_i x_i$.

During optimization we repeatedly call the solver with assumptions. Learned constraints are logged in the standard way. The solver might return in either of two states. If the solver found a solution, we log it using the `soli` rule and obtain a solution-improving constraint $f_o \leq u - 1$. Otherwise the solver finds the assumptions are contradictory. We process the contradiction into a cardinality constraint $\sum_{i \in S} x_i \geq d$, or $X_S \geq d$ for short, which we log and call the "core lower bound" constraint. We use this to reformulate the objective.

▶ **Lemma 2.** *The core lower bound is derivable with standard conflict analysis.*

To do this we introduce counting variables y_j for $j \in [0, |S|]$ with intended semantics $y_j = [\![X_S \geq j]\!]$. Then we can reformulate the objective to $f_r \doteq f_o - w\Delta_K$ where $\Delta_K \doteq X_S - Y$. Observe that $\Delta_K = 0$, hence the objective does not change. We call a triple (S, d, w) a core, and denote the set of all cores by $\mathcal{K}$.

Taking all cores into account, we have $\Delta \doteq \sum_{K \in \mathcal{K}} \Delta_K = 0$, which we can use to log objective lower bounds $f_o \geq \ell$ and to translate solution-improving constraints $f_o \leq u$ for the original objective into solution-improving constraints $f_r \leq u$ for the reformulated objective. Observe that we only use the fact that the reformulated objective lower bounds the original objective, in other words $f_r \leq f_o$ or $\Delta \geq 0$. We call such constraint the "global reformulation constraint", and we discuss how to derive it later.

In principle we might have $|S| + 1$ counting variables, but fewer variables are enough. Since $X_S \geq d$, we already know that $y_i = 1$ for $i \leq d$. Similarly, if we have an upper bound on the core of the form $u \geq X_S$, then we also know that $Y_{>u} = 0$, i.e., $y_i = 0$ for $i > u$.

We always have a trivial upper bound of the form $|S| \geq X_S$, but sometimes we can obtain a better bound from the solution-improving constraint. Let the "objective-to-core" constraint be $f_o - wX_S \geq 0$. If we have $f_o \leq u$, we have $X_S \leq \lfloor u/w \rfloor$. If $\lfloor u/w \rfloor < |S|$ then we log this constraint as the "core upper bound" constraint.

▶ **Lemma 3.** *The objective-to-core constraint is derivable from the global reformulation constraint.*

Proof. We have $f_r - wX_S \geq 0$ trivially. We add $\Delta \geq 0$ and obtain $f_o - wX_S \geq 0$. ◀

It is worth mentioning that, as an optimization, we delay logging this constraint until the time it is needed, if at all. This means that the reformulated objective may have changed in the meantime, and the actual way in which we log the objective-to-core constraint is by marking it as syntactically implied from $\Delta \geq 0$.

▶ **Lemma 4.** *The core upper bound is derivable from the solution-improving constraint and the objective-to-core constraint.*

Proof. We add $f_o \leq u$ and $f_o - wX_S \geq 0$ and obtain $wX_S \leq u$. Then we divide by w. ◀

Another use of the global reformulation constraint is hardening. Given the objective improving constraint $f_r \leq u$, any literal appearing in f_r with coefficient larger than u can be fixed to 0. We log such constraints by RUP.

How to proceed depends on how we encode $y_j = [\![X_S \geq j]\!]$.

B.1 Eager Encoding

With an eager encoding we reformulate the objective to $f_r \doteq f_o - w\Delta_K$ where $\Delta_K \doteq X_S - Y \doteq X_S - Y_{[d+1,u]} - d$. We need to derive an upper and a lower bound on Δ_K.

▶ **Lemma 5.** *The at-most constraint $\Delta_K \leq 0$ is derivable from the core upper bound.*

Proof. We derive $Y_{[d+1,u]} + d \geq X_S$ by redundance with witness $Y_{[d+1,u]} \mapsto 1$. Previously derived constraints are syntactically untouched by the witness and do not need proving. The conclusion becomes $u \geq X_S$, which is syntactically identical to the core upper bound. ◀

▶ **Lemma 6.** *The at-least constraint $\Delta_K \geq 0$ is derivable from the core lower bound.*

Proof. For $j = d+1, \ldots, u$, we derive $X_S \geq Y_{[d+1,j]} + d$ by redundance with witness $y_j \mapsto 0$. Previous constraints are untouched. The at-most constraint becomes $Y_{[d+i,u]\setminus\{j\}} + d \geq X_S$, which is a weakening of the negated conclusion $Y_{[d+1,j]} + d - 1 \geq X_S$. Intermediate at-least constraints are untouched. The conclusion becomes $X_S \geq Y_{[d+1,j)} + d$, which is either syntactically identical to the core lower bound if $j = d+1$, or to the previous intermediate at-least constraint otherwise.

We delete all intermediate at-least constraints at the end. ◀

Note that we do not use a single application of the redundance rule with witness $Y_{[d+1,u]} \mapsto 0$ because the negation of the at-least constraint does not necessarily imply the at-most constraint after applying the witness substitution.

▶ **Lemma 7.** *The ordering constraints $y_j \geq y_{j+1}$ are derivable.*

Proof. For $j = d+1, \ldots, u-1$ we derive $y_j \geq y_{j+1}$ by redundance with witness $y_{j+1} \mapsto y_j \mapsto y_{j-1} \mapsto \cdots \mapsto y_{d+1} \mapsto y_{j+1}$. Its negation propagates $y_j = 0$ and $y_{j+1} = 1$. The at-least and at-most constraints are mapped to themselves. $y_{d+1} \geq y_{d+2}$ is mapped to $y_{j+1} \geq y_{d+1}$, which is satisfied by unit propagation. Other previous ordering constraints are mapped to (different) previous ordering constraints. The conclusion is also mapped to a previous ordering constraint. ◀

B.2 Reified Encoding

Since introducing many variables at once may slow down the solver, an alternative to an eager encoding is to use a lazy encoding, where we introduce counting variables one by one on demand.

That is, instead of Δ_K, we subtract $\Delta_K^{(j)} \doteq X_S - Y_{[d+1,d+j]} - d$ from the objective. Observe that $\Delta_K^{(j)} \geq 0$, hence the reformulated objective still lower bounds the original objective, even though they might no longer be equal.

We begin by introducing a new variable y_{d+1}, and two constraints $y_{d+1} \to [\![X_S \geq d+1]\!]$ and $[\![X_S \geq d+1]\!] \to y_{d+1}$. As linear constraints, these correspond to $X_S \geq y_{d+1} + d$ ("at least") and $X_S \leq (u-d)y_{d+1} + d$ ("at most") respectively.

If we remove the counting variable from the objective during a subsequent objective reformulation, then the core would become useless and we would not be able to prove an exact lower bound on the objective. Therefore, in that case, we introduce a new counting variable and the relevant constraints, namely $X_S \geq jy_{d+j} + d$ and $X_S \leq (u-d-j+1)y_{d+j} + d + j - 1$. We write $\xi = (u - d - j + 1)$.

▶ **Lemma 8.** *The at-most constraint is derivable from the core upper bound.*

Proof. We derive $\xi y_{d+j} - X_S \geq -d - j + 1$ by redundance with witness $y_{d+j} \mapsto 1$. Previous constraints are untouched. The conclusion becomes $u - X_S \geq 0$, which is trivial for $u = |S|$ and syntactially identical to the core upper bound otherwise. ◀

▶ **Lemma 9.** *The at-least constraint is derivable from the core lower bound.*

Proof. We derive $X_S - jy_{d+j} \geq d$ by redundance with witness $y_{d+j} \mapsto 0$. Previous constraints are untouched. The at-most constraint becomes $-X_S \geq -d - j + 1$, which is a weakening of the negated conclusion $-X_S + jy_{d+1} \geq -d + 1$. The conclusion becomes $X_S \geq d$, which is syntactically identical to the core lower bound. ◀

▶ **Lemma 10.** *The ordering constraint $y_j \geq y_{j+1}$ is derivable from the at-least and previous at-most constraints.*

Proof. Add the at-least and the previous at-most constraints. Then divide by a large enough number such as $|S|$. ◀

The local reformulation constraint changes each time that we introduce a variable. The first constraint is $\Delta_K^{(1)} \doteq X_S - y_{d+1} - d \geq 0$, which is syntactically identical to the at-least constraint. Subsequent reformulation constraints need to be derived.

▶ **Lemma 11.** *The local reformulation constraint $\Delta_K^{(j)}$ is derivable from the at-most constraint and $\Delta_K^{(j-1)}$.*

Proof. Take a linear combination of $(j-1)\Delta_K^{(j-1)}$ and the at-most constraint. Then divide by j. ◀

Observe that on a lazy encoding, bounds on the core need not stay constant. If we find better solutions, we may end up with better upper bounds. In that case we need to rederive the core upper bound. Note that the objective might have changed in the meantime, but the objective-to-core constraint already takes that into account by using the reformulation constraint from the time the core was found.

In general, a better core upper bound will be taken automatically into account when a new variable is introduced. However, it may be that the new upper bound makes introducing new variables unnecessary. In that case we have $d + j = u$, and we may improve the last at-most constraint to $X_S \leq y_{d+j} + d + j - 1 = y_{d+j} + u - 1$ instead.

▶ **Lemma 12.** *The improved at-most constraint is derivable from the core upper bound and the at-most constraint.*

Proof. Take a linear combination of $(\xi - 1)(X_S \leq u)$ and the at-most constraint. Then divide by ξ. ◀

B.3 Lazy Sum Encoding

The lazy sum encoding follows the same principles as the reified encoding, but we use stronger at-most and at-least constraints for subsequent counter variables. Instead of $X_S \geq jy_{d+j} + d$ we have $X_S \geq Y_{[d+1,d+j]} + d$, and instead of $X_S \leq (u - d - j + 1)y_{d+j} + d + j - 1$ we have $X_S \leq (u - d - j + 1)y_{d+j} + Y_{[d+1,d+j-1]} + d$. Observe that we can derive the reified at-least constraint as a linear combination of its lazy sum counterpart and ordering constraints, and that the reified at-most constraint is a weakening of the lazy-sum at-most constraint.

The first at-least and at-most constraints are derived identically to the reified case. Subsequent constraints are derived as follows.

▶ **Lemma 13.** *The at-most constraint is derivable from the previous at-most constraint.*

Proof. We derive $\xi y_{d+j} + Y_{[d+1,d+j-1]} - X_S \geq -d$ by redundance with witness $y_{d+j} \mapsto y_{d+j-1}$. Previous constraints are untouched. The conclusion becomes syntactically identical to the previous at-most constraint. ◀

▶ **Lemma 14.** *The at-least constraint is derivable from the previous at-least constraint.*

Proof. We derive $X_S - Y_{d+1,d+j} \geq d$ by redundance with witness $y_{d+j} \mapsto 0$. Previous constraints are untouched. The at-most constraint becomes $Y_{[d+1,d+j-1]} - X_S \geq -d$, which is a weakening of the negated conclusion $-X_S + Y_{d+1,d+j} \geq -d + 1$. The conclusion becomes syntactically identical to the previous at-least constraint. ◀

The ordering constraint $y_j \geq y_{j+1}$ is derivable from the at-least and previous at-most constraints identically to the reified case. The local reformulation constraint is always identical to the at-least constraint and does not need to be derived.

We need to handle updated core upper bounds in a different way since logging the at-most constraint depends on the previous at-most constraint, which is based on a different core upper bound. What we do instead is to first derive an improvement of the previous at-most constraint, and then log the current at-most constraint as usual.

▶ **Lemma 15.** *The improved at-most constraint is derivable from the core upper bound, the ordering constraints, and the at-most constraint.*

Proof. Let d be the core lower bound. Let $u = \xi + d$ be the previous core upper bound (so ξ is the previous expected number of counting variables). Let $u' = \eta + d$ be the new core upper bound (so η is the new expected number of counting variables).

We derive $X_S \leq (\eta - j + 1)y_{d+j} + Y_{[d+1,d+j-1]} + d$ by redundance with an empty witness. We add the current at-most constraint and the negation of the conclusion to obtain $(u - u')y_{d+j} \geq 1$, which we saturate to $y_{d+j} \geq 1$. The ordering constraints then propagate $y_{d+1} = \cdots = y_{d+j} = 1$. We take a linear combination of these and $X_S \leq u$ to obtain the conclusion. ◀

Learning to Bound for Maximum Common Subgraph Algorithms

Buddhi W. Kothalawala
The Australian National University, Canberra, Australia

Henning Koehler
Massey University, Palmerston North, New Zealand

Qing Wang
The Australian National University, Canberra, Australia

Abstract

Identifying the maximum common subgraph between two graphs is a computationally challenging NP-hard problem. While the McSplit algorithm represents a state-of-the-art approach within a branch-and-bound (BnB) framework, several extensions have been proposed to enhance its vertex pair selection strategy, often utilizing reinforcement learning techniques. Nonetheless, the quality of the upper bound remains a critical factor in accelerating the search process by effectively pruning unpromising branches. This research introduces a novel, more restrictive upper bound derived from a detailed analysis of the McSplit algorithm's generated partitions. To enhance the effectiveness of this bound, we propose a reinforcement learning approach that strategically directs computational effort towards the most promising regions within the search space.

2012 ACM Subject Classification Theory of computation → Branch-and-bound; Mathematics of computing → Combinatorial optimization

Keywords and phrases Combinatorial Search, Branch and Bound, Graph Theory

Digital Object Identifier 10.4230/LIPIcs.CP.2025.22

Supplementary Material *Software (Source Code)*: `https://github.com/BuddhiWathsala/mcsplit-dsb`, archived at `swh:1:dir:7bbe80724d5f242e89478b8ab3956ac6a35d2daa`

Funding This research was supported partially by the Australian Government through the Australian Research Council's Discovery Projects funding scheme (project DP210102273).

Acknowledgements We are thankful to Dr Muhammad Farhan for helpful discussions and feedback.

1 Introduction

Graphs are widely employed in various real-world applications due to their effectiveness in modeling complex structures. A significant challenge in these applications is identifying common patterns between graphs. Given two graphs, G and H, the task of finding two induced subgraphs from G and H that are isomorphic and contain the maximum number of vertices is known as the *Maximum Common Induced Subgraph (MCIS)* problem [7]. For simplicity, the MCIS problem is also referred to as the maximum common subgraph problem. The MCIS problem is NP-hard [2], making it computationally challenging. Despite its complexity, it has broad applications across multiple fields. In chemical analysis, it helps to group chemical compounds based on their structural features [6, 9]. In bioinformatics, it facilitates the comparison of molecular structures, supporting drug discovery and genomic analysis [8]. MCIS is also crucial in graph similarity measures, which are commonly used for anomaly detection in networks [1], as well as in pattern recognition tasks like video analysis and image processing [20].

© Buddhi W. Kothalawala, Henning Koehler, and Qing Wang;
licensed under Creative Commons License CC-BY 4.0

Several exact [16] and approximate algorithms [19] have been developed to solve the MCIS problem. The exact algorithms employ various techniques such as constraint programming [25, 18], maximum clique algorithms [11, 10, 4], and the branch-and-bound (BnB) approach [16, 24, 17]. Among these, the branch-and-bound algorithm called McSplit, introduced by McCreesh et al. [16], has shown exceptional performance. The McSplit algorithm effectively tackles the MCIS problem using two essential heuristics: a branching heuristic that selects optimal vertex pairs for matching and a bounding heuristic that estimates the maximum possible size of the common subgraph within a given search branch. For branching, McSplit prioritizes vertices with higher degrees when choosing vertex pairs, while its bounding heuristic calculates an upper bound by counting the number of potential matching vertex pairs.

Several machine learning-based improvements have been introduced to enhance McSplit's branching heuristic using reinforcement learning (RL) techniques, all while maintaining its exactness. Prominent examples include McSplit+RL [12], McSplit+LL [27], GLSearch [3], and McSplit+DAL [13]. McSplit+RL selects vertex pairs with a high potential to significantly reduce the bound, using bound reduction as the RL reward. McSplit+LL extends this by incorporating both long-term and short-term rewards in selecting the next vertex pair and enables the simultaneous matching of leaf vertices to improve search efficiency. McSplit+DAL further enhances McSplit+RL's reward mechanism by prioritizing vertex pairs that simplify the MCIS problem. It also diversifies vertex pair selection using a hybrid branching policy that alternates between McSplit+RL and McSplit+DAL. In contrast, GLSearch leverages a deep Q-network to identify the next vertex pair, focusing on maximizing the size of the common subgraph rather than simply ending the search quickly, as done in McSplit+RL, McSplit+LL, and McSplit+DAL.

While previous research on enhancing the McSplit algorithm in reinforcement learning has largely focused on optimizing branching strategies, this paper introduces a new perspective on improving upper bound estimation. We observed that analyzing the internal structure of matching pairs can provide a tighter upper bound, allowing for earlier pruning of search branches compared to the current bound calculation. The key contributions of this paper are as follows:

1. **A Tighter Upper Bound for MCIS:** We establish a tighter upper bound for the MCIS problem by utilizing structural graph properties, such as edge cardinality and degree sequence, to improve pruning efficiency. Since the new bound is theoretically valid, it preserves the exactness of the MCIS search.
2. **Reinforcement Learning to Applying the Bound:** We introduce a RL framework that dynamically prioritizes bound computation in high-potential search regions, reducing redundant calculations while maintaining solution quality.
3. **Empirical Performance Gains:** We empirically demonstrate that improvements after incorporating the proposed bound improves McSplit and its extensions in solving the MCIS problem.
4. **Achieving Near-Maximal Bound Reduction:** We empirically show that our bound closely approximates the theoretically maximum achievable bound reduction in McSplit, validating its near-optimal pruning effectiveness.

We introduce a novel algorithm, McSplit+DSB (Degree Sequence Bound), which replaces the original bound calculation of the McSplit algorithm proposed by McCreesh et al. [16]. The proposed bound calculation is also adaptable to other RL-based McSplit extensions. Building on the reinforcement learning extensions McSplit+RL, McSplit+LL, and McSplit+DAL, we present McSplit+RL+DSB, McSplit+LL+DSB, and McSplit+DAL+DSB, which incorporate

Table 1 Comparison of branching and bounding heuristics across different McSplit variants. The table highlights the selection strategies for bidomains, vertices, and matches in various methods, as well as the proposed modification to the bounding heuristic in our approach (McSplit+DSB), which incorporates an additional reduction term δ to enhance pruning efficiency.

Method	Branching Heuristics			Bounding Heuristics
	Bidomain Selection	Vertex Selection	Match Selection	
McSplit	$max(\|V_l\|, \|U_l\|) \downarrow$	$d_G(v) \downarrow$	$d_H(u) \downarrow$	$\sum_{\langle V_l, U_l\rangle \in P_{GH}} \min(\|V_l\|, \|U_l\|)$
McSplit+RL	no change	$S_G(v) \downarrow$	$S_H(u) \downarrow$	no change
McSplit+LL	no change	$S_G(v)$ with decay $\downarrow$	$S_{GH}(v,u)$ with decay $\downarrow$	no change
McSplit+DAL	no change	$S'_G(v)$ with decay $\downarrow$	$S'_{GH}(v,u)$ with decay $\downarrow$	no change
McSplit+DSB (Ours)	no change	no change	no change	$\sum_{\langle V_l, U_l\rangle \in P_{GH}} \min(\|V_l\|, \|U_l\|) - \delta$

our new bound calculation into these existing algorithms. To assess the effectiveness of these approaches, we conducted extensive experiments on benchmark datasets for the MCIS problem.

The paper is structured as follows: Section 2 introduces the notations and definitions. Section 3 describes the McSplit algorithm. Section 4 details our degree sequence bound calculation method and the reinforcement learning approach designed to integrate it into the search process. Section 5 presents the experimental evaluation, where the proposed method is compared with state-of-the-art techniques. Finally, Section 6 provides concluding remarks.

2 Preliminaries

Let G be a simple and unweighted graph, where $V(G)$ is the set of vertices and $E(G)$ is the set of edges. For a vertex $v \in V(G)$, the *neighborhood* of v is defined as $N(G, v) = \{u \in V(G) \mid (u, v) \in E(G)\}$. Given a subset of vertices $V' \subseteq V(G)$, we use $G[V']$ to denote the *induced subgraph* of G, which contains all vertices in V' and edges in $\{(u, v) \in E(G) \mid \{u, v\} \subseteq V'\}$. To simplify, we use $G[V'] \subseteq G$ to indicate that $G[V']$ is an induced subgraph of G. Let G be a graph with $V(G) = n$ and $d_G(v)$ denote the degree of a vertex v in G. Then, the degree sequence of G is a non-decreasing sequence of the degrees of all its vertices, i.e. $Deg(G) = (d_G(v_1), \ldots, d_G(v_n))$, and the degree sum of G is defined as $d(G) = \sum_{i=1}^{n} d_G(v_i)$.

Two graphs G and H are *isomorphic*, denoted as $G \cong H$, if there exists a bijection $f : V(G) \to V(H)$ such that $(v_1, v_2) \in E(G) \Leftrightarrow (f(v_1), f(v_2)) \in E(H)$. Two induced subgraphs $G[V']$ and $H[U']$ are called *common induced subgraphs* of G and H iff $G[V'] \cong H[U']$. We use $\bar{G}$ to denote the *complement graph* of a graph G. If $G \cong H$, then $\bar{G} \cong \bar{H}$, and vice versa.

▶ **Definition 1.** *Given two graphs G and H, the* maximum common induced subgraph *(MCIS) problem is to find two induced subgraphs $G[V'] \subseteq G$ and $H[U'] \subseteq H$ such that $G[V'] \cong H[U']$ and there exist no common induced subgraphs of G and H that have a size larger than $|V'|$.*

Without loss of generality, throughout the paper, we consider two input graphs G and H where $|V(G)| \leqslant |V(H)|$, with $|V(G)| = n$ and $|V(H)| = m$.

3 Branch and Bound for MCIS

The branch-and-bound algorithm demonstrates significantly better performance than other algorithms for the MCIS problem. It typically employs a depth-first search, starting with an empty subgraph and adding matching vertex pairs, while partitioning the remaining

vertices based on their connectivity to the selected ones. In the BnB approach to MCIS, the branching strategy is based on selecting the next matching vertex, and the bound is calculated by estimating the number of matching pairs that can still be added to the current common subgraph.

3.1 McSplit: Partitioning-Based Algorithm

McCreesh et al. [16] propose a BnB algorithm to solve the MCIS problem, called McSplit. Given two graphs G and H, McSplit solves the MCIS problem by finding a maximum-cardinality mapping $M^* = \{(v_1, u_1), \ldots, (v_m, u_m)\}$, where all $v_i \in V(G)$ and $u_i \in V(H)$ are distinct from one another and satisfy the condition $(v_i, v_j) \in E(G) \Leftrightarrow (u_i, u_j) \in E(H)$ for any v_i, v_j with $1 \leqslant i \neq j \leqslant m$. Each vertex pair $(v_i, u_i) \in M^*$ is called a *matching vertex pair*. Such a maximum-cardinality mapping corresponds to the maximum common subgraphs $G[V^*]$ and $H[U^*]$ of G and H, where $V^* = \{v_1, \ldots, v_m\}$ and $U^* = \{u_1, \ldots, u_m\}$.

To find a maximum-cardinality mapping between G and H, McSplit employs a depth-first search, starting with the empty mapping. It then iteratively extends the mapping with matching vertex pairs, adding one pair at each iteration until no further extensions are possible. The key idea of finding matching vertex pairs is based on bidomain partitioning. This process uses a labelling function that assigns labels to unmatched vertices based on their connectivity with matched vertices.

▶ **Definition 2** (Labelling). *Let $M = \{(v_1, u_1), \ldots, (v_k, u_k)\}$ be a (not necessarily maximum-cardinality) mapping between two graphs G and H, with $V' = \{v_1, \ldots, v_k\}$ and $U' = \{u_1, \ldots, u_k\}$. A labeling function $\ell : V(G) \to \{0,1\}^k$ assigns a label to each vertex $w \in V(G)\backslash V' \cup V(H)\backslash U'$ such that the i^{th} bit of $\ell(w)$ is equal to 1 if $(w, v_i) \in E(G)$ or $(w, u_i) \in E(H)$; otherwise, it is 0.*

Based on the labels, McSplit partitions unmatched vertices in $V(G)\backslash V' \cup V(H)\backslash U'$ into a set of bidomains $P_{GH} = \{\langle V_1, U_1\rangle, \ldots, \langle V_t, U_t\rangle\}$ such that $\forall v \in V_i$ and $\forall u \in U_j$, $i = j \Leftrightarrow \ell(v) = \ell(u)$, ensuring that all vertices within a single partition share the same label, while vertices in different partitions have distinct labels.

Specifically, the McSplit algorithm starts by initializing $P_{GH} = \{\langle V(G), V(H)\rangle\}$ and picks a matching bidomain pair $\langle V_l, U_l\rangle$. Then, the algorithm selects a vertex $v \in V_l$ and v is matched against all the vertices $u \in U_l$. Matching v with any u gives $M = \{(v, u)\}$. Then, unmatched vertices in G and H are labelled based on the connectivity of v and u, respectively. This leads to bidomain re-partitioning such that each bidomain $\langle V_l, U_l\rangle \in P_{GH}$ is partitioned into two bidomains $\langle V_l \cap N(G, v), U_l \cap N(H, u)\rangle$ and $\langle V_l\backslash N(G, v), U_l\backslash N(H, u)\rangle$. Iteratively, the algorithm selects another vertex pair to expands M. Once v is matched against all the vertices $u \in U_l$, the algorithm removes the vertex v from V_l and continues with the search. The removal of v is denoted as matching v with the special symbol $\perp$. The algorithm continues until there are no more matching vertex pairs to select. In every iteration, alongside the current mapping M being expanded, the algorithm keeps track of the best solution discovered so far, referred to as the *incumbent*. It then uses the following upper bound to eliminate unpromising search branches.

$$UB \leftarrow |M| + \sum_{\langle V_l, U_l\rangle \in P_{GH}} \min(|V_l|, |U_l|) \tag{1}$$

If the current mapping M cannot be expanded more than *incumbent* in size, i.e., $UB \leqslant |incumbent|$, the algorithm immediately terminate the search, otherwise continues.

3.2 McSplit + Reinforcement Learning

Recent studies extend McSplit using reinforcement learning techniques [12, 27, 13], where an agent explores an environment by taking actions [23]. All these studies consider the McSplit algorithm as an agent whose goal is to reach a search tree leaf as early as possible to reduce the overall size of the search tree. The choice of a matching vertex pair (v, w) at each branching step is modelled as an action. This viewpoint naturally leads to using the bound reduction as a crucial factor for defining the reward of the action (v, w), and then incorporating accumulated rewards into heuristics for selecting matching vertex pairs.

3.2.1 McSplit+RL

McSplit+RL [12] extends McSplit by selecting a matching vertex pair (v, u) using a learning-based method. The main idea behind McSplit+RL is to define the reward for an action (v, u) based on how much *bound* is reduced after selecting (v, u) as the matching vertex pair. The value function maintains two score lists $S_G(v)$ and $S_H(u)$ for all $v \in V_l$ and $u \in U_l$ during the search. Then, at each branch step, a bidomain $\langle V_l, U_l \rangle$ is selected in the same way as McSplit. A vertex $v \in V_l$ with the highest score $S_G(v)$ is chosen and matched with a vertex $u \in U_l$ in decreasing order of $S_H(u)$. These together determine the choice of a matching vertex pair (v, u) in McSplit+RL.

3.2.2 McSplit+LL

McSplit+LL [27] further extends McSplit+RL with two additional operators: 1) long-short memory, and 2) leaf vertex union match. Essentially, McSplit+LL defines the reward and maintains the score list $S_G(v)$ in the same way as McSplit+RL. However, it uses a score list $S_{GH}(v, u)$ for matching vertex pairs, which is also arranged in decreasing order. Further, the scores of $S_G(v)$ and $S_{GH}(v, u)$ are decayed to a half when a matching vertex pair reaches their thresholds, respectively. When a matching vertex pair (v, w) is chosen, McSplit+LL allows to match as many unmatched leaf vertex pairs as possible simultaneously to expand the mapping M.

3.2.3 McSplit+DAL

McSplit+DAL [13] attempts to address two limitations of McSplit+RL: 1) bidomain simplification relating to different matches are not considered, and 2) vertices with high accumulated rewards are repeatedly chosen during backtracking. It defines a new reward for an action (v, u) by adding $|P_{GH}|$ to the original reward of McSplit+RL, which encourages the reward is given to more simplified states. Based on the new reward, two score lists $S'_G(v)$ and $S'_{GH}(v, u)$ are defined in a similar way as $S_G(v)$ and $S_{GH}(v, u)$ in McSplit+LL. A hybrid strategy that combines its proposed heuristic with the heuristic of McSplit+RL is also introduced to diversify the search space.

3.3 Limitations of Existing Algorithms

Within the branch and bound framework, search heuristics used by algorithms for solving the MCIS problem can be generally categorized into two types:

- *Branching heuristic*: This typically involves *partition selection* (selecting a bidomain $\langle V_l, U_l \rangle$ from P_{GH}), *node selection* (selecting a vertex v from V_l), and *match selection* (selecting a vertex u from U_l to form a matching vertex pair (v, u)).

- *Bounding heuristic*: This estimates the maximum number of vertices (i.e., upper bound) in the maximum common subgraph. If this estimated upper bound is less than the found common subgraph, it prunes that search branch.

Currently, all existing learning-based methods for McSplit focus on improving branching heuristics, which essentially involves deciding the "best" matching vertex pair to branch out, assuming that choosing different matching vertex pairs may lead to different bound reductions and thus reach the pruning condition at different speeds. They all use the bounding heuristic of McSplit that estimates the upper bound based on $\sum_{\langle V_l, U_l \rangle \in P_{GH}} \min(|V_l|, |U_l|)$ (see Equation (1)).

Observation. The bounding heuristic of McSplit is too generous. For instance, if $G[V_l]$ is a clique of five vertices (K_5) and $H[U_l]$ is the complement graph ($\bar{K_5}$), the size of their maximum common subgraphs is one, which is significantly smaller than $\min(|V_l|, |U_l|) = 5$ estimated in the bound.

This observation motivates us to develop a learning-based bounding heuristic that can adaptively estimate the bound by considering the underlying graph structure. Table 1 summarizes branching and bounding heuristics used in McSplit and the existing learning-based extensions.

4 Proposed Method

We propose a novel bounding heuristic with two synergistic components: (1) a tight upper bound calculation leveraging degree sequences and partition degree sums, and (2) a reinforcement learning framework that strategically deploys this bound in high-impact regions of the BnB search space. Before introducing our heuristic, we analyze existing approaches to motivate the research problem and discuss the unique challenges in developing learning-based bounding heuristics.

Section 4.1 analyzes the key challenges in developing a novel bound calculation method. Section 4.2 presents the theoretical foundation of our proposed bound, supported by practical examples. Finally, Section 4.3 introduces the reinforcement learning (RL) framework designed to effectively integrate this new bounding strategy.

4.1 Challenges

We consider the McSplit algorithm or any of its extensions McSplit+RL, McSplit+LL, and McSplit+DAL as an agent. Instead of reaching a search tree leaf as early as possible, the goal of our agent is to prune branches as early as possible. Thus, unlike existing methods where an action is to choose the best matching vertex pair, an action in our method is to decide whether to employ a new and tighter upper bound to prune a branch earlier. However, there are two major challenges in designing a learning-based bounding heuristic in this setting: (1) How to design an efficient algorithm that can provide a tighter upper bound than the original bound of McSplit? (2) How to design the reward and value function that can accurately predict when a pruning condition can be met under new and tighter bounds? Addressing these two challenges are the key to enhancing the overall performance of the algorithm. We discuss our approach below in detail.

4.2 Degree Sequence Bound

We begin by proposing a novel upper bound for MCIS, which is tighter than the bound of McSplit and can be used to reduce the search space for BnB algorithms based on McSplit. The intuition behind this novel upper bound is simple. Given a bidomain $\langle V_l, U_l \rangle$, we derive an upper bound for maximum common subgraphs between $G[V_l]$ and $H[U_l]$ based on their degree sequences, not merely the sizes of $|V_l|$ and $|U_l|$ as used by $\min(|V_l|, |U_l|)$ in McSplit. This is because the degree sequence of a graph can provide more useful structural information than computing the size of the small graph.

Let $M^*(G, H)$ denote a maximum-cardinality mapping between two graphs, G and H. An *independent set* is a set of pairwise non-adjacent vertices. Based on degree sequences, we derive the following (recall that degree sequences are monotonically increasing).

▶ **Lemma 3.** *For a graph G with degree sequence $Deg(G) = (d_G(v_1), \ldots, d_G(v_k), \ldots, d_G(v_n))$, the degree sum of any induced subgraph on k vertices is bounded from below by*

$$\sum_{i=1}^{k} d_G(v_i) - \sum_{j=k+1}^{n} d_G(v_i)$$

If $v_{k+1}, \ldots, v_n$ form an independent set, then this bound is tight.

Proof. Any induced subgraph $G[V_k]$ on k can be obtained by removing $n-k$ distinct vertices from G in some arbitrary order. Removing a vertex v_i reduces the degree-sum by precisely twice the degree of v_i in the current subgraph. The degree of v_i in any subgraph of G is upper-bounded by $d_G(v_i)$, so the degree-sum of the $n-k$ vertices removed to obtain $G[V_k]$ is upper-bounded by $\sum_{j=k+1}^{n} d_G(v_i)$. This shows the lower bound claim.

If $v_{k+1}, \ldots, v_n$ form an independent set, then removing some of them from G will not affect the degrees of others. Thus the bound is tight for $G[V_k] = G[v_1, \ldots, v_k]$. ◀

▶ **Theorem 4.** *Let $|V(G)| \leqslant |V(H)|$ and $Deg(G) = (d_G(v_1), \ldots, d_G(v_n))$. If*

(1) $\sum_{i=1}^{k} d_G(v_i) - \sum_{j=k+1}^{n} d_G(v_j) > d(H)$, *or*

(2) $\sum_{j=n-k-1}^{n} d_{\bar{G}}(v_j) - \sum_{i=1}^{n-k} d_{\bar{G}}(v_i) > d(\bar{H})$

then $|M^*(G, H)| \leqslant k-1$, where $d_{\bar{G}}(v_i) = |V(G)| - d_G(v_i) - 1$ for $i = 1, \ldots, n$.

Proof. From Lemma 3, we know that the expression,

$$\sum_{i=1}^{k} d_G(v_i) - \sum_{j=k+1}^{n} d_G(v_j)$$

represents lower bound for the degree sum of an induced subgraph $G[V'] \subseteq G$ with k vertices. Condition (1) implies that the degree sum of any induced subgraph $G[V_k]$ with k vertices exceeds the degree sum $d(H)$ of graph H. In particular $d(G[V_k])$ exceeds the degree-sum of any (induced) subgraph H_k of H, and thus cannot be isomorphic to it. This proves the first statement. Similarly, by arguing on complement graphs $\bar{G}$ and $\bar{H}$, we can prove the second statement. ◀

In light of Theorem 4, when $k \in [1, n]$ satisfies the property of being the smallest number satisfying Case (1) or Case (2), we call k the *dividing number* of the graphs G and H. It can be proven that Case (1) and Case (2) cannot be satisfied simultaneously, not even for different values of k, and that they can only be satisfied for $k > n/2$, so there exists at most one dividing number $k \in (n/2, n]$ between G and H.

Obviously, the bound in Theorem 4 is tighter than the bound of McSplit. To characterize the impact of Theorem 4 on the upper bounds of maximum common subgraphs of G and H, we define the *bound gap* δ_{GH} to be the difference between the bound of McSplit and the new bound in Theorem 4. Let k be the dividing number of G and H. Then $\delta_{GH} = n - k + 1$ if k satisfies Case (1) or Case (2). When there does not exist any dividing number $k \in (n/2, n]$ satisfying Case (1) or Case (2), we set $\delta_{GH} = 0$. This leads to the following corollary.

▶ **Corollary 5.** *For any bidomain $\langle V_l, U_l \rangle \in P_{GH}$, the size of maximum common subgraphs of $G[V_l]$ and $H[U_l]$ is upper bounded by $\min(|V_l|, |U_l|) - \delta$ where δ (the subscript is omitted) refers to the bound gap of $G[V_l]$ and $H[U_l]$.*

Proof. From Theorem 4, we know the δ is a valid reduction that we can apply to obtain a more restricted bound for $\langle V_l, U_l \rangle$. So, in McSplit bound in Equation 1, we can replace $\min(|V_l|, |U_l|)$ with $\min(|V_l|, |U_l|) - \delta$. Hence, the corollary is proved. ◀

Based on Corollary 5, we define the following new bound:

$$UB_{dsb} \leftarrow |M| + \sum_{\langle V_l, U_l \rangle \in P_{GH}} \min(|V_l|, |U_l|) - \delta \tag{2}$$

We shall refer to this new bound as the *Degree Sequence Bound (DSB)*.

McSplit is an exact algorithm that guarantees the retrieval of the maximum common subgraph upon completion. Given that δ provides a valid estimate for the maximum common subgraph of V_l and U_l as stated in Theorem 4, incorporating the proposed bound from Equation (2) preserves the exactness of the McSplit algorithm.

4.2.1 Further Insights

In this section, we further explore the theoretical foundations of Theorem 4, examining the key properties of its conditions. In particular, we demonstrate that the two cases presented in the theorem are mutually exclusive and analyze the feasibility of meeting these conditions based on the parameters n and k. These insights offer a deeper understanding of the theorem.

▶ **Lemma 6.** *Cases (1) and (2) of Theorem 4 are mutually exclusive.*

Proof. Assume both conditions hold. From (1) it follows that $d(G) > d(H)$, while (2) implies that $d(\bar{G}) > d(\bar{H})$. Thus we have $d(G) + d(\bar{G}) > d(H) + d(\bar{H})$, contradicting $|V(G)| \leqslant |V(H)|$. ◀

▶ **Lemma 7.** *Conditions (1) and (2) of Theorem 4 cannot be satisfied for $k \leqslant n/2$. For every (n, k) with $k > n/2$ there exist graph pairs (G_1, H_1) and (G_2, H_2) satisfying conditions (1) and (2).*

Proof. For $k \leqslant n/2$ the lower bounds in (1) and (2) become non-positive, and thus cannot be strictly greater than $d(H)$.
For $k > n/2$ let G_1 be a cycle of length n, so that all vertices have degree two, and H_1 the empty graph. Then the left hand side of the inequality in (1) remains strictly positive, so the condition in (1) is met. For (2) let $(G_2, H_2) = (\bar{G}_1, \bar{H}_1)$. ◀

4.2.2 Complexity Analysis

McSplit has a time complexity of $O(n+m)$ for calculating the bound in Equation (1). The new bound requires $O(m^2)$ time because the computation of $d(H)$ take $O(m^2)$ and the computation of $Deg(G)$ takes $O(n^2 + n\log n)$, where $n \leqslant m$. Since calculating the new bound is asymptotically dominated by the quadratic term m^2, it is desirable to impose a limit of $\sigma_{\max}$ on the size of $|V_l|$ and $|U_l|$ to maintain efficiency, and use the original bound if the limit is exceeded. This constraint is applied to individual bidomains rather than the entire input graph. When appropriately chosen (we used $\sigma_{max} = 16$), this limit does not significantly affect pruning power, as Theorem 4 is more effective with smaller bidomains.

4.3 Learning-based Bounding Heuristic

Since computing the new bound requires additional computational time compared to McSplit bound, it is essential to determine where the new bound can be most effectively utilized during the search - particularly in areas where there is a high likelihood of pruning branches. To address this, we propose incorporating a reinforcement learning agent into a learning-based framework. This agent will dynamically decide when to apply the proposed tighter bound, aiming to maximize both the efficiency and effectiveness of the pruning process.

We denote power sets by $\mathcal{P}(\cdot)$. Let ϕ be a property of bidomains, e.g., given a bidomain $\langle V_l, U_l\rangle$, ϕ may hold if $|V_l|$ or $|U_l|/|V_l|$ is smaller than a threshold. Then a *ϕ-activation function* is defined as $\lambda : \mathcal{P}(V(G)) \times \mathcal{P}(V(H)) \to \{active, inactive\}$ such that $\lambda(V', U') = active$ if $\langle V', U'\rangle$ satisfies the property ϕ. At each branching step, P_{GH} contains a subset of active bidomains $P^{\dagger}_{GH} = \{\langle V_l, U_l\rangle \in P_{GH} \mid \lambda(V_l, U_l) = active\}$.

The reward for an action on P_{GH} is based upon the reward for each active bidomain $\langle V_l, U_l\rangle \in P_{GH}$. This is because the estimated bound gap between the bound of McSplit and the new bound for P_{GH} depends on the estimated bound gaps for its active bidomains. Thus, we define the reward for a bidomain as the bound gap between the bound of McSplit and the new bound.

$$R(V_l, U_l) = \delta_{G[V_l]H[U_l]} \tag{3}$$

The value function of a bidomain $\langle V_l, U_l\rangle$ maintains a score $S(V_l, U_l)$, which is initialized to 1 and is then averaged with the previous score each time to capture the historical information. Let $\alpha \in [0, 1]$. When α is close to 1, more weight is given to recent bound reductions (short-term), whereas when α is close to 0, the bound reductions over the entire history (long-term) are given more weight.

$$S(V_l, U_l) \leftarrow (1-\alpha)S(V_l, U_l) + \alpha R(V_l, U_l) \tag{4}$$

The value function for the set of bidomains maintains a score $S(P_{GH})$ for P_{GH} which reflects the accumulated rewards of its active bidomains.

$$S(P_{GH}) \leftarrow \sum_{\langle V_l, U_l\rangle \in P^{\dagger}_{GH}} S(V_l, U_l) \tag{5}$$

At each branching step, the algorithm checks the following condition to decide whether to compute the new bounds:

$$UB - S(P_{GH}) \leqslant |incumbent| \tag{6}$$

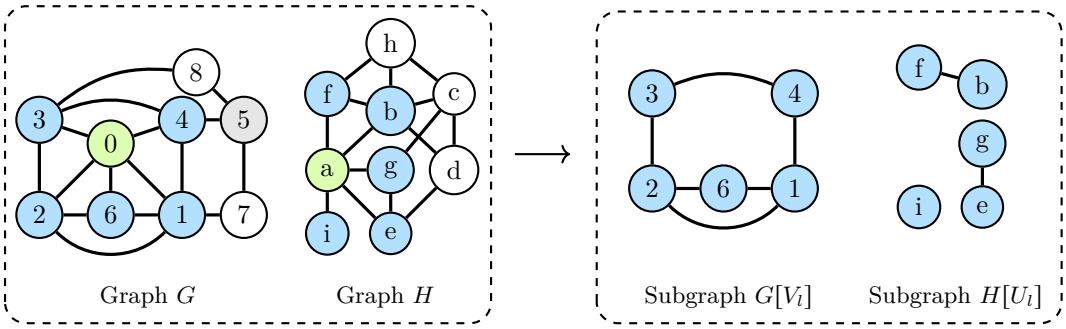
The intention of the above condition is to predict whether computing the new bound can lead to prune the branch. Thus, if the condition is satisfied, the new bound is computed for each active bidomain, and the reward and value scores are updated; otherwise the algorithm uses the bound of McSplit without any changes on rewards and value scores.

Algorithm 1 McSplit+DSB.

```
1  function DSB(P_GH, M):
2      UB_dsb ← |M| + Σ_{⟨V_l,U_l⟩∈P_GH} min(|V_l|, |U_l|)
3      P†_GH ← select active bidomains using φ
4      S(P_GH) ← Σ_{⟨V_l,U_l⟩∈P†_GH} S(V_l, U_l)
5      if UB_dsb − S(P_GH) > |incumbent| then
6          return UB_dsb
7      for ⟨V_l, U_l⟩ ∈ P†_GH do
8          δ ← compute using Theorem 4
9          S(V_l, U_l) ← (1 − α)S(V_l, U_l) + αδ
10         UB_dsb ← UB_dsb − δ
11     return UB_dsb
12
13 function Search(M, P_GH):
14     if |M| > |incumbent| then
15         incumbent ← M
16     bound ← DSB(P_GH, M)
17     if bound ⩽ |incumbent| then
18         return
19     ⟨V_l, U_l⟩ ← max(|V_l|, |U_l|) ↓
20     v ← d_G(v) ↓
21     for w ∈ U_l do
22         P′_GH ← compute new partitions
23         Search(M ∪ {(v, w)}, P′_GH)
24     V_l ← V_l\{v}
25     if |V_l| = 0 then
26         P_GH ← P_GH\{⟨V_l, U_l⟩}
27     Search(M, P_GH)
28
29 function McSplit+DSB(G, H):
30     global incumbent ← ∅
31     Search(∅, {⟨V(G), V(H)⟩})
32     return incumbent
```

4.3.1 Algorithm Description

Algorithm 1 details the proposed algorithm, McSplit+DSB. The function `CalcUB` calculates the upper bound UB_{dsb} by first summing $\min(|V_l|, |U_l|)$ for each bidomain $\langle V_l, U_l \rangle \in P_{GH}$ along with the size of the current mapping $|M|$ (Line 2). Active bidomains are then selected from the partitions P_{GH} and included in $P^{\dagger}_{GH}$ using the ϕ-activation function (Line 3). Then, the score $S(P_{GH})$ is calculated by summing $S(V_l, U_l)$ from these active bidomains (Line 4). If $UB_{dsb} - S(P_{GH})$ is insufficient to prune the branch compared to $|incumbent|$, UB_{dsb} is returned (Lines 5-6). Otherwise, for each active bidomain $\langle V_l, U_l \rangle \in P^{\dagger}_{GH}$, the function calculates δ using Theorem 4, updates the score $S(V_l, U_l)$ with δ, and subsequently updates UB_{dsb} based on δ. The `Search` function is similar to the `Search` function in McSplit [16], except the bound calculation is replaced by the `CalcUB`.



(a) Partitions P_{GH} after matching $(0, a)$ and $(5, \perp)$.

(b) Subgraphs formed by pairs of matching partitions $V_l = \{1, 2, 3, 4, 6\}$ and $U_l = \{b, e, f, g, i\}$.

Figure 1 Figure (a) shows the partitions $P_{GH} = \{\langle V_1, U_1\rangle, \langle V_2, U_2\rangle\}$ after matching $(0, a)$ and $(5, \perp)$, where $V_1 = \{1, 2, 3, 4, 6\}$, $U_1 = \{b, e, f, g, i\}$, $V_2 = \{7, 8\}$, and $U_2 = \{c, d, h\}$. Figure (b) shows the subgraphs $G[V_1]$ and $H[U_1]$, respectively. Since $Deg(G[V_1]) = (2, 2, 2, 3, 3)$ and $d(H[U_1]) = 4$, by Case (1) of Theorem 4, we have the dividing number $k = 4$ for $G[V_1]$ and $H[U_1]$, which yields $|M^*(G[V_1], H[U_1])| \leqslant 3$ and $\delta_{G[V_1]H[U_1]} = 2$. For $G[V_2]$ and $H[U_2]$, we have $\delta_{G[V_2]H[U_2]} = 0$. Hence, although the bound of McSplit is $UB = 8$ by Equation (1), the proposed bound is $UB_{dsb} = 6$ by Equation (2). Assume that $|incumbent| = 7$ during the search, we can prune this branch using UB_{dsb}, while McSplit cannot.

▶ **Example 8.** Figure 1(a) shows the partitioning created by the McSplit algorithm after matching $(0, a)$ and $(5, \perp)$. Matching $(5, \perp)$ means the vertex 5 is removed from further matching. This results in $P_{GH} = \{\langle 1, 2, 3, 4, 6, b, e, f, g, h, i\rangle, \langle 7, 8, c, d, h\rangle\}$. Let $V_1 = \{1, 2, 3, 4, 6\}$ and $U_1 = \{b, e, f, g, h, i\}$, $V_2 = \{7, 8\}$, and $U_2 = \{c, d, h\}$. The degree sequence of $G[V_1]$ is $Deg(G[V_1]) = (2, 2, 2, 3, 3)$, and $d(H[U_1]) = 4$. To satisfy the inequality $\sum_{i=1}^{k} d_G(v_i) - \sum_{j=k+1}^{n} d_G(v_j) > d(H)$ in Theorem 4, we need to select the first 4 elements from $Deg(G[V_1])$, making dividing number $k = 4$ the minimum value to satisfy the inequality. Then, $|M^*(G[V_1], H[U_1])| \leqslant 3$ and $\delta_{G[V_1]H[U_1]} = 2$. For $G[V_2]$ and $H[U_2]$, we have $\delta_{G[V_2]H[U_2]} = 0$. Therefore, according to Corollary 5, $UB_{dsb} = 6$, while the McSplit previous bound gives us $UB = 8$. At this point in the search, we have found the incumbent $M = \{(0, a), (5, c), (4, g), (3, e), (7, h), (6, i), (8, d)\}$ of size 7. Thus, this example demonstrates that the McSplit bound cannot prune this branch, whereas the proposed bound can successfully prune the branch.

5 Experiments

We use a 13th Gen Intel(R) Core(TM) i9-13900K server with 32 CPUs and 128GB of main memory. The implementation of our algorithm and other baselines are in C++ and compile all of them using g++ version 11.4.0 with C++17.

5.1 Benchmarks

We evaluate our approach on multiple standard benchmark datasets for maximum common subgraph detection. The Images-PR15 dataset [22] features a single large target graph (4,838 vertices) and 24 pattern graphs (4-170 vertices), creating 24 test pairs generated from segmented images. The more comprehensive Images CVIU11 dataset [5] offers 6,278 graph

pairs, combining 43 pattern graphs (22-151 vertices) with 146 target graphs (1,072-5,972 vertices). Its counterpart, Meshes CVIU11 [5], provides 3,018 pairs using 6 pattern graphs (40-199 vertices) and 503 target graphs (208-5,873 vertices). For larger-scale testing, the LV dataset [16] contains 2,352 pairs (49 patterns, 48 targets, 10-6,671 vertices), while the LargerLV extension [16] increases this to 3,430 pairs (49 patterns of 10-128 vertices, 70 targets of 138-6,671 vertices). We also include the Scalefree dataset [26, 21] (100 pairs with targets of 200-1,000 vertices and patterns at 90% target size) and the SI dataset [26, 21] (1,170 pairs with targets of 200-1,296 vertices and patterns at 20-60% target size). Finally, the Phase dataset [15] contributes 200 Erdos-Rényi graph pairs (pattern size 30, target size 150), providing a controlled random graph benchmark.

5.2 Solvers

We begin by evaluating our method against the state-of-the-art McSplit algorithm and its variants. For each of these baseline approaches McSplit [16], McSplit+RL [12], McSplit+LL [27], and McSplit+DAL [13], we develop enhanced versions (McSplit+DSB, McSplit+RL+DSB, McSplit+LL+DSB, and McSplit+DAL+DSB) by integrating our proposed bound calculation method. Since McSplit and its variations have shown a considerable advantage over other methods for solving the MCIS problem, such as maximum clique algorithms and constraint programming, we focused our experiments on McSplit and its extensions.

Our experimental evaluation uses all benchmark graph pairs described previously, with a strict 1800-second (30-minute) time limit per instance. We classify problem instances into three categories based on solution time: *easy* (solved within 10 seconds), *moderate* (solved between 10 seconds and 1800 seconds), and *hard* (unsolved within the time limit). This categorization enables detailed analysis of performance across different difficulty levels.

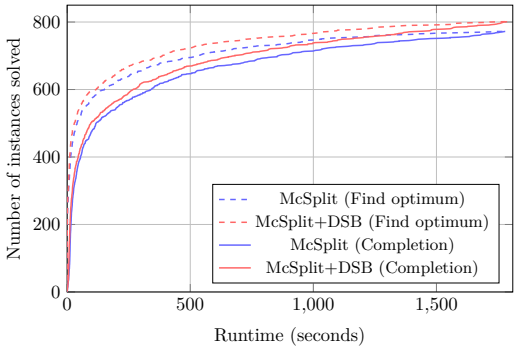


Figure 2 Cactus plot of McSplit and McSplit+DSB on time taken to complete all the enumerations and find the optimum.

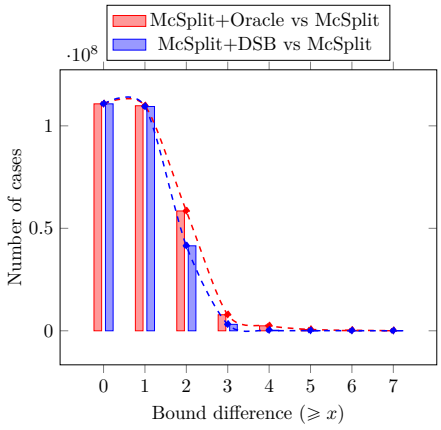


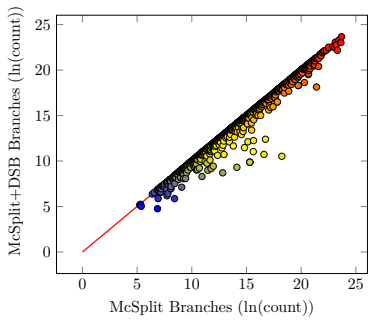
Figure 3 Comparison of the number of cases with bound differences $\geqslant x$ for two distributions: McSplit+Oracle vs. McSplit and McSplit+DSB vs. McSplit.

5.3 Comparison with McSplit

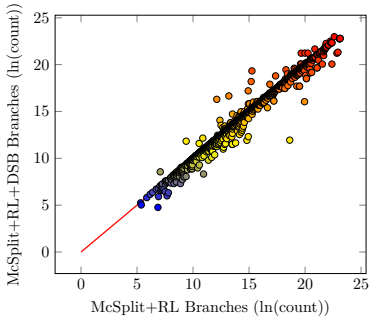
A comprehensive comparative analysis reveals that McSplit+DSB outperforms McSplit by successfully solving a greater number of moderate instances, achieving a notable 3.41% improvement. This enhancement is visually demonstrated in Figure 2, which compares

the performance of McSplit and McSplit+DSB over a 1800-second time frame. The y-axis represents the number of solved instances, while the x-axis denotes the time in seconds. The graph employs dotted lines to indicate the time taken to find the maximum common subgraph (i.e., the optimal solution) and solid lines to represent the time required to exhaustively explore the entire search space. The results clearly illustrate that the integration of the proposed bound significantly boosts the number of solved instances within the McSplit framework. Furthermore, the inclusion of this bound not only accelerates the discovery of the optimal solution but also reduces the overall time needed to verify and complete the search process. The 3.41% improvement refers to instances solved within a fixed time limit, rather than overall running time. Although it may seem modest, this consistent improvement across diverse instances highlights the generality of our approach.

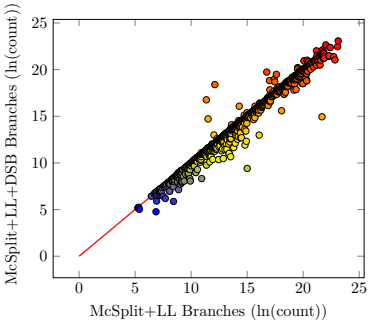
Additionally, Figure 4a provides a comparative analysis of the number of cases solved by McSplit and McSplit+DSB in logarithmic scale. The Y-axis shows the number of branches taken by the McSplit+DSB, while the X-axis shows number of branches taken by McSplit. Each point (y_1, x_1) represents a single instance, where y_1 is the branch count with DSB and x_1 is the count without it. A reference line $y = x$ is included in the plot for comparison. Points below this line indicate that our method reduced the number of branches needed, while points above suggest an increase. As seen in the plots, all the data points lie on or below the $y = x$ line, highlighting the overall efficiency of our method underscoring the computational advantages of the proposed bound. These findings collectively highlight the effectiveness of McSplit+DSB in enhancing both the speed and efficiency of the maximum common subgraph problem.



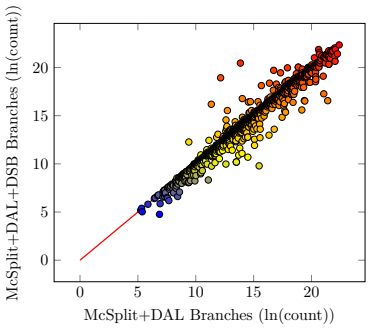
(a) McSplit vs McSplit+DSB.



(b) McSplit+RL vs McSplit+RL+DSB.



(c) McSplit+LL vs McSplit+LL+DSB.



(d) McSplit+DAL vs McSplit+DAL+DSB.

Figure 4 Comparison of the number of search branches for McSplit, McSplit+RL, McSplit+LL, and McSplit+DAL before and after applying the proposed method.

5.4 Comparison with McSplit+RL, McSplit+LL, and McSplit+DAL

Next, we assess the impact of integrating the proposed bound into the machine learning extensions of McSplit, namely McSplit+RL [12], McSplit+LL [27], and McSplit+DAL [13]. Although the bound leads to a similar number of solved instances across these methods, its true value lies in the substantial reduction of computational effort. As illustrated in Figure 4b, Figure 4c, and Figure 4d, the bound dramatically decreases the number of branches explored in moderate cases, with the majority of solved instances exhibiting a branch ratio greater than one. This indicates that the proposed bound calculation consistently explores fewer branches than its counterparts, enhancing efficiency.

However, Figure 4b, Figure 4c, and Figure 4d also reveals a few outlier points that falls above the $y = x$ line, meaning the original algorithm (without DSB) explores fewer branches. This phenomenon occurs because, in certain cases, early pruning disrupts the learning process of dynamic vertex selection models like McSplit+RL. These models rely on iterative refinement of vertex selection, and pruning initial branches can hinder this refinement, altering the selection process in subsequent iterations. In contrast, McSplit, which employs a static degree-based vertex selection method (as shown in Figure 4a), remains unaffected by such disruptions. This nuanced behavior highlights the trade-offs involved in integrating the bound with machine learning extensions, emphasizing its benefits in reducing search complexity while acknowledging its occasional interference with dynamic learning mechanisms.

5.5 Comparison with Oracle

In this experiment, we evaluate how closely our proposed bound approximates the theoretical limit of bound tightness based on bidomains alone. While even tighter bounds are possible in principle by considering relationships between vertices from different bidomains, such approaches run the risk of becoming too expensive to improve overall performance.

For a given bidomain $\langle V_l, U_l \rangle \in P_{GH}$, the exact upper bound or the ultimate bound reduction is computed by determining the maximum common subgraph of the induced subgraphs $G[V_l]$ and $H[U_l]$. We refer to this ideal bound calculation as the *oracle*. Our objective is to compare the deviations of both the proposed bound and the McSplit bound from this oracle. To facilitate this comparison, we introduce McSplit+Oracle, a variant of the McSplit algorithm that replaces its standard bound calculation with the oracle's method of computing the MCS for each matching partition as follows:

$$UB_{oracle} \leftarrow |M| + \sum_{\langle V_l, U_l \rangle \in P_{GH}} |McSplit(G[V_l], H[U_l])|, \tag{7}$$

where $McSplit(G[V_l], H[U_l])$ returns the maximum common subgraph of the induced subgraphs $G[V_l]$ and $H[U_l]$.

Figure 3 visually contrasts the bound differences between McSplit+Oracle and McSplit (represented by red bars) and McSplit+DSB and McSplit (represented by blue bars). The comparison between the red and blue bars allows us to measure the correlation between McSplit+DSB and McSplit+Oracle. Notably, the red and blue plots align closely at $x = 0$ and $x = 1$, indicating strong agreement in McSplit+DSB and McSplit+Oracle. However, as x increases beyond 1, both plots exhibit a noticeable decline.

Crucially it shows that in virtually all cases where Oracle improves on the basic McSplit bound, DSB does as well, and in most of those cases by the same amount. These results highlight the effectiveness of the proposed bound in bridging the gap toward the ultimate bound reduction, offering a significant improvement over the baseline McSplit algorithm.

Table 2 The Larger Common Subgraph Rate indicates the percentage of hard instances in which a given algorithm finds a larger common subgraph than its counterpart. The Δ column represents the difference between the success rates of the algorithm with the proposed bound and the baseline version without it.

Algorithm	Larger Common Subgraph Rate (%)	Δ (%)
McSplit	0.28	3.9
McSplit+DSB	**4.18**	
McSplit+RL	2.26	0.32
McSplit+RL+DSB	**2.58**	
McSplit+LL	1.91	1.73
McSplit+LL+DSB	**3.64**	
McSplit+DAL	5.09	0.18
McSplit+DAL+DSB	**5.27**	

5.6 Further Analysis

MCIS instances that remain unsolved by each algorithm and its branch sampling extension within the 1800-second time limit are categorized as hard instances. While these instances cannot be fully resolved within the given timeframe, identifying a larger common subgraph during this period remains highly valuable for practical applications [14]. Such partial solutions provide critical insights or approximations when exact solutions are out of reach, making them indispensable in real-world scenarios.

Table 2 compares our proposed methods against existing approaches on *hard* instances using the *Larger Common Subgraph Rate*, which quantifies the percentage of cases where our method identifies a larger common subgraph than its competitors. These larger subgraphs are subsequently used as lower bounds to prune more branches, significantly reducing computational time. The results in Table 2 demonstrate that our method consistently outperforms McSplit and its variants when the proposed bound calculation is applied to these difficult cases. In all other scenarios, both methods yield subgraphs of equal size. This performance gain highlights the proposed method's enhanced pruning efficiency and its ability to navigate the search space more effectively. These advantages establish it as a robust and practical solution for tackling complex MCIS problems, particularly under stringent time constraints. By providing larger subgraphs in *hard* cases, our method underscores the importance of integrating bound calculations into MCIS search algorithms.

The frequency with which the new bound is computed depends on both the structure of the input graph pairs and the behavior of the learned model. In our experiments, we found that the new bound was activated in approximately 1.64% of the cases, with a standard deviation of 7.99%.

6 Conclusion

In this research, we demonstrate how leveraging the structural properties of graphs can tighten the upper bound of the BnB algorithm used to solve the MCIS problem. Our reinforcement learning-driven approach strategically optimizes computations by focusing on the most significant areas of the search space. Extensive experiments show that our method consistently outperforms McSplit and its variants in both execution time and the

number of branches explored. Additionally, we establish that our proposed bound calculation closely approximates the maximum possible reduction in bounds achievable during the MCIS search. Further analysis reveals that our approach excels in challenging instances, successfully identifying larger common subgraphs for MCIS compared to existing methods.

Looking ahead, this work opens several promising directions for future research in combinatorial search problems. The principles behind our bound computation could be extended to related problems, such as subgraph isomorphism, to refine bounding strategies in other graph-based search algorithms. While this study focuses on structural properties like vertex degree and edge count to improve the upper bound in MCIS, future work could explore additional graph properties to further enhance bound calculations. Another promising direction is investigating alternative learning paradigms for more efficient application of these improved bounds, potentially leading to even greater performance gains in combinatorial search algorithms.

References

1 Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph-based anomaly detection and description: A survey. *Data Mining and Knowledge Discovery*, 29, April 2014. doi:10.1007/s10618-014-0365-y.

2 Tatsuya Akutsu and Takeyuki Tamura. On the complexity of the maximum common subgraph problem for partial k-trees of bounded degree. In *Algorithms and Computation: 23rd International Symposium, ISAAC 2012, Taipei, Taiwan, December 19-21, 2012. Proceedings 23*, pages 146–155. Springer, 2012. doi:10.1007/978-3-642-35261-4_18.

3 Yunsheng Bai, Derek Xu, Yizhou Sun, and Wei Wang. Glsearch: Maximum common subgraph detection via learning to search. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 588–598. PMLR, July 2021. URL: http://proceedings.mlr.press/v139/bai21e.html.

4 Egon Balas and Chang Sung Yu. Finding a maximum clique in an arbitrary graph. *SIAM Journal on Computing*, 15(4):1054–1068, 1986. doi:10.1137/0215075.

5 Guillaume Damiand, Christine Solnon, Colin de la Higuera, Jean-Christophe Janodet, and Émilie Samuel. Polynomial algorithms for subisomorphism of nd open combinatorial maps. *Comput. Vis. Image Underst.*, 115(7):996–1010, 2011. Special issue on Graph-Based Representations in Computer Vision. doi:10.1016/j.cviu.2010.12.013.

6 Hans-Christian Ehrlich and Matthias Rarey. Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review. *WIREs Computational Molecular Science*, 1(1):68–79, 2011.

7 Ruth Hoffmann, Ciaran McCreesh, and Craig Reilly. Between subgraph isomorphism and maximum common subgraph. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017. doi:10.1609/aaai.v31i1.11137.

8 Xiuzhen Huang, Jing Lai, and Steven Jennings. Maximum common subgraph: Some upper bound and lower bound results. *BMC bioinformatics*, 7 Suppl 4:S6, February 2006. doi:10.1186/1471-2105-7-S4-S6.

9 Takeshi Kawabata and Haruki Nakamura. 3d flexible alignment using 2d maximum common substructure: Dependence of prediction accuracy on target-reference chemical similarity. *Journal of Chemical Information and Modeling*, 54(7):1850–1863, 2014. PMID: 24895842. doi:10.1021/ci500006d.

10 Ina Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1-2):1–30, 2001. doi:10.1016/S0304-3975(00)00286-3.

11 Giorgio Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9(4):341–352, 1973.

12 Yanli Liu, Chu-Min Li, Hua Jiang, and Kun He. A learning based branch and bound for maximum common subgraph related problems. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, volume 34, pages 2392–2399. AAAI Press, April 2020. doi:10.1609/aaai.v34i03.5619.

13 Yanli Liu, Jiming Zhao, Chu-Min Li, Hua Jiang, and Kun He. Hybrid learning with new value function for the maximum common induced subgraph problem. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(4):4044–4051, June 2023. doi:10.1609/aaai.v37i4.25519.

14 Simone Marini, Michela Spagnuolo, and Bianca Falcidieno. From exact to approximate maximum common subgraph. In *Graph-Based Representations in Pattern Recognition: 5th IAPR International Workshop, GbRPR 2005, Poitiers, France, April 11-13, 2005. Proceedings 5*, pages 263–272. Springer, 2005. doi:10.1007/978-3-540-31988-7_25.

15 Ciaran McCreesh, Patrick Prosser, Christine Solnon, and James Trimble. When subgraph isomorphism is really hard, and why this matters for graph databases. *Journal of Artificial Intelligence Research*, 61:723–759, 2018. doi:10.1613/jair.5768.

16 Ciaran McCreesh, Patrick Prosser, and James Trimble. A partitioning algorithm for maximum common subgraph problems. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 712–719. ijcai.org, 2017. doi:10.24963/ijcai.2017/99.

17 James J McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):23–34, 1982. doi:10.1002/spe.4380120103.

18 Samba Ndojh Ndiaye and Christine Solnon. Cp models for maximum common subgraph problems. In Jimmy Lee, editor, *Principles and Practice of Constraint Programming – CP 2011*, pages 637–644, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-23786-7_48.

19 Jochem H. Rutgers, Pascal T. Wolkotte, Philip K.F. Hölzenspies, Jan Kuper, and Gerard J.M. Smit. An approximate maximum common subgraph algorithm for large digital circuits. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 699–705, 2010. doi:10.1109/DSD.2010.29.

20 Kim Shearer, Horst Bunke, and Svetha Venkatesh. Video indexing and similarity retrieval by largest common subgraph detection using decision trees. *Pattern Recognition*, 34(5):1075–1091, 2001. doi:10.1016/S0031-3203(00)00048-0.

21 Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artif. Intell.*, 174(12-13):850–864, 2010. doi:10.1016/j.artint.2010.05.002.

22 Christine Solnon, Guillaume Damiand, Colin de la Higuera, and Jean-Christophe Janodet. On the complexity of submap isomorphism and maximum common submap problems. *Pattern Recognit.*, 48(2):302–316, 2015. doi:10.1016/j.patcog.2014.05.019.

23 Richard S. Sutton and Andrew G. Barto. *Reinforcement learning – An introduction, 2nd Edition*. MIT Press, 2018. URL: http://www.incompleteideas.net/book/the-book-2nd.html.

24 James Trimble. *Partitioning algorithms for induced subgraph problems*. PhD thesis, University of Glasgow, UK, 2023. doi:10.5525/GLA.THESIS.83350.

25 Philippe Vismara and Benoît Valéry. Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. In Le Thi Hoai An, Pascal Bouvry, and Pham Dinh Tao, editors, *Modelling, Computation and Optimization in Information Systems and Management Sciences, Second International Conference, MCO 2008, Metz, France – Luxembourg, September 8-10, 2008. Proceedings*, volume 14 of *Communications in Computer and Information Science*, pages 358–368. Springer, 2008. doi:10.1007/978-3-540-87477-5_39.

26 Stéphane Zampelli, Yves Deville, and Christine Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15:327–353, July 2010. doi:10.1007/s10601-009-9074-3.

27 Jianrong Zhou, Kun He, Jiongzhi Zheng, Chu-Min Li, and Yanli Liu. A strengthened branch and bound algorithm for the maximum common (connected) subgraph problem. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, 2022. doi:10.24963/ijcai.2022/265.

RPID: Rust Programmable Interface for Domain-Independent Dynamic Programming

Ryo Kuroiwa
National Institute of Informatics, Tokyo, Japan
The Graduate University of Advanced Studies, SOKENDAI, Tokyo, Japan

J. Christopher Beck
Department of Mechanical and Industrial Engineering, University of Toronto, Canada

Abstract

In domain-independent dynamic programming (DIDP), a problem is formulated as a dynamic programming (DP) model and then solved by a general-purpose solver. In the existing software for DIDP, a model is defined using expressions composed of a predefined set of operations. In this paper, we propose the Rust Programmable Interface for DIDP (RPID), new software for DIDP, where a model is defined by Rust functions. We discuss the design of RPID and compare it with existing DP-based frameworks, including decision diagram-based (DD-based) solvers. In our experiments, RPID is up to hundreds of times faster than the existing DIDP implementation with the same models. In addition, new DIDP models, enabled by the flexibility of RPID, outperform existing models in multiple problem classes. We also show that the relative performance of RPID and existing DD-based solvers depends on problem class with, so far, no clear dominant solver technology.

2012 ACM Subject Classification Theory of computation → Dynamic programming

Keywords and phrases Decision Diagrams & Dynamic Programming, Modelling & Modelling Languages

Digital Object Identifier 10.4230/LIPIcs.CP.2025.23

Supplementary Material *Software*: `https://github.com/domain-independent-dp/rpid` [13]
archived at `swh:1:dir:9f071b7f5f3daf914c19752ae76692b3a8d1b108`
Software: `https://github.com/Kurorororo/didp-rust-models` [12]
archived at `swh:1:dir:810758e7707cef89647835b7e209d2c94e7bdfd2`

Funding SciNet is funded by ISED Canada; the Digital Research Alliance of Canada; Ontario Research Fund:RE; and the University of Toronto.
J. Christopher Beck: This work is supported by the Natural Sciences and Engineering Research Council of Canada.

Acknowledgements Computations were performed on the Niagara supercomputer at the SciNet HPC Consortium. SciNet is funded by ISED Canada; the Digital Research Alliance of Canada; Ontario Research Fund:RE; and the University of Toronto.

1 Introduction

Dynamic programming (DP) [1] is a problem-solving methodology based on a state-based representation. Recent work has developed model-based paradigms for combinatorial optimization based on DP, where a problem is formulated as a declarative DP model and then solved by a general-purpose solver, similarly to constraint programming (CP). For such paradigms, there are currently two primary directions with different solving approaches: domain-independent dynamic programming (DIDP) [14, 17], using state space search algorithms for solvers, and decision diagram-based (DD-based) solvers [2, 8, 18], using branch-and-bound algorithms with graph data structures called decision diagrams (DDs).

© Ryo Kuroiwa and J. Christopher Beck;
licensed under Creative Commons License CC-BY 4.0

The software implementation of DIDP, didp-rs [14, 17], provides three interfaces: a Rust library, a Python library, and a command-line interface that takes files written in a modeling language as input. In all interfaces, the DP model is described by expressions composed of a predefined set of operations. Internally, the expressions are converted to the same expression tree data structure in Rust and then evaluated by an interpreter. In contrast, the existing two DD-based solvers, ddo [8] and CODD [18], are libraries in which the model is defined by structs and functions in the corresponding programming language.

Ideally, a problem solving system exhibits strong computational performance while allowing modelers to quickly develop and experiment with problem formulations. Research progress for DP-based approaches has been slowed because the competing approaches differ in both the solver technology and the modeling interface. For example, it is unclear whether observed performance differences among solvers are due to underlying advantages of the solver technology or to the requirement for run-time interpretation of expressions. Similarly, it is difficult to determine whether improvements in modeling productivity are worthwhile if there is a performance impact. Indeed, this tension revives earlier debates in the CP community about "packages vs. languages" [20].

We propose the Rust Programmable Interface for DIDP (RPID), a new interface for DIDP, where a DP model is defined using Rust code, similar to ddo and CODD. If writing a model in Rust is acceptable, RPID is a faster and more flexible option for using DIDP than didp-rs. In addition, with RPID, we can compare DIDP and the DD-based solvers, excluding the fundamental differences in interface design. Our contributions are as follows:

- We introduce RPID, novel DIDP software, that is faster and more flexible.
- We discuss design choices in RPID, comparing it to existing software.
- We empirically compare the performance of RPID and existing DIDP and DD-based solver software. We show that RPID is up to hundreds of times faster than didp-rs, and each of RPID, ddo, and CODD outperforms the other two in different problem classes.
- We show that new DIDP models, facilitated by the flexibility of RPID, outperform existing DIDP models in four of five problem classes tested.

2 Dynamic Programming for Combinatorial Optimization

In dynamic programming (DP) [1], a problem is represented by a *state*, and the *value* of the state corresponds to the optimal objective value of the problem. In this paper, we assume that a state S is transformed into another state $S[\![\tau]\!]$, called *a successor state*, by applying a *transition* τ, and the value of a state $V(S)$ is recursively defined by the values of its successor states. To prevent infinite recursion, when a state S satisfies particular conditions, $V(S)$ is non-recursively defined by a function v: $V(S) = v(S)$. In such a case, we call S a *base state*. For computing $V(S)$, we introduce the following assumption: a weight function w_τ is associated with each transition, which returns the *transition weight* $w_\tau(S) \in \mathbb{Q}$ given a state S, and $V(S)$ is computed by applying a binary operator $\circ$, such as $+$, to $w_\tau(S)$ and $V(S[\![\tau]\!])$. Let $\mathcal{T}(S)$ be the set of applicable transitions in a state S. In a minimization problem, $V(S)$ is represented by the following recursive equation, called a Bellman equation:

$$V(S) = \begin{cases} v(S) & \text{if } S \text{ is a base state} \\ \min\limits_{\tau \in \mathcal{T}(S)} w_\tau(S) \circ V(S[\![\tau]\!]) & \text{otherwise.} \end{cases} \tag{1}$$

The optimal objective value of the problem can be computed by solving the above equation. For maximization, min is replaced with max. We assume that $V(S) = \infty$ if $\mathcal{T}(S) = \emptyset$ in the second line ($V(S) = -\infty$ for maximization).

As an example, in the traveling salesperson problem with time windows (TSPTW) [5] we are given a set of n customers $N = \{0, ..., n-1\}$, where 0 is the depot, and the travel time c_{ij} from customer i to j. A solution is a tour starting from the depot at time $t = 0$, visiting each customer i within its time window $[a_i, b_i]$, and returning to the depot. The objective function is to minimize the total travel time $\sum_{i=0}^{n-1} c_{x_i, x_{i+1}}$ where x_i is the i-th customer in the tour with $x_0 = x_n = 0$. In the DP formulation, a state is represented by a set of unvisited customers $U \subseteq N \setminus \{0\}$, the current location $i \in N$, and the current time $t \geq 0$. Each transition visits one of the unvisited customers j that can be reached by the deadline b_j. The Bellman equation is defined as

$$V(U, i, t) = \begin{cases} c_{i0} & \text{if } U = \emptyset \\ \min\limits_{j \in U : t + c_{ij} \leq b_j} c_{ij} + V(U \setminus \{j\}, j, \max\{t + c_{ij}, a_j\}) & \text{if } U \neq \emptyset. \end{cases} \tag{2}$$

The original problem corresponds to state $(N \setminus \{0\}, 0, 0)$.

2.1 Domain-Independent Dynamic Programming (DIDP)

Domain-independent dynamic programming (DIDP) is a model-based paradigm where a combinatorial optimization problem is formulated as a declarative DP model and is solved by a general-purpose solver [14, 17]. The model is defined in Dynamic Programming Description Language (DyPDL), which is explicitly designed for combinatorial optimization by allowing a user to incorporate redundant information via *state constraints*, *state dominance*, and a *dual bound function*.

A state constraint is a condition that must be satisfied by all states. In TSPTW, since we need to visit all customers, a state does not lead to a solution if one of the customers cannot be reached by its deadline. Let c^*_{ij} be the shortest travel time from customer i to j, which can be precomputed from the travel costs. A state (U, i, t) must satisfy $t + c^*_{ij} \leq b_j$ for each customer $j \in U$. Using the value function V,

$$V(U, i, t) = \infty \quad \text{if } \exists j \in U, t + c^*_{ij} > b_j. \tag{3}$$

This constraint is redundant, i.e., implied by Equation (2), but can be helpful for a solver.[1]

A state S dominates another state S' if the value of S is equal to or better than that of S', i.e., $V(S) \leq V(S')$ in minimization. In DyPDL, a user can explicitly define a sufficient condition for state dominance. In the TSPTW example, (U, i, t) is at least as good as (U, i, t') if $t \leq t'$, i.e.,

$$V(U, i, t) \leq V(U, i, t') \quad \text{if } t \leq t'. \tag{4}$$

A dual bound function η defines a lower/upper bound on $V(S)$ in minimization/maximization, i.e., $V(S) \geq \eta(S)$ for minimization. In TSPTW, the travel cost to visit customer j can be underestimated by $c^{\text{to}}_j = \min_{k \in N \setminus \{j\}} c_{kj}$. Given a state (U, i, t), the sum of c^{to}_j over $j \in U \cup \{0\}$ is a dual bound function. Similarly, the travel cost from j to a customer is underestimated by $c^{\text{from}}_j = \min_{k \in N \setminus \{j\}} c_{jk}$. In other words,

$$V(U, i, t) \geq \max \left\{ \sum_{j \in U \cup \{0\}} c^{\text{to}}_j, \sum_{j \in U \cup \{i\}} c^{\text{from}}_j \right\}. \tag{5}$$

[1] In general, a state constraint is not necessarily redundant.

2.1.1 State Space Search Solvers for DIDP

Kuroiwa and Beck [14, 16, 17] developed DIDP solvers using state space search algorithms. In particular, they used heuristic search algorithms such as A* [9] and beam search [21]. At each step, a state space search algorithm selects one state S from a set of candidates. Then, S is *expanded*, i.e., S is removed from the candidates and its successor states are added. In the beginning, the state corresponding to the original problem is the only candidate and, subsequently, the expansion of a state is repeated until termination criteria are met. The solvers use the dual bound function to select a state to expand and prune states that do not lead to a better solution than the current incumbent. The solvers also exploit state dominance to prune states that are known not to be better than another candidate or an already expanded state.

2.1.2 didp-rs: Software Implementation of DyPDL

Kuroiwa and Beck [14, 17] developed didp-rs, a software implementation of DyPDL and its solvers. There are four components: the modeling library dypdl, the solver library dypdl-heuristic-search, the command line interface didp-yaml, and the Python interface DIDPPy, all of which are implemented in Rust.

In didp-rs, states are defined by *state variables*, and each state variable has a type, either of *set*, *element*, *integer*, or *continuous*. In the TSPTW example, U is a set variable, i is an element variable, and t is an integer or continuous variable depending on a problem instance. The state dominance is specified by defining element, integer, and continuous variables as *resource variables* with a preference, either of less or greater. In our example, to represent Inequality (4), t is defined as a resource variable where less is preferred.

Transitions and base states are described by *expressions*, which are composed of predefined operations on state variables and evaluated given a state. In the TSPTW example, the condition to be a base state, $U = \emptyset$, and the condition to apply a transition, $t + c_{ij} \leq b_j$, are expressions returning a Boolean value. To update state variables, *set expression* $U \setminus \{j\}$, *element expression* j, and *integer or continuous expression* $\max\{t + c_{ij}, a_j\}$ are used. Expressions c_{i0} and c_{ij} are used to compute the value of a state. The state constraint (Equation (3)) and the dual bound function (Inequality (5)) are also defined by expressions.

With the dypdl library, a DP model is formulated in Rust by defining state variables and transitions. As of dypdl 0.8.0, an expression is represented as a tree data structure. The library overloads arithmetic operations in Rust and provides functions with which an expression tree can be constructed. The DP model can be solved by calling a solver implemented in dypdl-heuristic-search. During solving, to evaluate an expression tree given a state, a solver uses an interpreter implemented in the dypdl library. The two interfaces, didp-yaml and DIDPPy, convert a DP model written in a specific syntax into the data structure in dypdl and call solvers in dypdl-heuristic-search. In didp-yaml, YAML-DyPDL, a modeling language based on the YAML data format,[2] is implemented employing a LISP-like syntax for expressions. In DIDPPy, a user constructs expressions using Python syntax.

2.2 Decision Diagram-Based Solvers

Hooker [11] proposed that a DP formulation can be represented as a decision diagram (DD), a data structure based on a directed graph, and that a solution can be extracted from such a DD. Based on this observation, Bergman et al. [2] proposed a branch-and-bound algorithm

[2] `https://yaml.org`

to solve DPs based on DDs. Since constructing an exact DD, which fully represents the DP formulation, is computationally expensive, the algorithm repeatedly constructs restricted and relaxed DDs, which are computationally cheaper and provide bounds on the optimal objective value. To construct a relaxed DD, a function called a merge operator is required, which maps two states to a single state. DD-based branch-and-bound can also exploit state dominance and a dual bound function when they are defined [3, 7, 18].

2.2.1 Ddo

Ddo is a software library for DD-based solvers [8] in Rust and uses generics for modeling. In particular, ddo uses *traits*, which define an abstract interface. With traits, a function can be defined generically: one function definition is sufficient for different types of arguments as long as each type implements the required traits. In ddo 2.0.0, a DP model is formulated as a Rust program that defines a data type and implements a particular trait, Problem, for it. The data type for a state is defined as an *associated type*, State, and a concrete type is specified when implementing the trait. Problem requires seven methods:

1. initial_state returns the initial state, which corresponds to the original problem (for TSPTW, $(N \setminus \{0\}, 0, 0)$).
2. nb_variables defines the number of transitions in a solution, which is assumed to be fixed in all solutions. With this assumption, a base state is not defined.
3. for_each_domain defines transitions applied to a given state.
4. transition returns the successor state, given a state and a transition.
5. transition_cost returns the transition weight, given a state, a transition, and the resulting successor state.
6. initial_value returns a constant offset for the objective value. In ddo, the objective value of a solution is the sum of the transition weights and the offset.
7. next_variable controls the behavior of a solver and is not part of a DP model.

In addition, ddo requires the implementation of the Relaxation trait to define a merge operator and the StateRanking trait to define the order to select states to merge using the merge operator during solving. Note that the merge operator is an instruction to a solver rather than model information. A dual bound function is optionally defined in Relaxation, and state dominance is optionally defined by implementing the Dominance trait.

Ddo has a Python interface, Py-DDO, where a DP model is formulated as a Python class implementing a particular set of methods. Internally, Py-DDO defines a Rust struct that has a Python object as a member and implements Problem, Relaxation, and StateRanking for the struct by calling methods of the Python class.

2.2.2 CODD

CODD is a software library for DD-based solvers in C++ [18]. In CODD, a solver takes first-order functions defined as C++ lambda functions as input, representing a DP model and a merge operator. For a DP model, five functions are mandatory, which return the following:

1. Initial state.
2. Base state.
3. Set of applicable transitions, given a state.
4. Successor state, given a state and a transition.
5. Transition weight, given a state and a transition. The objective value of a solution is the sum of the weights.

In addition, a function for a merge operator is required. A dual bound function and state dominance are optionally defined as first-order functions. Unlike ddo, CODD does not assume that the number of transitions in a solution is fixed and explicitly defines a single base state.

3 Rust Programmable Interface for DIDP

We propose the Rust Programmable Interface for DIDP (RPID), a DIDP implementation using a strategy similar to DD-based solvers: a DP model is formulated as a Rust program using traits. To solve a model, we implement heuristic search solvers, following didp-rs.

Listing 1 The DP model for TSPTW in RPID.

```
struct Tsptw { a: Vec<i32>, b: Vec<i32>, c: Vec<Vec<i32>> }
struct S { u: FixedBitSet, i: usize, t: i32 }

impl Dp for Tsptw {
    type State = S;
    type CostType = i32;

    fn get_target(&self) -> Self::State {
        let mut u = FixedBitSet::with_capacity(self.a.len());
        u.insert_range(1..);
        S { u, i: 0, t: 0 }
    }
    fn get_successors(&self, s: &Self::State)
        -> impl IntoIterator<Item = (Self::State, Self::CostType, usize)> {
        s.u.ones().filter_map(|j| {
            if s.t + self.c[s.i][j] > self.b[j] { return None; }
            let mut u = s.u.clone();
            u.remove(j);
            let t = cmp::max(s.t + self.c[s.i][j], self.a[j]);
            if u.ones().any(|k| t + self.c[j][k] > self.b[k]) {
                None
            } else {
                Some((S { u, i: j, t }, self.c[s.i][j], j))
            }
        })
    }
    fn get_base_cost(&self, s: &Self::State) -> Option<Self::CostType> {
        if s.u.is_clear() { Some(self.c[s.i][0]) } else { None }
    }
}
```

We show the DP model for TSPTW formulated with RPID in Listing 1. First, Tsptw, a struct containing data of a problem instance (a and b for the time windows and c for the travel time), is defined. Then, S is defined to represent a state of the DP model, where field u is the set of unvisited customers U, i is the current location i, and t is the current time t. Trait Dp is implemented for Tsptw. Associated type State is used for a state, and CostType is used for the objective value. Dp has three required methods:

1. get_target returns the state corresponding to the original problem, which we call the target state following the convention in DIDP.
2. get_successors returns the successor states, transition weights, and transitions labels, given a state.

3. get_base_cost returns the value of a given state if it is a base state and None otherwise. The return type of get_successors is impl IntoIterator, which means that any data type implementing the IntoIterator trait can be used. The easiest way is to return a dynamic array in the Rust standard library (Vec), which implements IntoIterator. In our example, filter_map is used to avoid allocating memory with Vec. Inside filter_map, each customer j included in s.u is examined, and the transition is labeled with j, representing visiting j. The successor states are filtered by the constraint in Equation (3), assuming $c_{ij} = c^*_{ij}$ for simplicity.

By default, the objective value of a solution is the sum of transition weights and the value of the base state. The binary operator to combine the transition weights ($\circ$ in Equation (1)) can be changed by overriding method combine_cost_weights. Similarly, minimization is assumed by default, and maximization can be selected by overriding method get_optimization_mode.

Listing 2 State dominance in RPID.

```
impl Dominance for Tsptw {
    type State = S;
    type Key = (FixedBitSet, usize);

    fn get_key(&self, s: &Self::State) -> Self::Key {
        (s.u.clone(), s.i)
    }
    fn compare(&self, s1: &Self::State, s2: &Self::State) -> Option<Ordering> {
        Some(s2.t.cmp(&s1.t))
    }
}
```

Listing 3 Dual bound function in RPID.

```
impl Bound for Tsptw {
    type State = S;
    type CostType = i32;

    fn get_dual_bound(&self, s: &Self::State) -> Option<Self::CostType> {
        let sum_to = s.u.ones().map(|j| self.c_to[j]).sum::<i32>();
        let sum_from = s.u.ones().map(|j| self.c_from[j]).sum::<i32>();
        Some(cmp::max(sum_to + self.c_to[0], sum_from + self.c_from[s.i]))
    }
}
```

Implementing the Dominance trait defines a sufficient condition for state dominance (Listing 2). Our design is inspired by ddo. The associated type Key is the type of the key, the part of a state that must be the same if one state dominates another. Given two states, if their keys extracted by the required method get_key are the same, then dominance is checked by method compare. The return value is None or Some(Ordering), an enumerated type in the Rust standard library, with value of Equal, Less, or Greater. Given two states, Equal is returned if they dominate each other, Less if the first state is known to be dominated by the second and not vice versa, Greater if the first is known to dominate the second and not vice versa, and None if no dominance is detected. In our example, Greater is returned if the first state has smaller t.

In practice, Dominance is required by all solvers currently implemented. However, a user does not always need to come up with sufficient conditions to check state dominance. Implementing compare is optional, and it returns Equal by default, meaning two states have the same value if their keys are the same. Thus, the easiest way to implement Dominance is to let get_key return the state itself. In such a case, a solver uses Dominance just to detect

duplicate states to avoid redundant work. A user can also use a customized implementation of get_key without overriding compare when using only parts of the state data structure is sufficient; a state may cache information resulting from expensive computation.

A dual bound function is defined by trait Bound, which has only one required method, get_dual_bound, returning the value of a dual bound function given a state. It returns None if it turns out that the state does not lead to a solution, i.e., $V(S) = \infty$ in minimization. In our example, assuming that fields c_to and c_from are added to Tsptw, corresponding to c^{to} and c^{from} in Inequality (5), the bound is computed as in Listing 3.

3.1 Discussion

We discuss the design of RPID. First, we argue the advantage of RPID over didp-rs. Then, we highlight the differences between RPID and existing DD-based solvers.

3.1.1 RPID vs. didp-rs

In didp-rs, expression trees are used for modeling. As of dypdl 0.8.0, expressions are designed for declarative definitions and are somewhat limited; loops cannot be used, and thus, implementing complicated algorithmic procedures or data structures is difficult. In the TSPTW example, as a dual bound function, we could use the minimum spanning tree (MST) weight in a complete graph, where U is the set of nodes and c_{jk} is the weight of edge (j, k). We can further improve this bound by constructing a 1-tree [10]: since U does not include the current location i and the depot 0, the cheapest edge from i to a customer in U and the cheapest edge from a customer in U to the depot are added to the MST. However, computing the MST weight using expressions is difficult in didp-rs. In contrast, RPID, ddo, and CODD are more flexible since we can directly write algorithms in the programming languages in which they are implemented.

Another potential advantage of RPID over didp-rs is performance. Methods in RPID are directly compiled Rust and so running them is substantially faster than evaluating expressions using the intermediate interpreter in didp-rs.

Note, however, that RPID complements didp-rs rather than replacing it; didp-rs is preferred in some use cases. For example, didp-rs provides a Python interface, DIDPPy. We could provide a Python interface for RPID similar to Py-DDO by defining a Rust struct with a Python object as a member. However, it may not be as efficient as DIDPPy since we need to call Python functions during solving. In addition, with explicit expression trees, a DIDP solver can analyze and exploit particular structures of expressions, as done in Kuroiwa and Beck [15]. In contrast, when components of a DP model are implemented as Rust methods, they become black-boxes for a solver, and such analysis and exploitation are difficult.

We could also port the didp-rs interfaces to RPID by implementing structs and traits that parse expressions. While it is unclear that such a system would have any computational advantages over didp-rs, it would facilitate a workflow of rapid prototyping in the didp-rs interface followed by production implementation in the lower-level Rust. Such a unification is one direction for future work.

3.1.2 Traits vs. Functions

A design advantage of traits is their explicitness: required methods and their signatures are clear for a user from the trait definitions. The downside is that the struct implementing the trait may have many fields to provide all necessary information to each method, e.g., a, b, c,

c_to, and c_from in Tsptw. In contrast, CODD can avoid defining such a struct since each first-order function implemented as a C++ lambda function captures necessary information in addition to its arguments.

3.1.3 Successor Generation

Ddo and CODD use separate functions to identify applicable transitions, apply each transition to generate the successor state, and compute the transition weight. In contrast, RPID performs all of them at once in get_successors. This design choice was made for two reasons.

First, successor generation becomes more explicit. When reading a model, a user does not need to refer to multiple methods to understand how successor states are generated. While decomposing the successor generation function into pieces may be useful when the function is complicated, a user can do that in their own way, not forced by the trait definition.

Second, successor generation becomes more efficient when the same information is required in multiple places. Since it is not the case with TSPTW, we introduce single machine total weighted tardiness $(1||\sum w_i T_i)$ as a motivating example. In this problem, a set of jobs N is processed on a single machine, and each job $j \in N$ has the processing time p_j, the due date d_j, and the weight w_j. The optimal solution is a sequence of the jobs that minimizes the total weighted tardiness $\sum_{j \in N} w_j \max\{C_j - d_j, 0\}$, where C_j is the completion time of job j. In our DP model, we represent a state by a single state variable S, representing the set of processed jobs, and process one job $j \in N \setminus S$ in each decision. We present the DP model in Equation (6) and its implementation with RPID in Listing 4.[3]

$$V(S) = \begin{cases} 0 & \text{if } S = N \\ \min\limits_{j \in N \setminus S} w_j \max\left\{\left(\sum_{k \in S} p_k\right) + p_j - d_j, 0\right\} + V(S \cup \{j\}) & \text{if } S \neq N. \end{cases} \tag{6}$$

Listing 4 The DP model for $1||\sum w_i T_i$ in RPID.

```
struct Wt { p: Vec<i32>, d: Vec<i32>, w: Vec<i32> }

impl Dp for Wt {
    type State = FixedBitSet;
    type CostType = i32;

    fn get_target(&self) -> Self::State {
        FixedBitSet::with_capacity(self.p.len())
    }
    fn get_successors(&self, s: &Self::State)
        -> impl IntoIterator<Item = (Self::State, Self::CostType, usize)> {
        let t = s.ones().map(|k| self.p[k]).sum::<i32>();
        s.zeroes().map(move |j| {
            let mut next_s = s.clone();
            next_s.insert(j);
            let tardiness = cmp::max(t + self.p[j] - self.d[j], 0);
            (next_s, self.w[j] * tardiness, j)
        })
    }
    fn get_base_cost(&self, s: &Self::State) -> Option<Self::CostType> {
        if s.is_full() { Some(0) } else { None }
    }
}
```

[3] Our actual DP model used in the experimental evaluation exploits precedence between jobs extracted by preprocessing following Kuroiwa and Beck [16, 17].

In this model, the time job j starts is $\sum_{k \in S} p_k$, which requires $O(|N|)$ time to compute. If we compute it for each j, we need $O(|N|^2)$ computation to generate all successors. To avoid such computation, we could introduce a redundant state variable t that is increased by p_j when j is processed, but this approach increases the amount of memory used for each state. In our implementation with RPID, we compute $\sum_{k \in S} p_k$ only once, save it as a Rust variable `t`, and use it for each `j`. We use similar approaches in the DP models for the minimization of open stacks problem and talent scheduling evaluated in Section 4.

A disadvantage of this design is that a solver has less flexibility in generating successor states: it cannot compute applicable transitions, the successor states, and the transition weights separately, potentially limiting the design of a solving algorithm. In the current implementation, our heuristic search solvers are not affected by this restriction.

4 Empirical Evaluation

We compare the performance of RPID against existing DIDP and DD-based solvers. Building on previous DIDP solvers [14, 16, 17], we implement cost-algebraic A* [6] and complete anytime beam search (CABS) [21]; A* is a fundamental algorithm in heuristic search, and CABS performs the best in the existing DIDP solvers due to its memory efficiency. We publish the source code for RPID[4] and DP models.[5] We use Rust 1.76.0 for all solvers implemented in Rust. For each problem instance, we use a 30-minute time limit, an 8GB memory limit, and a single core of Intel Xeon Gold 6418, run with GNU parallel [19].

4.1 RPID vs. didp-rs

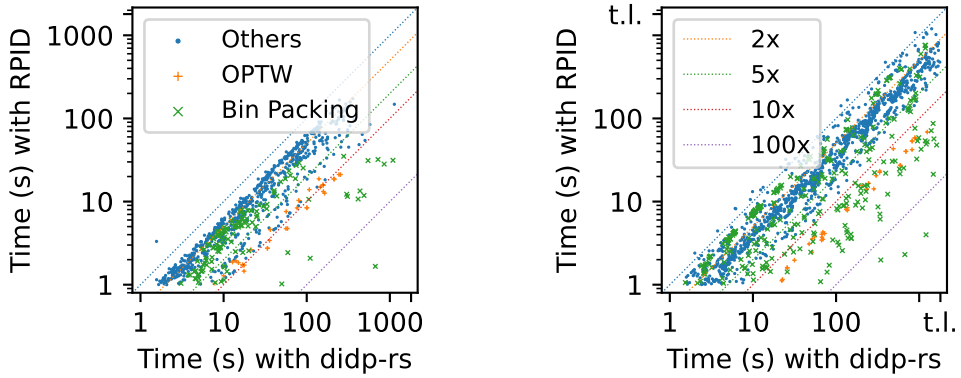
With didp-rs, previous work [17] used DP models for eleven problem classes: TSPTW, the capacitated vehicle routing problem (CVRP), the multi-commodity pickup and delivery traveling salesperson problem (m-PDTSP), the orienteering problem with time windows (OPTW), the multi-dimensional knapsack problem (MDKP), bin packing, the simple assembly line balancing problem (SALBP-1), $1||\sum w_i T_i$, talent scheduling, the minimization of open stacks problem (MOSP), and graph-clear. DIDP solvers in didp-rs outperformed commercial CP and mixed-integer programming solvers in TSPTW, m-PDTSP, OPTW, SALBP-1, $1||\sum w_i T_i$, talent scheduling, MOSP, and graph-clear. The original models are available in YAML-DyPDL and DIDPPy in a public repository,[6] and we reimplement them in RPID. As a baseline, we also reimplement these models in Rust using didp-rs 0.8.0: we define the DP models by writing expressions with the dypdl library and solve them using the dypdl-heuristic-search library in Rust. We call this baseline "didp-rs."

We confirmed that the numbers of expanded states are the same in RPID and didp-rs for the same solver and the same problem instance. Thus, we compare the number of optimally solved instances and the average time to solve an instance in each problem class. As shown in Table 1, RPID dominates didp-rs: solving more instances than didp-rs in six problem classes with A* and eight with CABS and never failing to solve an instance solved by didp-rs. On average, RPID is faster than didp-rs in all problem classes and reduces the solving time by at least half in five classes with A* and eight with CABS. We also show the number of solved instances against time for TSPTW, m-PDTSP, and OPTW in Figure 2 and for other problem classes in Appendix B.

[4] `https://github.com/domain-independent-dp/rpid/releases/tag/v0.1.0`
[5] `https://github.com/Kurorororo/didp-rust-models`
[6] `https://github.com/Kurorororo/didp-models`

Table 1 RPID vs. didp-rs with the same DP models. '#opt' is the number of optimally solved instances, and "time" is the time in seconds to solve an instance optimally, averaged over instances where didp-rs takes at least 1 second. The higher value of '#opt' is in bold, and "time" by RPID is in bold if less than half of that of didp-rs.

	A*				CABS			
	didp-rs		RPID		didp-rs		RPID	
	#opt	time	#opt	time	#opt	time	#opt	time
TSPTW (340)	257	64	257	**17**	259	196	**267**	**44**
CVRP (207)	6	54	6	29	6	118	6	**46**
m-PDTSP (1178)	953	24	953	14	1034	149	**1049**	**66**
OPTW (144)	64	54	64	**5**	64	212	**85**	**13**
MDKP (276)	4	3	4	2	5	63	5	33
Bin Packing (1615)	922	42	**939**	**4**	1168	140	**1230**	**33**
SALBP-1 (2100)	1657	31	**1667**	**8**	1801	195	**1821**	**71**
$1\|\|\sum w_i T_i$ (375)	270	31	**277**	18	288	173	**299**	**66**
Talent Scheduling (1000)	207	62	**225**	**26**	235	277	**257**	**88**
MOSP (570)	483	8	**487**	4	527	172	527	142
Graph-Clear (135)	78	17	**80**	11	103	173	**104**	135



(a) A*. **(b)** CABS.

Figure 1 Time in seconds to solve each instance optimally (only instances with at least 1 second).

We present an instance-wise comparison of the solving time by RPID and didp-rs in Figure 1. In the majority of instances, RPID achieves a speedup of more than two times. For A*, RPID is slower in one SALBP-1 instance, where we observe that RPID proves optimality faster than didp-rs but takes more time for termination, including freeing allocated memory. For CABS, RPID is slower than didp-rs in 19 MOSP instances. This performance degradation seems to be due to the implementation of state variables. The DP model for MOSP has two set state variables, and each transition performs set operations on them. Both our RPID model and didp-rs use the same library, fixedbitset, to represent set state variables. However, while didp-rs 0.8.0 uses fixedbitset 0.4.2, our model uses 0.5.7 (the latest version as of writing). We observe that using 0.4.2 with our RPID model results in better performance than didp-rs. We suspect that the overhead of didp-rs is relatively low in MOSP, sometimes outweighed by the difference in the set variable implementation.

Table 1 and Figure 1 present large performance gains in bin packing and OPTW. In particular, RPID achieves more than 100 times speedup in some bin packing instances. In bin packing, we minimize the number of bins to pack a set of items N, where a bin has the

Table 2 Comparison of models with the new dual bound functions vs. the models by Kuroiwa and Beck [17] using RPID. We use the 1-tree bound for TSPTW, CVRP, and m-PDTSP and the Dantzig bound for OPTW and MDKP. '#opt' is the number of optimally solved instances, "time" is the time in seconds to solve an instance optimally, and '#expanded' is the number of expansions before the last f-layer. "time" and '#expanded' are averaged over instances where the model by Kuroiwa and Beck [17] takes at least 1 second. Better values are bolded.

	Kuroiwa and Beck [17]			1-Tree/Dantzig		
A*	#opt	time	#expanded	#opt	time	#expanded
TSPTW (340)	257	**25**	3,681,570	257	89	**3,344,561**
CVRP (207)	6	29	3,577,960	**8**	**5**	**390,277**
m-PDTSP (1178)	953	19	2,516,593	**1075**	**5**	**356,029**
OPTW (144)	64	8	2,897,171	**71**	**4**	**1,220,461**
MDKP (276)	4	2	735,507	**6**	**0**	**342**
CABS	#opt	time	#expanded	#opt	time	#expanded
TSPTW (340)	**267**	**161**	16,855,270	247	287	**4,420,454**
CVRP (207)	6	55	9,579,821	**8**	**19**	**989,025**
m-PDTSP (1178)	1049	129	18,352,633	**1074**	**34**	**820,919**
OPTW (144)	85	340	78,952,253	**92**	**97**	**20,692,072**
MDKP (276)	5	33	16,356,224	**6**	**0**	**29,756**

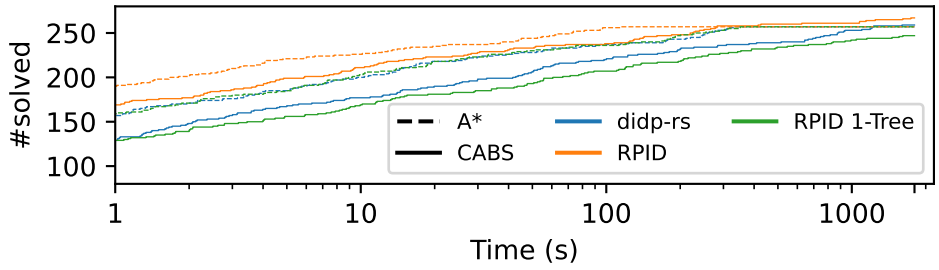
capacity c, and each item $i \in N$ has the weight w_i. In the DP model, a state is represented by the remaining capacity of the current bin r and the set of unpacked items U. Each transition packs an item $i \in U$ with $w_i \leq r$ and reduces r by w_i. If no such item exists, only one transition is applicable, which opens a new bin and packs an arbitrary item i, updating r to $c - w_i$. In didp-rs, two transitions are defined for each item i, one to pack it in the current bin and another to pack it in a new bin. The latter has a precondition $j \notin U \vee w_j > r$ for each $j \in N$ to confirm that no item can be packed in the current bin, requiring $O(|N|)$ time to check in the worst case. Since there are $O(|N|)$ such transitions, identifying applicable transitions requires $O(|N|^2)$ time. In RPID, we can avoid such computation by checking $\forall j \in U, w_j > r$ only once in the successor generator function, as presented in Appendix A.

For OPTW, the restrictions of the DyPDL syntax result in a large expression tree with less efficient execution than what can be done with Rust code. In the model, the dual bound function computes $\sum_{j \in U : \phi_j} c_j$, where $U \subseteq N$ is a set state variable, ϕ_j is a Boolean condition on j, and c_j is a constant depending on j. To represent this computation, the current implementation takes the sum of expressions representing "c_j if $j \in U \wedge \phi_j$ and 0 otherwise" for all $j \in N$, resulting in an expression tree with depth $O(|N|)$.

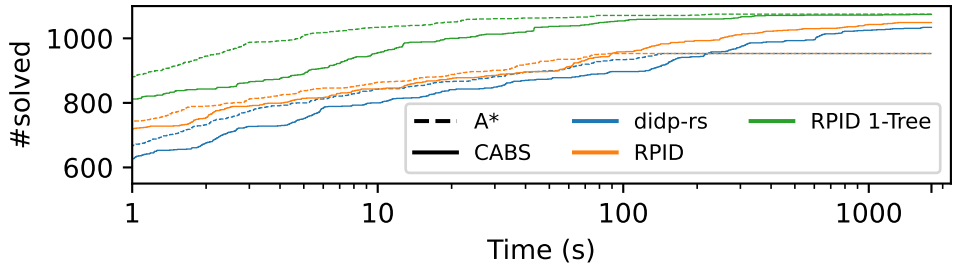
4.2 Comparison of Dual Bound Functions

As we discussed in Section 3.1.1, a dual bound function based on the 1-tree weight can be used for the DP model of TSPTW. The DP models for CVRP and m-PDTSP are similar to that of TSPTW, and dual bound functions similar to Inequality (5) are used, so they can also be replaced with the 1-tree bound. We implement such DP models in RPID, using Kruskal's algorithm to compute the MST weight, which requires sorting edges in the ascending order of the weights. In our implementation, sorting is performed in preprocessing, and Kruskal's algorithm ignores edges connected to visited customers in a given state.

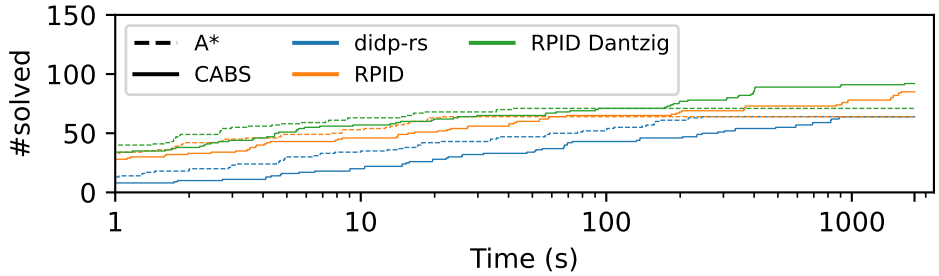
Since OPTW and MDKP can be considered generalizations of 0-1 knapsack, the Dantzig bound [4] can be used as a dual bound function. Due to restrictions of expressions in didp-rs, Kuroiwa and Beck [17] approximated the Dantzig bound in their DP models. In RPID, we



(a) TSPTW.



(b) m-PDTSP.



(c) OPTW.

Figure 2 Time in seconds vs. the number of optimally solved instances.

implement DP models using the Dantzig bound as a dual bound function. We compare the new DP models with the original models (the ones used in Section 4.1) using A* and CABS in RPID, measuring the number of state expansions by each heuristic search algorithm to solve an instance optimally. Our heuristic search algorithms maintain the global dual bound, a lower/upper bound of the optimal objective value in minimization/maximization. We subtract the number of expansions after the global dual bound matches the optimal objective value from the total number of expansions. This metric is called "expansions before the last f-layer" and is conventionally used to compare different heuristic functions for A*.

As shown in Table 2, our dual bound functions reduce the number of expansions in all problem classes. In CVRP, m-PDTSP, OPTW, and MDKP, the number of optimally solved instances is increased, and the time to solve an instance is reduced by a factor of two to six. We also present the number of solved instances against time for TSPTW, m-PDTSP, and OPTW in Figure 2. The result shows that the flexibility of the RPID modeling can contribute to significant performance improvement. However, in TSPTW, the time to solve an instance is increased, and the number of instances solved optimally by CABS is decreased.

Table 3 Comparison of the RPID model implementations for $1||\sum w_i T_i$ where $\sum_{k \in S} p_k$ is computed once for all successors (Original), separately computed for each successor (Separate), and cached as a state variable (StateCache). '#opt' is the number of optimally solved instances, and "time" is the time in seconds to solve an instance optimally, averaged over co-solved instances where Original takes at least 1 seconds. The best value is bolded.

	Original		Separate		StateCache	
	#opt	time	#opt	time	#opt	time
A*	**277**	**27**	**277**	28	274	**27**
CABS	**299**	**139**	295	154	298	144

One potential reason is the quadratic computational complexity of Kruskal's algorithm on a complete graph compared to the linear complexity of the dual bound function in Inequality (5). While the expensive dual bound function pays off in CVRP and m-PDTSP, it does not in TSPTW, possibly because many states are already pruned by Equation (3).

4.3 Impact of the Successor Generation Interface

As discussed in Section 3.1.3, the single successor generation function of RPID can be beneficial when the same information is required by multiple transitions. We evaluate the impact of this interface design using the DP model for $1||\sum w_i T_i$, presented in Equation (6). In our original implementation presented in Listing 4 and used in Section 4.1, the total processing time of already processed jobs, $\sum_{k \in S} p_k$, is computed once for all successor states. We consider two different implementations, "Separate", where $\sum_{k \in S} p_k$ is separately computed each time a successor state is generated, and "StateCache", where $\sum_{k \in S} p_k$ is stored as a state variable and increased by p_j when j is added to S.

We compare the three implementations in Table 3. With A*, Separate slightly increases the average time to solve an instance, and StateCache reaches the memory limit in three instances solved by Original and Separate, possibly due to increased memory usage per state. With CABS, both Separate and StateCache increase average solving time and solve fewer instances than Original due to the time limit. These results confirm that our interface design has a positive impact on performance.

4.4 RPID vs. Ddo and CODD

We compare RPID, didp-rs, ddo, and CODD using the problem classes with which previous work compared CODD with didp-rs and ddo [18]: 0-1 knapsack, Golomb ruler, and the maximum independent set problem (MISP). For 0-1 knapsack and MISP, instances are retrieved from the CODD repository.[7] For Golomb ruler, an instance is uniquely determined from a parameter n. While previous work compared heuristic search solvers and DD-based solvers [17, 18], the interface designs of the solvers are different; didp-rs uses expression trees and DD-based solvers use functions in the programming language in which they are implemented. With RPID, we conduct the first empirical evaluation comparing heuristic search solvers and DD-based solvers without such differences.

[7] `https://github.com/ldmbouge/CODD/tree/main/data`

Table 4 RPID vs. didp-rs, ddo, and CODD in time in seconds to optimally solve a 0-1 knapsack instance. "t.o." indicates time out, and "m.o." indicates memory out. The smallest value is in bold.

	Dantzig					No Dantzig			
	Ddo	CODD		RPID		didp-rs		RPID	
Instance	width=4	width	time	A*	CABS	A*	CABS	A*	CABS
PI:1 2000	0.37	64	0.76	**0.12**	0.25	2.72	2.96	1.13	0.92
PI:1 5000	0.47	64	2.73	**0.13**	0.28	175.14	106.83	41.58	32.77
PI:1 10000	0.71	64	m.o.	**0.16**	0.28	t.o.	1227.55	566.74	321.07
PI:2 2000	0.16	64	3.70	**0.02**	0.04	1.05	1.73	0.42	0.61
PI:2 5000	0.44	64	m.o.	**0.15**	0.29	52.65	64.20	13.97	21.09
PI:2 10000	0.76	64	m.o.	**0.16**	0.27	835.28	870.12	220.60	222.25
PI:3 1000	0.26	1024	**0.08**	0.14	0.27	0.90	1.53	0.50	0.48
PI:3 2000	3.23	2048	3.47	**0.34**	1.38	10.92	14.27	3.40	4.89
PI:3 5000	4.87	4096	6.29	**0.74**	4.83	243.29	297.93	66.10	81.09
PI:3 10000	4.32	8192	m.o.	**1.26**	8.73	t.o.	t.o.	651.70	656.50

We use ddo 2.0.0 and the latest model implementations available in its repository at the time of writing.[8] Ddo requires a parameter called a width as input, and each model code defines a default width. In 0-1 knapsack, we use a width of 4 as it performs better than the default width of 2. In the other two problem classes, we use the default parameters, 10 for Golomb ruler and the width automatically decided based on a problem instance in MISP.

For CODD, from models in its repository,[9] we use knapsack2 for 0-1 knapsack, gruler_midlb for Golomb ruler, and misp5 for MISP,[10] compiled with GCC 13.2.0. Similar to ddo, CODD also requires a width parameter as input. Following the previous work using the same instances [18], we use 64 for Golomb ruler and 128 for MISP. In 0-1 knapsack, we use the best width for each instance reported in the previous work [18].

For didp-rs and RPID, we implement models following the CODD models. For 0-1 knapsack, ddo, CODD, and RPID models use the Dantzig bound as a dual bound function. As mentioned earlier, the Dantzig bound is difficult to use with didp-rs, so we define a dual bound function similar to those used in the DP models for MDKP and OPTW, and we also evaluate a RPID model with such a dual bound function (No Dantzig).

Tables 4–6 present the results, omitting instances solved within 1 second by all solvers. There is no single winner: RPID with A* shows a clear advantage in 0-1 knapsack, CODD is the best in Golomb ruler, and ddo is almost always the best in MISP. While previous work reported that didp-rs with CABS fails to solve four 0-1 knapsack instances in their evaluation [18], it solves all but one in our evaluation. We suspect that previous work did not use any dual bound function with DIDP, following the model in the didp-rs repository.[11] In our evaluation, CODD reaches the 8GB memory limit in four instances of 0-1 knapsack. Given a larger memory limit, CODD solves all such instances in at most a few tens of seconds, but it is slower than DDO and RPID with the Dantzig bound. Further analysis of the performance difference between DIDP and DD-based solvers is left for future work.

[8] https://github.com/xgillard/ddo/tree/3b39798874b66ac965a0ce915c6f21f562ebaa6e/ddo/examples
[9] https://github.com/ldmbouge/CODD/tree/main/examples
[10] They seem to be the best models according to authors' description and our preliminary experiments.
[11] https://github.com/domain-independent-dp/didp-rs/blob/main/didppy/examples/knapsack.ipynb

Table 5 RPID vs. didp-rs, ddo, and CODD in time in seconds to optimally solve a Golomb ruler instance. "t.o." indicates time out, and "m.o." indicates memory out. The smallest value is in bold.

	Ddo	CODD	didp-rs		RPID	
n	width=10	width=64	A*	CABS	A*	CABS
8	0.71	**0.04**	2.61	5.73	1.88	1.19
9	6.89	**0.20**	26.25	51.49	14.02	13.85
10	50.55	**0.68**	m.o.	532.55	m.o.	143.84
11	m.o.	**10.51**	m.o.	t.o.	m.o.	m.o.
12	m.o.	**55.56**	m.o.	t.o.	m.o.	t.o.
13	m.o.	**1318.98**	m.o.	t.o.	m.o.	m.o.
14	m.o.	t.o.	m.o.	t.o.	m.o.	t.o.

Table 6 RPID vs. didp-rs, ddo, and CODD in time in seconds to optimally solve an MISP instance. "t.o." indicates time out, and "m.o." indicates memory out. The smallest value is in bold.

	Ddo	CODD	didp-rs		RPID	
Instance		width=128	A*	CABS	A*	CABS
johnson8-4-4	**0.17**	0.24	0.81	1.04	0.70	0.58
johnson16-2-4	2.64	1.43	1.04	1.11	0.92	**0.62**
keller4	**5.47**	29.63	27.55	65.46	15.62	39.02
hamming6-2	**0.17**	0.28	2.84	14.22	1.56	9.07
hamming8-2	**0.30**	63.69	m.o.	t.o.	m.o.	t.o.
hamming8-4	**29.65**	36.81	m.o.	m.o.	m.o.	m.o.
hamming10-2	**10.01**	1520.08	m.o.	t.o.	m.o.	t.o.
brock200-1	**403.20**	t.o.	m.o.	t.o.	m.o.	m.o.
brock200-2	**1.10**	3.68	5.68	15.09	3.41	8.72
brock200-3	**7.88**	22.21	m.o.	131.98	m.o.	80.07
brock200-4	**22.67**	102.52	m.o.	797.01	m.o.	455.68
p_hat300-1	**0.49**	1.25	2.09	1.53	1.42	1.01
p_hat300-2	**19.82**	317.83	m.o.	t.o.	m.o.	m.o.

5 Conclusion

We propose new software for domain-independent dynamic programming (DIDP): the Rust Programmable Interface for DIDP (RPID). We use traits in Rust for modeling, following an existing decision diagram-based (DD-based) solver. RPID is novel in that the successor generation is defined in a single function, motivated by readability and efficiency. As DIDP software, RPID enables flexible modeling and fast execution by using Rust functions. Our experiment shows that given the same models, RPID is faster than the existing DIDP implementation, didp-rs, in most cases, and further performance improvement is achieved with the better models facilitated by the flexibility of RPID. We also demonstrate that the relative performance of RPID and existing DD-based solvers changes by problem classes.

As we discussed, didp-rs is preferred to RPID in some use cases that are particularly related to model development and analysis. To close the performance gap between didp-rs and RPID, improving the flexibility and efficiency of expressions in didp-rs is an important direction. For example, we may want to implement specialized expressions for particular computations that are commonly used in DIDP models.

References

1 Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

2 David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and J. N. Hooker. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016. doi:10.1287/ijoc.2015.0648.

3 Vianney Coppé, Xavier Gillard, and Pierre Schaus. Modeling and exploiting dominance rules for discrete optimization with decision diagrams. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research – 21st International Conference, CPAIOR 2024*. Springer Nature Switzerland, 2024. doi:10.1007/978-3-031-60597-0_15.

4 George B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):266–277, 1957. doi:10.1287/opre.5.2.266.

5 Yvan Dumas, Jacques Desrosiers, Eric Gelinas, and Marius M Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations Research*, 43(2):367–371, 1995. doi:10.1287/opre.43.2.367.

6 Stefan Edelkamp, Shahid Jabbar, and Alberto Lluch Lafuente. Cost-algebraic heuristic search. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI)*, pages 1362–1367. AAAI Press, 2005. URL: http://www.aaai.org/Library/AAAI/2005/aaai05-216.php.

7 Xavier Gillard, Vianney Coppé, Pierre Schaus, and André Augusto Cire. Improving the filtering of branch-and-bound MDD solver. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research – 18th International Conference, CPAIOR 2021*, pages 231–247. Springer International Publishing, 2021. doi:10.1007/978-3-030-78230-6_15.

8 Xavier Gillard, Pierre Schaus, and Vianney Coppé. Ddo, a generic and efficient framework for MDD-based optimization. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 5243–5245. International Joint Conferences on Artificial Intelligence Organization, 2020. Demos. doi:10.24963/ijcai.2020/757.

9 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi:10.1109/TSSC.1968.300136.

10 Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970. doi:10.1287/opre.18.6.1138.

11 John 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*, pages 94–110. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-38171-3_7.

12 Ryo Kuroiwa. DIDP Rust Models. Software, swhId: swh:1:dir:810758e7707cef89647835b7e209d2c94e7bdfd2 (visited on 2025-07-28). URL: https://github.com/Kurorororo/didp-rust-models, doi:10.4230/artifacts.24206.

13 Ryo Kuroiwa and J. Christopher Beck. RPID. Software, version 0.1.0., swhId: swh:1:dir:9f071b7f5f3daf914c19752ae76692b3a8d1b108 (visited on 2025-07-28). URL: https://github.com/domain-independent-dp/rpid, doi:10.4230/artifacts.24205.

14 Ryo Kuroiwa and J. Christopher Beck. Domain-independent dynamic programming: Generic state space search for combinatorial optimization. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 236–244. AAAI Press, 2023. doi:10.1609/icaps.v33i1.27200.

15 Ryo Kuroiwa and J. Christopher Beck. Large neighborhood beam search for domain-independent dynamic programming. In *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CP.2023.23.

16 Ryo Kuroiwa and J. Christopher Beck. Solving domain-independent dynamic programming problems with anytime heuristic search. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 245–253. AAAI Press, 2023. doi:10.1609/icaps.v33i1.27201.

17 Ryo Kuroiwa and J. Christopher Beck. Domain-independent dynamic programming, 2025. arXiv:2401.13883. doi:10.48550/arXiv.2401.13883.

18 Laurent Michel and Willem-Jan van Hoeve. CODD: A decision diagram-based solver for combinatorial optimization. In *ECAI 2024 – 27th European Conference on Artificial Intelligence*, volume 392 of *Frontiers in Artificial Intelligence and Applications*, pages 4240–4247. IOS Press, 2024. doi:10.3233/FAIA240997.

19 Ole Tange. GNU parallel – The command-line power tool. *;login: The USENIX Magazine*, 36:42–47, 2011. URL: https://www.usenix.org/publications/login/february-2011-volume-36-number-1/gnu-parallel-command-line-power-tool.

20 Mark Wallace. Languages versus packages for constraint problem solving. In *Principles and Practice of Constraint Programming – CP 2003*, pages 37–52. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-45193-8_3.

21 Weixiong Zhang. Complete anytime beam search. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI)*, pages 425–430. AAAI Press, 1998. URL: http://www.aaai.org/Library/AAAI/1998/aaai98-060.php.

A DP Model for Bin Packing

Listing 5 The DP model for bin packing in RPID.

```
struct BinPacking { c: i32, w: Vec<i32> }
struct S { u: FixedBitSet, r: i32 }

impl Dp for BinPacking {
    type State = S;
    type CostType = i32;

    fn get_target(&self) -> Self::State {
        let mut u = FixedBitSet::with_capacity(self.w.len());
        u.insert_range(..);
        S { u, r: 0 }
    }
    fn get_successors(&self, s: &Self::State)
        -> impl IntoIterator<Item = (Self::State, Self::CostType, usize)> {
        let candidates = s.u.ones().filter(|&i| s.r >= self.w[i])
            .collect::<Vec<_>>();
        if candidates.is_empty() {
            let i = s.u.ones().next().unwrap();
            let mut next_u = s.u.clone();
            next_u.remove(i);
            vec![(S { u: next_u, r: self.c - self.w[i] }, 1, i)]
        } else {
            candidates.into_iter().map(|i| {
                let mut next_u = s.u.clone();
                next_u.remove(i);
                (S { u: next_u, r: s.r - self.w[i]}, 0, i)
            }).collect()
        }
    }
    fn get_base_cost(&self, s: &S) -> Option<Self::CostType> {
        if s.u.is_clear() { Some(0) } else { None }
    }
}
```

In bin packing, a set of items N is given, and each item $i \in N$ has the weight w_i. The objective is to minimize the number of bins to pack all items, where each bin has the capacity c. In our DP model, as state variables, we use the set of unpacked items U and the remaining capacity r of the current bin. Each transition packs item $i \in U$ in a bin. If i fits in the current bin, i.e., $w_i \leq r$, we can pack it in the current bin, and r is decreased by w_i. If $w_i > r$, we need to open a new bin to pack i, and r becomes $c - w_i$ in such a case. Without loss of optimality, we open a new bin only when no item can be packed in the current bin. In addition, any item can be selected as the first item packed in the newly opened bin.[12] We show the DP model in Equation (7) and its implementation with RPID in Listing 5.

$$V(U, r) = \begin{cases} 0 & \text{if } U = \emptyset \\ 1 + V(U \setminus \{i\}, c - w_i) & \text{if } \forall j \in U, w_j > r \wedge \exists i \in U \\ \min\limits_{i \in U : w_i \leq r} V(U \setminus \{i\}, r - w_i) & \text{otherwise.} \end{cases} \tag{7}$$

B Number of Optimally Solved Instances Against Time

We present the number of optimally solved instances against time for CVRP, MDKP, bin packing, SALBP-1, $1||\sum w_i T_i$, talent scheduling, MOSP, and talent scheduling.

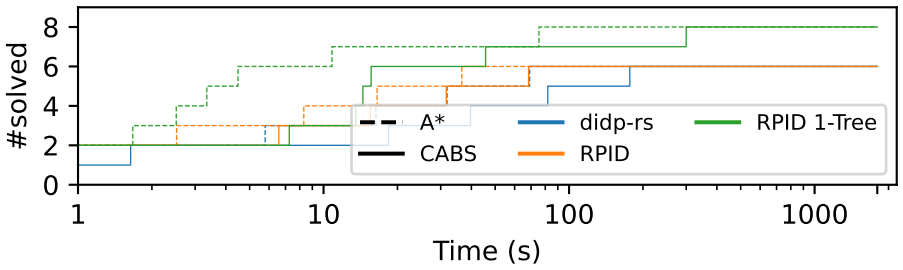


Figure 3 Time in seconds vs. the number of optimally solved instances for CVRP.

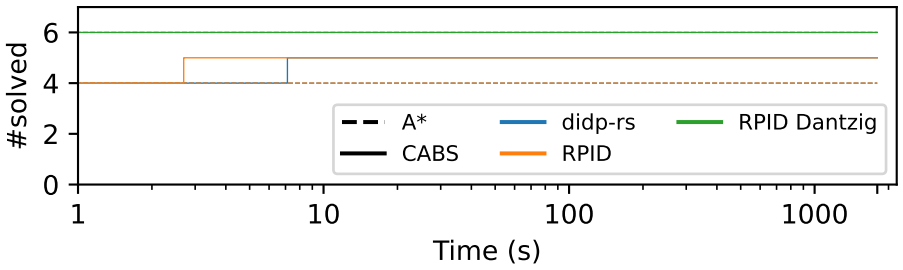


Figure 4 Time in seconds vs. the number of optimally solved instances for MDKP.

[12] In our DP model used in the experimental evaluation, we further break symmetries by enforcing that item i is packed in the i-th or earlier bin, following Kuroiwa and Beck [14, 17].

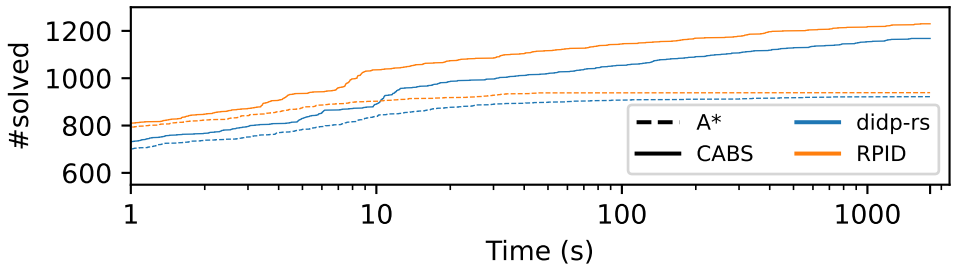


Figure 5 Time in seconds vs. the number of optimally solved instances for bin packing.

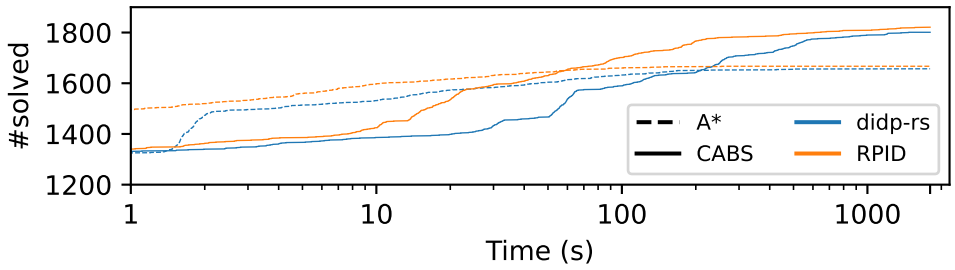


Figure 6 Time in seconds vs. the number of optimally solved instances for SALBP-1.

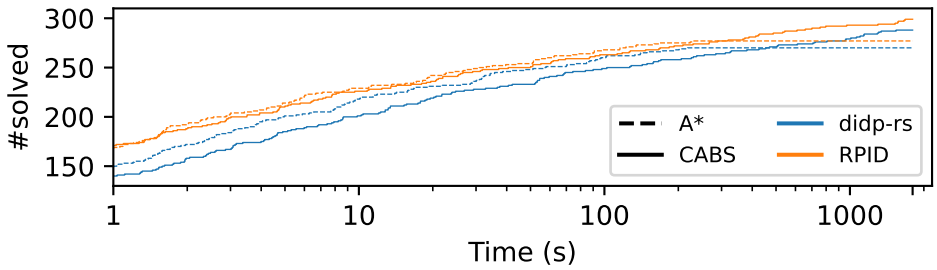


Figure 7 Time in seconds vs. the number of optimally solved instances for $1||\sum w_i T_i$.

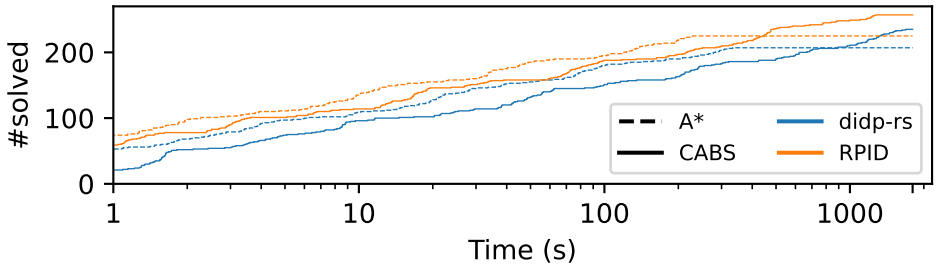


Figure 8 Time in seconds vs. the number of optimally solved instances for talent scheduling.

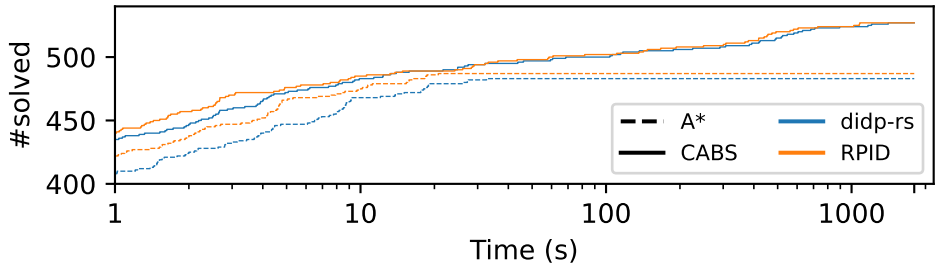


Figure 9 Time in seconds vs. the number of optimally solved instances for MOSP.

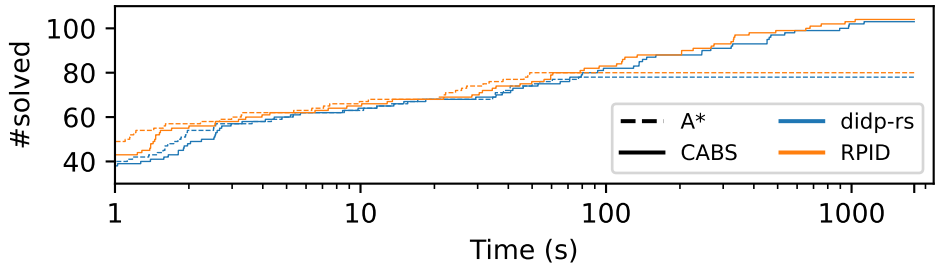


Figure 10 Time in seconds vs. the number of optimally solved instances for graph-clear.

The Work Task Variation Problem

Mikael Z. Lagerkvist[1]
SambaNova Systems, Stockholm, Sweden
Optischedule, Stockholm, Sweden

Magnus Rattfeldt[2]
Jeppesen, Gothenburg, Sweden
Optischedule, Stockholm, Sweden

Abstract

This paper introduces the Work Task Variation (WTV) problem, a novel scheduling post-processing challenge focused on improving worker shift quality by rearranging tasks within their assigned time slots. The objective is to avoid excessively short or long durations of specific task types, creating smoother and more ergonomic work patterns.

We present RosterLogic Variation, a constraint-based local search (CBLS) inspired solver originally developed at Optischedule and successfully deployed in real-world retail settings. This solver rapidly improves existing schedules using tailored invariants and heuristics. We also provide a complete MiniZinc model and a set of generated realistic publicly available benchmark instances.

We compare our solver's performance with that of modern CP solvers using the MiniZinc model. Contemporary state-of-the-art CP solvers are approaching the interactive performance of our CBLS solver for coarse planning, representing a significant advancement since the original design and implementation of our solver.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Planning and scheduling

Keywords and phrases Constraint-Based Local Search, Constraint Programming, Metaheuristics, Scheduling

Digital Object Identifier 10.4230/LIPIcs.CP.2025.24

Supplementary Material *Software (Source Code)*: `https://github.com/optischedule/work-task-variation-instances/tree/cp2025`
archived at `swh:1:dir:42f8c7d521bea4ed7fffcd5463d2b2c89353a236`

Acknowledgements We thank the anonymous reviewers for their helpful comments, Linnea Stjerna, and all those with whom we discussed the WTV problem.

1 Introduction

Employee scheduling is a common and significant challenge across various industries, with many studies focusing on generating feasible schedules while minimizing global metrics (e.g., [27, 23]). However, the quality of individual worker shifts – particularly concerning task variation and ergonomic factors – often remains a secondary consideration. This oversight can lead to worker fatigue, reduced productivity, regulation violations, and even potential safety risks.

This paper introduces the Work Task Variation (WTV) problem, addressing this gap by focusing on post-processing existing schedules. Given a schedule with fixed worker time slots and task assignments, the core challenge of the WTV problem lies in rearranging tasks

[1] Current affiliation is SambaNova Systems; work carried out while at Optischedule.
[2] Current affiliation is Jeppesen; work carried out while at Optischedule.

© Mikael Z. Lagerkvist and Magnus Rattfeldt;
licensed under Creative Commons License CC-BY 4.0

between workers while preserving the original time slot allocations to improve flow and ergonomics. The cost function penalizes both excessively short and long spans of a single task, aiming for a balanced variety minimizing repetitive strain and promotes engagement.

The WTV problem is particularly relevant when work-safety regulations limit continuous time on specific tasks, for example hand-intensive tasks common in retail. WTV provides an effective mechanism to ensure compliance by varying tasks within predefined schedules. It is especially useful when an initial schedule is generated automatically, but requires refinement to meet operational and regulatory needs.

Our approach, motivated by the need for fast and interactive schedule improvement, is a custom solver called RosterLogic Variation that is inspired by Constraint-Based Local Search (CBLS) [15, 10, 19]. Developed at Optischedule, this solver utilizes carefully chosen invariants for feasibility and efficient neighborhood exploration, combined with metaheuristics to guide the search process effectively. Importantly, this solver has been successfully deployed in practice, improving work schedules in multiple retail locations.

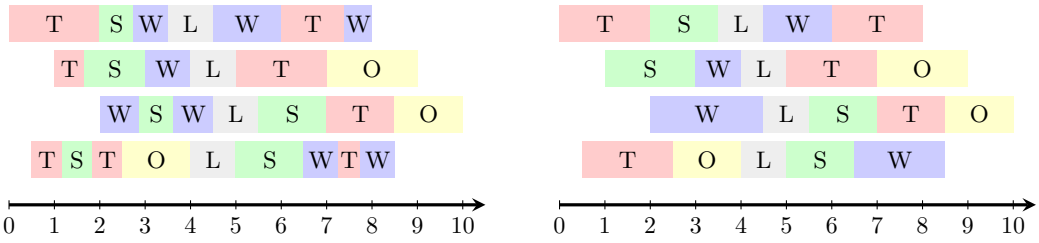
To facilitate further research and benchmarking within the constraint programming community, we have also developed a complete MiniZinc [18] model for the WTV problem. This model serves as a precise description of the problem and can, together with a set of generated publicly available problem instances, be used as a benchmarking problem for constraint programming tools and other solution approaches.

Contributions. This paper contributes: **(i)** defining the Work Task Variation (WTV) problem, a novel scheduling problem with practical implications for worker well-being; **(ii)** a description of RosterLogic Variation, a CBLS-inspired solver (developed at Optischedule and proven in practice) for the WTV problem, focused on rapid improvement of existing schedules; **(iii)** a complete MiniZinc model of the WTV problem for research and exploration with standard CP tools; and **(iv)** introducing a set of realistic, publicly available benchmark instances for evaluating solution approaches. Benchmarks of RosterLogic Variation against the MiniZinc model with different CP backend solvers shows that the custom solver performs well.

Structure. The remainder of the paper is structured as follows: Section 2 describes the WTV problem. In Section 3 the MiniZinc model is described to formally define the problem, and in Section 4 the custom CBLS-inspired solver is described. Section 5 describes the generated benchmark instances and presents the results comparing the custom solver with CP backend solvers. Finally, Section 6 concludes and summarizes the findings.

2 The Work Task Variation Problem

The Work Task Variation (WTV) problem is a scheduling optimization problem designed as a post-processing step to enhance pre-existing work schedules. Unlike traditional scheduling, which often focuses on creating shifts based on requirements (such as in Minimum Shift Design [6]) and then assigning workers to shifts and tasks [4], WTV assumes fixed worker-to-time-slot allocations and task quantities per time slot. The challenge is to rearrange task assignments within these time slots, optimizing task variation and distribution for each worker. For an example schedule with four shifts, see Figure 1 with one bad set of shifts **(a)** with lots of task-switching, and a corresponding good set of shifts **(b)** with reasonable task lengths.



(a) Fragmented Task Assignments. **(b)** Good Task Assignments.

Figure 1 Comparison of two different sets of shifts for the same work. The tasks are *till* (T), *store* (S), *warehouse* (W), *other* (O), and *lunch* (L). In **(a)** the work is very fragmented with frequent switching between shorter tasks. In **(b)** the tasks are longer and less fragmented.

The primary motivation is to improve worker well-being, reduce fatigue, and ensure compliance with work-safety regulations, especially in industries with repetitive or physically demanding tasks. Previous research has explored similar concerns, highlighting the need for ergonomic considerations in workforce scheduling [27, 23]. Key application areas for WTV include:

Retail Working at the till/cash register in a retail store is repetitive both physically and mentally.

Warehouses/Logistics Picking, packing, loading, and unloading in a warehouse is often repetitive and lead to physical stress.

Manufacturing Rotating workers between assembly line stations can reduce repetitive strain injuries.

In these scenarios, using a WTV solver that improves the ergonomic concerns and creates smooth operational schedules is beneficial for creating good schedules. On a process level, it is a refinement step, not a complete rescheduling.

The WTV optimization seeks an assignment of tasks to workers within each time slot that minimizes a cost function reflecting two complementary objectives directly derived from customer requirements: good lengths for task runs, and limiting the number of task initiations within each shift. This naturally relates to broader discussions of fairness and balancing in scheduling [2, 3, 11], as we aim to create a smooth task flow while respecting ergonomic principles. It is common to use WTV in cyclical schedules [1, 17], where fairness between shifts is less important as all workers will eventually work all types of shifts. For task run lengths, the cost function typically penalizes both excessively short and excessively long runs of a single task.

The ideal solution balances these, creating a *smooth* task flow with appropriate durations – neither too long nor too short. What is appropriate is defined by costs for duration ranges, with an ideal range incurring zero cost. In addition, any solution to the WTV problem must respect the following hard constraints from the initial schedules:

Fixed Time Slots Worker shift times and the schedule's time slot structure remain fixed.

Fixed Resource Requirements The number of workers required for each task in each time slot is predetermined. WTV rearranges task assignments among workers but preserves the total number of workers assigned to each task per slot.

Optional Fixed Assignments Specific task assignments for certain workers and time slots can be predefined and must be respected.

Single work periods Each resource only has a single work period (no split shifts).

Listing 1 Data Definitions.

```
enum Resources;
enum Activities;

enum ActivitiesOrNone = A(Activities) ++ { None };

int: slots;
set of int: Slots = 1..slots;
set of int: SlotsAndZero = 0..slots;

array[Activities, Slots] of 0..card(Resources): requirements;

array[Resources, Slots] of opt ActivitiesOrNone: fixed;

array[Activities, SlotsAndZero] of int: activity_run_cost;
array[Activities, SlotsAndZero] of int: activity_frequency_cost;

array[ActivitiesOrNone, SlotsAndZero] of int: extended_activity_run_cost =
    array2d(ActivitiesOrNone, SlotsAndZero,
        array1d(activity_run_cost) ++
        % Cost for None, which is 0 for all lengths
        [0 | r in SlotsAndZero]
);

array[ActivitiesOrNone, Slots] of 0..card(Resources): extended_requirements =
    array2d(ActivitiesOrNone, Slots,
        % Base requirements for all activities
        array1d(requirements) ++
        % The complement of the sum of requirements with respect to total resources
        [card(Resources) - sum(requirements[.., s]) | s in Slots]
);
```

It is important to note that WTV is designed as a post-processing step in a scheduling workflow. Some existing system or process is responsible for the initial schedule generation (using either automatic or manual scheduling), creating a schedule considering factors such as staffing levels, demand, competencies, rules and regulations, and availability. This initial schedule may not fully address ergonomics. Given the initial schedule, the WTV solver rearranges task assignments within time slots, optimizing for suitable variation by minimizing the associated cost. The improved schedule can be presented to a human scheduler for manual adjustments after which the WTV solver can be re-run iteratively.

This approach integrates WTV seamlessly into existing processes, improving schedule quality without a system overhaul. It offers a targeted solution for worker well-being and efficiency.

As a simplification, we do not consider competencies for workers for the WTV problem in this paper. While in full generality competencies might be needed, in most cases we have seen the sets of tasks that are involved in a single WTV optimization instance typically require a uniform competency level among all involved workers.

3 A MiniZinc Model

The problem definition of Work Task Variation has a natural encoding into a MiniZinc model. The fact that schedules are typically defined in blocks that are synchronized across all shifts makes for a model that operates on an abstract set of discretized slots (a grid), without regard to real time spans. This model formally defines the problem's constraints and objective function, enabling experimentation with different constraint solvers and providing a means to validate solutions obtained via other methods.

The data-definitions are shown in Listing 1. Each schedule is defined over the `Resources` (the workers), the `Activities` (the tasks) with `None` representing time outside a shift, and finally the `Slots` to schedule in. For each slot there is a set of `requirements` for the number

Listing 2 Variable Definitions.

```
% The schedule: what activities are done when for each resource
array[Resources, Slots] of var ActivitiesOrNone: schedule;

% Markers for when runs end
array[Resources, Slots] of var bool: run_end;

% Length for each run at the current slot from the currents runs start
array[Resources, Slots] of var SlotsAndZero: run_length;

% Cost for each run at the end of a run with zero cost in the middle of runs
array[Resources, Slots] of var int: run_cost;

% Cost for number of runs of each activity per resource
array[Resources, Activities] of var int: frequency_cost;

% The total cost of runs
var int: cost = sum(run_cost) + sum(frequency_cost);
```

Listing 3 Constraints for Structure.

```
% All shifts are only Activities (that is, not None) and surrounded with None
constraint forall (r in Resources) (
    regular(schedule[r, ..], "None* [^None]* None*")
);

% Always respect the requirements for each slot (column in the schedule)
constraint forall (s in Slots) (
    global_cardinality(schedule[.., s], ActivitiesOrNone, extended_requirements[.., s])
);

% Always respect the fixed requirements
constraint forall (r in Resources, s in Slots where occurs(fixed[r, s])) (
    schedule[r, s] = deopt(fixed[r, s])
);

% Mark when runs end
constraint forall (r in Resources, s in Slots) (
    if s = slots then
        run_end[r, s] = true
    else
        run_end[r, s] = (schedule[r, s] != schedule[r, s+1])
    endif
);

% Count length of runs
constraint forall (r in Resources, s in Slots) (
    if s = 1 \/ run_end[r, s-1] then
        run_length[r, s] = 1
    else
        run_length[r, s] = run_length[r, s-1] + 1
    endif
);
```

of activities of that type needed for that slot. There are also a set of `fixed` activities for each resource. The cost is defined as the cost associated with each possible run length and frequency count for every activity.

The run costs and the requirements are extended using MiniZinc to work over activities and the `None` value, as that enables simpler expressions for calculating costs and enforcing requirements.

In Listing 2 the variables for the problem are defined. The main variable is the `schedule` matrix that contains the actual schedule. The remaining variables are all for computing the cost of the schedule. The costs are defined over maximal *runs* of adjacent slots in the schedule with the same activity. We use the end of a run in `run_end` as the marker for when a run is finished. The `run_length` is the length of each run *up to that point*. The `run_cost` are the costs for each run based on their length, and is only non-zero at the end of runs. The `frequency_cost` is based on the number of occurrences of runs for each activity type for each resource. Finally, the `cost` is simply the sum of the run costs and the frequency costs.

Listing 4 Constraints for Calculating Costs.

```
% Count run costs
constraint forall (r in Resources, s in Slots) (
    if run_end[r, s] then
        run_cost[r, s] = extended_activity_run_cost[schedule[r, s], run_length[r, s]]
    else
        run_cost[r, s] = 0
    endif
);

%Count frequency costs
constraint forall (r in Resources, a in Activities) (
    let {
        var int: activity_run_count = count(s in Slots) (
            run_end[r, s] /\ schedule[r, s] = A(a)
        )
    } in
    frequency_cost[r, a] = activity_frequency_cost[a, activity_run_count]
);
```

Listing 3 contains the main constraints for the problem. First, `regular` constraints are used to make sure that each shift has the expected format (`None` at start and end, and `Activities` between). The `global_cardinality` constraints ensure that for each time slot, the required number of activities (and no more) is allocated. After that, all fixed requirements in the instance are set using simple assignments. The `run_end` variables are connected to the schedule using simple reified not equal constraints. Relatedly, the `run_length` is a simple increasing sequence that is reset to 1 after a run ends.

Finally, in Listing 4, the actual cost of the schedule is computed. The run costs are fetched at the end of a run using the corresponding run length. The frequency costs are computed by counting the number of runs of each activity type.

Putting it all together. Listings 1 to 4 collectively define all the required parts to solve the WTV problem. The only missing parts are inclusion of the `globals.mzn` library and a `solve minimize cost`. Output statements can be added as needed. For the benchmarks experiments reported in Section 5, no specific MiniZinc search strategy was specified, and the output was limited to the objective cost value.

4 RosterLogic Variation: A Local Search Solver for WTV

To address the Work Task Variation (WTV) problem, we have developed a constraint-based local search (CBLS) [16] solver designed for interactive use within a real-world scheduling process called *RosterLogic Variation* [20]. Local search [12] is a natural fit for problems that involve improving an existing solution. Our approach prioritizes rapid solution improvement over finding provably optimal solutions, making it well-suited for integration with a user-facing scheduling tool, where quick feedback and iterative refinement are paramount. The solver has been successfully applied in operational retail scheduling environments.

The core of our approach lies in a carefully designed combination of: a compact and efficiently manipulated state representation, a flexible and extensible neighborhood exploration strategy, a configurable cost model, and a portfolio of local search algorithms including metaheuristics. These components are interconnected to find high-quality WTV solutions in limited time.

When we set out to solve the WTV problem in 2018, our experiments with using traditional CP directly did not indicate that we would get suitable solutions quickly enough. In addition, given that we had usable initial solutions, a local search approach was a natural fit. Our prototype for the RosterLogic Variation solver quickly showed us that the custom CBLS style solver approach was feasible.

Table 1 Example of a schedule and the corresponding run-length encoding.

Slot	1	2	3	4	5	6
Shift 1	1	1	1	2	2	0
Run Lengths	3	−1	−2	2	−1	1
Shift 2	0	0	2	2	2	2
Run Lengths	2	−1	4	−1	−2	−3

4.1 State Representation and Invariants

The key design principle for the state of a local search solver is to enable efficient incremental evaluation of solution quality and move feasibility. To achieve this, we maintain a set of invariants [15, 25] that are dynamically updated during search in addition to the core schedule. We also use simulation [19] when possible to speed up move evaluation further.

Our solver represents a complete schedule (a solution) as an assignment of activity types to each worker (resource) for each time slot within a given planning horizon.

The primary state representation is a matrix where rows represent workers and columns represent time slots. Each cell in this matrix contains an activity assignment.

Run-length encoding. A crucial optimization is the use of an auxiliary run-length encoding state to represent consecutive assignments of the same activity. At the start of each run of activities (a sequence of consecutive identical activities) the length of the run is maintained. At each interior position the number of steps to go backwards to find the start of the run instead. This structure allows for O(1) access to run lengths and boundaries from each activity position, significantly accelerating cost calculations related to activity durations.

Consider the example illustrated in Table 1. The table shows a sample schedule using 0 for no work and 1 and 2 for two different task types, alongside the corresponding run-lengths. Positive values in the "Run Lengths" rows indicate the start of a run and the length of that run. Negative values indicate positions within a run, showing the offset to the run's start.

Activity counts. We maintain aggregate counts of the number of occurrences of each activity type within each time slot; this allows for efficient verification of constraints related to the overall distribution of activities across the work force at any given time.

We maintain counts of the number of times each activity is assigned to each worker; this is essential for evaluating costs and constraints related to the frequency of activities for individual workers.

4.2 Cost Model and Constraints

The objective function, or cost, of a schedule is defined by a configurable cost model. The cost model for the WTV includes both duration costs and frequency costs for runs of tasks in each shift. The run-length encodings and the activity counts described in the previous subsection are the core invariants used to calculate these costs.

The cost model is designed to be extensible. We support both a hard-coded cost model with predefined rules and a configurable model that loads parameters from a JSON configuration file. This allows users to tailor the cost function to their specific needs without modifying the solver's core logic. The JSON-based configuration provides a high degree of flexibility, enabling users to define costs for different activity durations using range mappings.

4.3 Neighborhood Structure and Move Generation

The search space is explored through a neighborhood structure defined by a set of move generators. These generators produce moves that transform one schedule into a neighboring schedule while maintaining the invariants for the run lengths and activity counts. We have designed several move generators, each capturing different types of schedule modifications relevant to the WTV problem:

Local swaps The most basic move generator produces swaps between workers for the same time slot. Swaps can have configurable durations, allowing the exchange of sequences of activities. This generator can target specific activity types, e.g., a swap generator that swaps a *till* and *store* activity in the same slot.

Pattern-based moves A more sophisticated generator searches for predefined patterns of activities that can be exchanged to improve the schedule's structure. This allows for larger, non-local changes to the schedule. See below for more details.

Constrained move generation Move generators can be configured with checks for valid moves. This is achieved through a filtering mechanism that takes a base move generator and a predicate defining allowed moves. For instance, we can enforce constraints that prevent assigning certain activities to workers who are not qualified for those tasks, or that limit the number of times a particular activity can be assigned within a certain time window. This flexible constraint integration allows for incorporating domain-specific knowledge and hard constraints directly into the search process.

The move generators produce instances of a generic interface for a set of moves, which provides methods for applying the move to a schedule, computing the resulting cost difference (without applying the move first), and querying the move's properties. This abstraction allows different move types to be handled uniformly by the search algorithms. The base moves collection for local swaps uses a compact representation using a single array of integers in order to use fewer allocations and keep memory locality high.

Pattern-based move generation. This is a move generator (which turned out to be crucial for achieving good performance) where swaps are defined by high-level activity patterns, in order to achieve larger and non-local changes to the schedule. A pattern is defined as a sequence of activity types – such as `SSSTT` for three *store* activities followed by two *till* activities – representing a desirable target activity sequence. Note that these patterns are strongly connected to the duration cost part of the used cost model.

The pattern-based move generation identifies local sequences of activities in the schedule that can be transformed into one of the desired patterns, by performing a set of vertical swaps between shifts at the same time slots. The following is a sketch of the pattern-based move generation process.

Candidate matching For each pattern (and its reverse if it is different), the schedule is scanned to identify sequences of matching lengths that consist only of *store* or *till* shifts. These sequences do not have to (and should not) match the pattern exactly but must be transformable into it.

Mismatch analysis For each such candidate (mis)match, the schedule is scanned vertically – that is, within the same time slots across other shifts – for sequences in other shifts that match the required activity sequence. This step ensures that each non-matching element has a possible replacement available.

Slot	1	2	3	4	5	6	7	8
Shift 0	S	T	S	T	S	O	O	O
Shift 1	S	S	T	S	T	T	T	T
Shift 2	T	S	S	S	S	O	O	O
Shift 3	T	T	T	S	T	S	S	S

(a) Initial Schedule.

Slot	1	2	3	4	5	6	7	8
Shift 0	S	**S**	S	T	**T**	O	O	O
Shift 1	S	S	T	S	T	T	T	T
Shift 2	T	**T**	S	S	S	O	O	O
Shift 3	T	T	T	S	**S**	S	S	S

(b) Schedule After Pattern Swaps.

Figure 2 Example `SSSTT` pattern swap operation. In **(a)** the initial schedule is shown, and in **(b)** the resulting schedule after the pattern swap operation is shown (cells whose values changed are printed in **bold** to indicate the modification.).

Swap generation Once all required replacements are identified, all viable combinations of vertical swaps that could realize the pattern are generated. These combinations are then ranked according to **(i)** highest cost decrease and **(ii)** the lowest total vertical distance traveled by the swaps (tiebreaker). The most promising combinations are selected to generate actual swap candidates.

Patterns are extended with respect to the block size, and grouped by length for efficient parallel processing. To reduce redundancy, only one direction of each symmetric pattern (e.g., `SSSTT` and `TTTSS`) is stored, and its flipped version is automatically included. See Appendix C for a list of all patterns defined in RosterLogic Variation.

Example. Consider the desired pattern `SSSTT`. Figure 2 illustrates the process using a segment of a roster. The initial state is shown in Figure 2a.

The algorithm identifies the sequence `STSTS` in Shift 0 (Figure 2a), starting at slot 1, as a candidate. It determines that slots 2 and 5 must be replaced to match the desired pattern. Matching activities `S` and `T` are located vertically in shifts 2 and 3, respectively. Swapping these values results in the updated roster in Figure 2b. This introduces the desired pattern `SSSTT` in Shift 0, while also strengthening the contiguous blocks of *till* shifts in Shift 2 and *store* shifts in Shift 3.

4.4 Search Algorithms

Our solver implements a portfolio of local search algorithms, enabling experimentation and selection of the most effective approach for a given problem family.

Steepest descent A greedy algorithm that always selects the best improving move in the neighborhood.

Tabu search A metaheuristic that maintains a *"tabu list"* of recently visited states to avoid cycling and encourage exploration of the search space [8, 9].

Simulated annealing A probabilistic metaheuristic that accepts worsening moves with a probability that decreases over time, inspired by the physical process of annealing [13].

Restart A restarting metaheuristic that runs a base algorithm multiple times on scrambled states returning the best solution found.

Portfolio A metaheuristic that wraps several different algorithms or configurations of the same algorithm. All algorithms are called, and the best result is returned.

Parallel restart and portfolio To leverage multi-core processors, we implement parallel variants of both the restart and portfolio metaheuristics.

Scramble A randomized algorithm that makes a set of random moves. This is useful in combination with restart and parallel portfolio in order to explore new neighborhoods.

These algorithms can be combined and configured. For instance, a typical configuration might involve a parallel portfolio of steepest descent, tabu search, and simulated annealing. This allows for a highly adaptable and robust system.

4.5 Algorithm Configuration

RosterLogic Variation has a default configuration of the algorithm components that is optimized for the retail interactive use cases. The system builds on a core search algorithm configuration that has a parallel portfolio consisting of a greedy steepest descent asset, three different tabu search assets, a simulated annealing asset, and a scrambling asset. The parallel portfolio is furthermore wrapped in a restart algorithm with four restarts. Building on this base algorithm, the final algorithm has the following 5 stages.

1. Apply base algorithm with moves for swapping *till* and *store* activities.
2. Apply base algorithm with moves for swapping *warehouse* activities with *till* and *store* activities.
3. Apply a greedy steepest descent algorithm for swapping *till* and *store* activities.
4. Apply a greedy steepest descent algorithm with a pattern moves generator with 10 common patterns for improving *till* and *store* layouts.
5. Repeat the greedy algorithms in 3 and 4 again.

As is obvious from this description, the specific configuration is heavily geared towards a problem instance space where the *till* activities are a significantly limiting factor. However, this is quite a common problem setting for the WTV problem, with a specific activity that is the most important to focus on,

Similarly, the fact that only a single step (of a composite algorithm) actually includes *warehouse* tasks is because these are often quite forgiving and easy to parcel out, especially after an initial pass has been made that gives a decent set of *till* and *store* activities.

4.6 Development and Pragmatics

The RosterLogic Variation solver has been developed over the course of several years starting in 2018 with most of the development in 2019. The system has been used in practice by expert planners, mainly for retail planning.

Development environment. The RosterLogic Variation solver is implemented in Java and Kotlin for the JVM platform. The state is maintained as a set of compact arrays with suitably chosen types to minimize the memory usage, with most arrays either using bytes or shorts. This type of state management makes the total memory footprint low, ensuring good cache behavior even though the code is written in a pointer-based, garbage collected language like Java.

Development of the solver has been on Linux and Mac, and the application has been packaged as a self-contained executable for the Windows platform as that is the platform used by most planners. For the expert planners that we have worked with there is a willingness to use command-line applications with structured input and output formats. Thus, we could focus on the core algorithmic challenges and avoid too much focus on user interface challenges.

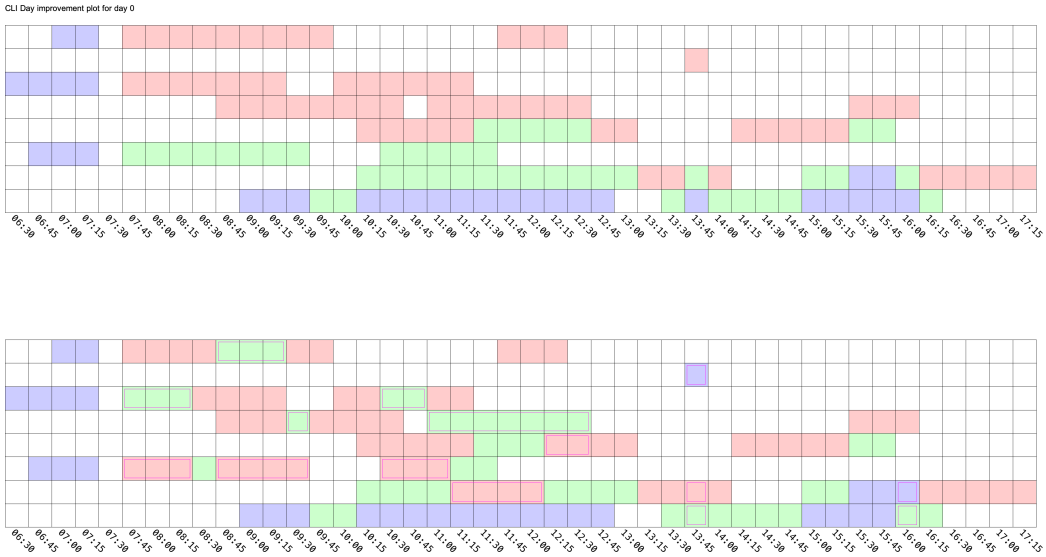


Figure 3 The RosterLogic Variation debug plot output when solving the length 12, workers 8, blocks size 15, seed 3 instance. The plot shows only the blocks with *till*, *store*, and *warehouse* activities.

Repeatability and feedback. All algorithms use a controlled stable source of randomness for repeatability. Newer parts of the framework at Optischedule use the xorshift* [26], which has better properties for maintaining and modifying the state and to create parallel streams of randomness.

All algorithms use a structured progress reporting system to indicate how far along in the search they are. A progress reporter instance can be split with adjustable amounts for each part. For example, a Portfolio algorithm with 5 different sub-algorithms would split it's progress reporter into 5 equal parts. A base algorithm can report either a percentage done, or the number of steps out of a configurable step count. All together, the progress reporter is a best-effort approximation, but it is very useful for user to see a rough estimate of how far along the system is.

A simple plotting system gives visual feedback for both planners and developers. Having a clear visualization of how changes are made to the schedule is critical for effective development. An example of the plot is shown in Figure 3.

5 Experiments

To evaluate the RosterLogic Variation solver and compare and contrast it with the results for the MiniZinc model, a set of benchmark instances is needed. Since the data from our customers is confidential, we have created a set of artificial problems with similar characteristics. Below we will describe the generated instances, the different solvers we benchmark against, and finally discuss the actual results.

5.1 Generated Instances

We have written an instance generator for WTV that mimics real-world instances that we have worked with, without disclosing any sensitive customer data. The domain for the instances is the retail case, where WTV is applied to *till*, *store*, and *warehouse* tasks. In particular, working at the *till* is the limiting factor, with stricter costs for the right task lengths and a significant cost for each worker that works too long or too often at the *till*.

The set of instances are generated with a fixed block length of either 15 or 5 minutes, which are both common planning resolutions in real-world settings. Naturally, using a 5 minute block size gives larger and harder instances than 15 minute block sizes for the same length of the schedule. The schedule has a length of 10 to 16 hours in steps of 2 hours. Within these times, 8 to 14 hours are designated as the store opening time, within which all the *till* and *store* tasks are allocated. The *warehouse* task (and other tasks) may extend outside this time.

The number of worker shifts varies between 8 and 16. Workers shifts are randomized over and around the start time of the store, and the three main tasks are interspersed with other work (that is fixed in time and place) and breaks. Workers have lunch breaks between 30 and 60 minutes long, and sometimes a morning and/or afternoon coffee break of 15 minutes, for long enough shifts.

For each combination, 10 different instances have been generated to give a good variety of instances, for a total of $2 * 4 * 5 * 10 = 400$ instances. The generated instances are available at `https://github.com/optischedule/work-task-variation-instances` including the full MiniZinc model. For brevity of exposition, only 4 instances per size are tested (so 160 different instances in 40 different sizes).

The generated instances represent typical sizes for normal workloads. For larger sites, it is common to schedule personnel in groups (often divided based on competencies), leading to smaller groups of workers to exchange tasks between.

5.2 Solvers and System

We benchmark our RosterLogic Variation solver against 8 different variants of OR-Tools CP-SAT [22] and Gecode [7], using the MiniZinc formulation. In total we ran $1 + 8$ different variations of the following solvers:

RosterLogic Variation RosterLogic Variation is run in its default configuration described in Section 4.5.

OR-Tools CP-SAT 9.12 The current release of CP-SAT and available in MiniZinc 2.9.2.

OR-Tools CP-SAT 9.3 A release of CP-SAT from 3 years ago (March 2022).

Gecode 6.3.0 The standard Gecode solver supplied with MiniZinc 2.9.2.

Gecode 6.3.0, with restarts The standard Gecode solver supplied with MiniZinc 2.9.2, with settings to enable restarts (`--restart luby`, restarts using the Luby sequence [14]), no-good recording (`--nogoods`), and an increased copying distance (`--c-d 50`, peak depth is over 100, and with restarts early copies are wasteful).

All tests are limited to a maximum of 60 seconds run-time. The common use-case for solving the WTV problem is in an interactive setting, where the planner expects answers in a short timeframe. We consider 60 seconds to be on the outside of what would be accepted in the intended context.

All tests are run on a MacBook Pro M1 Max (10 cores) with 64 GiB of memory. RosterLogic Variation is naturally parallelized, and uses up to 5 cores at a time in its default configuration. Worth noting is that the default configuration was set in 2019, and that the common use-case was to use it on computers with 4-6 cores. The framework supports using more parallelism, but we have not tested this. All CP solvers are tested both single-threaded (1t) and multi-threaded with 10 threads (10t). Using a multi-threaded configuration is the intended usage of CP-SAT. The single-threaded runs for CP-SAT 9.12 are mostly shown for completeness and the ones for CP-SAT 9.3 are skipped as it did not show any interesting differences across versions. Full results are available on request from the authors.

Table 2 Number of instances where the no solution is found out of 80 tested instances for each block size.

Slot length	CP-SAT 9.12 1t	CP-SAT 9.12 10t	CP-SAT 9.3 10t
5	40	36	40
15	40	13	35

For timing of RosterLogic Variation, note that the system is written in Java, and while it can be used as a stand-alone tool, the intended deployment scenario is as a long-running server. To make the stand-alone tool for short-running processes faster, static compilation of the system with, for example, GraalVM [21] could be applied.

5.3 Results

In this section the results of the experiments are shown, more detailed results are given in Appendix A.

No solution. In many cases, CP-SAT fails to find any solution at all. In Table 2 the number of cases of no solution is shown for all the configurations tested. Note that all variations of Gecode found a solution, although generally not a good one. Since the problem is not inherently hard to solve if the right variables are branched on (the `schedule` variables in Listing 2), we believe that CP-SAT tries to focus on the extra variables used for cost calculations instead. Adding an explicit search annotation did not help in our experiments.

Plots. All costs in the plots are given normalized to the cost produced by RosterLogic Variation. This is because the objective cost values are not meaningful in themselves and vary widely in scale between different instances. As RosterLogic Variation has been used effectively in production, we can view it as a base-line level of *"good enough"*. A value lower than 1 for the cost (below the vertical line at 1) means that a better solution than RosterLogic Variation was found (e.g., 0.5 means a solution twice as good was found), while a value higher than one means a worse solution was found (e.g., 2.5 means the solution found was two and a half times worse than RosterLogic Variation's solution).

Similarly, the time is given as a normalized multiple compared with RosterLogic Variation, as that gives an indication of the complexity of the problem and is suitable for comparison in collected plots. For block size 15, run times for RosterLogic Variation in its default configuration is around 1.8 to 7.4 seconds, and for block size 5 it is between 3.7 and 16.2 seconds. A value below 1 (to the left of the vertical line at 1) means that solution was found faster than RosterLogic Variation, while conversely a value to the right of the line means the solution was found after RosterLogic Variation was done.

The plots use a base 2 logarithmic scale, and are intended to show the typical behavior of the solvers. We include all intermediate solutions reported through MiniZinc in the plots, as the any-time behavior of the solvers is of general interest for an interactive tool. RosterLogic Variation is not configured to give intermediate solutions currently, but could be retrofitted to do that. We omit the plot for Gecode-R with 1 thread for space reasons, as it showed no significant difference in behavior compare with Gecode-R with 10 threads.

Block size 15. In Figure 4 results for instances with block size 15 are reported. As can be seen there are very few markers that are both faster and have better results than RosterLogic Variation. CP-SAT in the multi-threaded configuration often reaches better results than RosterLogic Variation, although it takes more time to do so.

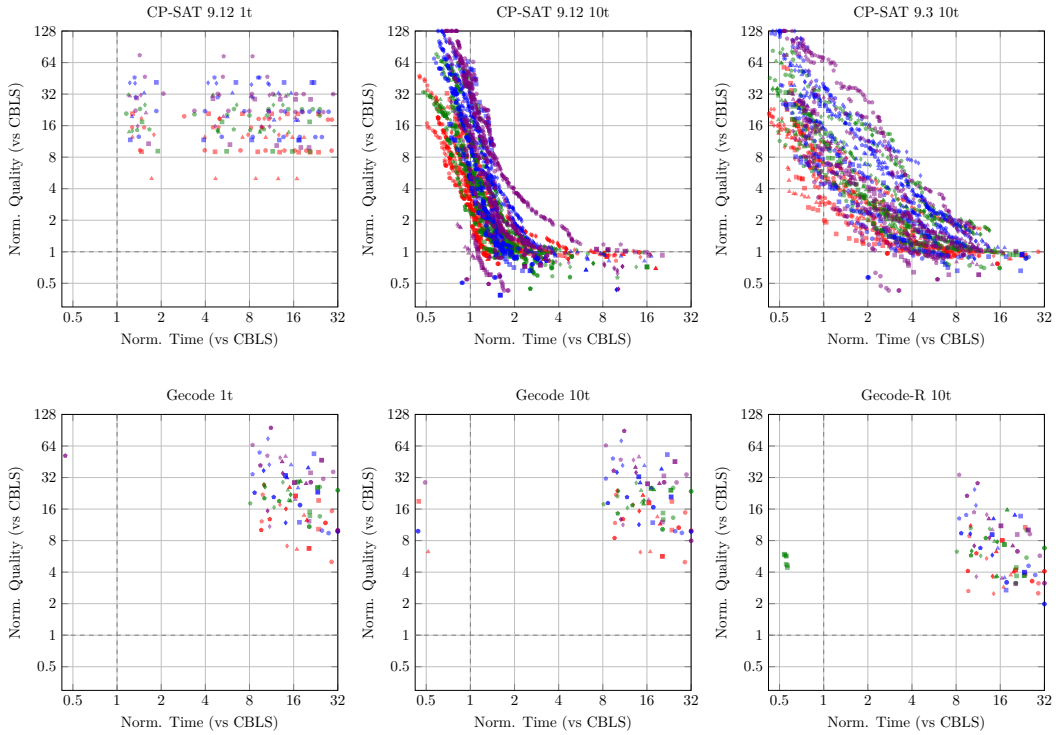


Figure 4 Normalized Quality vs. Time (vs RosterLogic Variation) for block size 15. Lower values are better. Dashed lines at 1.0 mark CBLS performance. Shapes denote number of workers W8 ●, W10 ■, W12 ▲, W14 ♦, and W16 as ⬟. Colors denote length of schedule: L10 ●, L12 ●, L14 ●, and L16 as ●.

CP-SAT in the single-threaded configuration gets significantly worse results than in the multi-threaded configuration. This is not surprising, as the single-threaded configuration does not include the portfolio assets that use LNS [24] and local search [5] that find improved solutions.

From the difference in the tables for CP-SAT 9.12 and 9.3 for 10 threads, it is obvious that CP-SAT is much faster at finding good solutions now than it was three years ago, as the collected point cloud has shifted to the left in the plot.

Gecode with the normal settings does not gain much from multiple threads (except a few cases where a solution was found very quickly), as it is used to help a search that is mostly weighted towards DFS exploration. Gecode with restarts produces a lot better solutions, and here the multi-threading quite often helps, although not always (results for the single-threaded Gecode-R is shown in Appendix A).

Block size 5. In Figure 5 results for instances with block size 5 are reported. With block size 5, the same length of schedule has three times as many variables. This naturally means that the run-times are also longer, although the limits for what is an acceptable wait time for an interactive use-case is still the same. Note that this means that the scale on run-time has changed compared with the plot for block size 15.

The big difference in results for block size 5 is that RosterLogic Variation also wins most cases on cost, except for a few of the ones with a lower number of workers that CP-SAT 9.12 with 10 threads finds better solutions for.

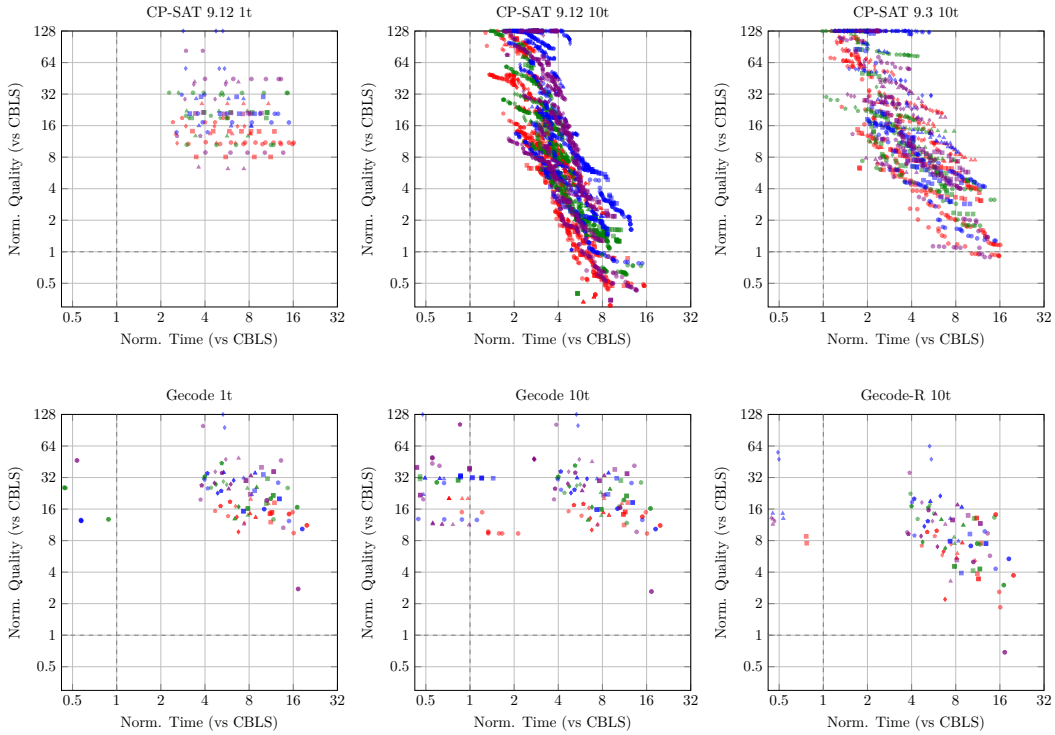


Figure 5 Normalized Quality vs. Time (vs RosterLogic Variation) for block size 5. Lower values are better. Dashed lines at 1.0 mark CBLS performance. Shapes denote number of workers W8 ●, W10 ■, W12 ▲, W14 ♦, and W16 as ⬟. Colors denote length of schedule: L10 ●, L12 ●, L14 ●, and L16 as ●.

5.4 Solver Choice for the WTV Problem

It is quite clear in the previous results that with a modern solver (Google OR-Tools CP-SAT 9.12) using multiple threads, investing the resources into creating a custom solver like RosterLogic Variation is likely not justified, if the end goal is to solve instances with 15 minute blocks. This assumes that the issue with sometimes not finding solutions can be solved. However, for the common case of scheduling in 5 minute blocks, RosterLogic Variation still outperforms modern CP variants. The cases where no solution is produced in 60 seconds is also concerning, but something that we believe is not a fundamental problem as shown by the Gecode solver always finding solutions.

It is also clear from the difference between CP-SAT versions 9.12 and 9.3 that development of automatic heuristics for solvers has progressed a lot in just a few years, and the recommendations may be very different in just a few years from now.

6 Conclusion

We have introduced the Work Task Variation (WTV) problem as a post-processing optimization challenge, particularly useful in personnel scheduling. Our work relates to existing research in ergonomic workforce scheduling [27, 23] and fairness trade-offs in resource allocation [2, 3, 11]. Although the WTV problem is understudied and lacks system support in many scheduling domains, it is often very important for creating ergonomic work shifts.

We presented a custom CBLS-inspired solver, RosterLogic Variation, designed for fast, interactive schedule improvement. RosterLogic Variation provides rapid feedback suitable for iterative refinement in practical settings. We hope that our description of the solver can be used as inspiration for others that wish to develop a custom CBLS inspired solver. The complete MiniZinc model developed for the problem enables comparison using standard CP solvers and facilitates further research. Together with the provided benchmark instances, these contributions allow researchers to investigate the WTV problem and develop more efficient solution methods.

Our experiments show that the custom solver, RosterLogic Variation, provides excellent performance for interactive use cases, rapidly finding good quality solutions. Furthermore, it demonstrates superior solution quality compared to current standard solvers on instances with finer-grained time resolutions (5-minute blocks).

References

1 Kenneth R. Baker. Workforce allocation in cyclical scheduling problems: A survey. *Journal of the Operational Research Society*, 27(1):155–167, 1976. doi:10.1057/jors.1976.30.

2 Dimitris Bertsimas, Vivek F. Farias, and Nikolaos Trichakis. The price of fairness. *Operations Research*, 59(1):17–31, 2011. doi:10.1287/opre.1100.0869.

3 Dimitris Bertsimas, Vivek F. Farias, and Nikolaos Trichakis. On the efficiency-fairness trade-off. *Management Science*, 58(12):2234–2250, 2012. doi:10.1287/mnsc.1120.1554.

4 Edmund Burke, Patrick De Causmaecker, Greet Vanden Berghe, and Hendrik Van Landeghem. The state of the art of nurse rostering. *J. Scheduling*, 7:441–499, November 2004. doi:10.1023/B:JOSH.0000046076.75950.0b.

5 Toby O. Davies, Frédéric Didier, and Laurent Perron. Violationls: Constraint-based local search in cp-sat. In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 13958 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 2024. doi:10.1007/978-3-031-60597-0_16.

6 Luca Di Gaspero, Johannes Gaertner, Guy Kortsarz, Nysret Musliu, Andrea Schaerf, and Wolfgang Slany. The minimum shift design problem. *Annals of Operations Research*, 155:79–105, August 2007. doi:10.1007/s10479-007-0221-1.

7 Gecode Team. Gecode: Generic Constraint Development Environment. URL: https://www.gecode.org/.

8 Fred Glover. Tabu search – Part I. *ORSA Journal on Computing*, 1(3), 1989. doi:10.1287/IJOC.1.3.190.

9 Fred Glover. Tabu search – Part II. *ORSA Journal on Computing*, 2(1), 1990. doi:10.1287/IJOC.2.1.4.

10 Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2009.

11 John N. Hooker and H. Paul Williams. Combining equity and utilitarianism in a mathematical programming model. *Management Science*, 58(9):1682–1693, 2012. doi:10.1287/mnsc.1110.1510.

12 Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Elsevier, Amsterdam, The Netherlands, 2004.

13 Scott Kirkpatrick, C. Daniel Gelatt, Jr., and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983. doi:10.1126/SCIENCE.220.4598.671.

14 Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993. doi:10.1016/0020-0190(93)90029-9.

15 Laurent Michel and Pascal Van Hentenryck. Localizer. *Constraints*, 5(1):43–84, 2000. doi:10.1023/A:1009990419456.

16 Laurent Michel and Pascal Van Hentenryck. Constraint-based local search. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, pages 223–260. Elsevier, Amsterdam, The Netherlands, 2018. doi:10.1016/B978-0-444-52726-4.50010-6.

17 Nysret Musliu, Andreas Schutt, and Peter J. Stuckey. Solver independent rotating workforce scheduling. In Willem-Jan van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 429–445, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-93031-2_31.

18 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard cp modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-74970-7_38.

19 M. A. Hakim Newton, Duc Nghia Pham, Abdul Sattar, and Michael Maher. Kangaroo: An efficient constraint-based local search system using lazy propagation. In *Principles and Practice of Constraint Programming*, volume 6876 of *Lecture Notes in Computer Science*, pages 645–659, Perugia, Italy, 2011. Springer. doi:10.1007/978-3-642-23786-7_48.

20 Optischedule AB. RosterLogic Variation, 2019.

21 Oracle Labs. GraalVM: Run Programs Faster Anywhere. https://www.graalvm.org/. Accessed: 21st July 2025.

22 Laurent Perron and Frédéric Didier. CP-SAT. URL: https://developers.google.com/optimization/cp/cp_solver/.

23 S. S. Sana, H. Ospina-Mateus, F. G. Arrieta, and J. A. Chedid. Application of genetic algorithm to job scheduling under ergonomic constraints in manufacturing industry. *Journal of Ambient Intelligence and Humanized Computing*, 10:2063–2090, 2019. doi:10.1007/S12652-018-0814-3.

24 Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-François Puget, editors, *Principles and Practice of Constraint Programming – CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 1998. doi:10.1007/3-540-49481-2_30.

25 Pascal Van Hentenryck and Laurent Michel. Differentiable invariants. In Frédéric Benhamou, editor, *Proceedings of CP'06*, volume 4204 of *LNCS*, pages 604–619. Springer-Verlag, 2006. doi:10.1007/11889205_43.

26 Sebastiano Vigna. Further scramblings of Marsaglia's xorshift generators. *Journal of Computational and Applied Mathematics*, 315:175–181, 2017. doi:10.1016/j.cam.2016.11.006.

27 Thitinan Wongwien and Suebsak Nanthavanij. Priority-based ergonomic workforce scheduling for industrial workers performing hazardous jobs. *Journal of Industrial and Production Engineering*, 34(1):52–60, 2017.

A Detailed results

This appendix gives more detailed results compared with the plots in Section 5.3. Similar to the plots, the costs are shown normalized to the cost produced by RosterLogic Variation. The tables show the best solution found, and if no solution was found within 60 seconds then a "-" is shown. In contrast to the plots the timings are shown in seconds. If the time is less than 60, the solver has proven the solution to be optimal.

The Time To Improvement (TTI) column indicates how long it took for the solver to find a solution that was equal to or better than the solution from RosterLogic Variation in relation to the time RosterLogic Variation took. For example, when there is a TTI value of 2.73x and RosterLogic Variation took 4.5 seconds, it means that the solver found a solution of equal or better value than RosterLogic Variation after 12.3 seconds. Note that the solver may have continued running until the 60-second time limit and found solutions even better than those merely on par with RosterLogic Variation. If no improvement over RosterLogic Variation was found, then a "-" is shown.

For each row, the best cost found is marked with dark blue, and values within five percent are marked with blue, and values within ten percent of the best are marked with light blue. The same is done independently for the timings.

Block size 15. In Table 3 results for instances with block size 15 are reported. RosterLogic Variation run times are between 2 and 7 seconds, and are for the most part dominating in speed. CP-SAT 9.12 with 10 threads dominates in solution quality, although the TTI is almost never below 1x (just three times), indicating the RosterLogic Variation still produces decent results faster.

CP-SAT 9.12 with 10 threads proves the optimal solution for 52 of the 80 instances, finds a solution but does not prove it optimal in 15 cases, and fails to solve 13 instances. This can be compared with CP-SAT 9.3 with 10 threads that proves the optimal solution in just 12 of the instances and fails to solve 36, with 32 solved but not proven optimal.

Gecode always finds solutions, but quite often very bad solutions. Gecode with the normal settings does not gain much from multiple threads, as it is used to help a search that is mostly weighted towards DFS exploration. Gecode with restarts produces a lot better solutions, and here the multi-threading quite often helps, although not always.

Block size 5. In Table 4 results for instances with block size 5 are reported. Using 5 minute blocks makes the problem a lot harder for all solvers. RosterLogic Variation increases its run-time to up to around 15 seconds here.

The big difference is that RosterLogic Variation also wins most cases on cost, except for a few of the ones with fewer workers that CP-SAT 9.12 with 10 threads can solve better, although there are also a lot of cases that CP-SAT does not solve at all. With 5-minute blocks, CP-SAT 9.12 with 10 threads proves optimality for only 9 instances, and CP-SAT 9.3 with 10 threads does not do it for any instance.

Gecode still finds solutions to all instances (and Gecode with restarts even find the best solution for one case), but the quality of the solutions found is quite bad in general.

B Large plots for CP-SAT 9.12 10 threads

In Figure 6 the plot for the results of CP-SAT 9.12 with 10 threads is given in a larger format for more detailed inspeaction compared with in Section 5.3. Similarly, in Figure 7 the results for CP-SAT 9.12 with 10 threads is shown.

C Pattern Swap Patterns

In Table 5 all the patterns defined for the Pattern Swap moves described in Section 4.3 for swapping *till* and *store* tasks. Note that these patterns are the ones that were found to be useful for the types of instances of WTV that we worked with in practice, for the common style of cost functions. These patterns are shown for 15-minute block sizes, for 5-minute block sizes the patterns are adjusted accordingly to account for the same durations (e.g., a pattern `SST` would become `SSSSSSTTT`). Out of the 10 defined patterns, 4 are symmetric which gives a total of 16 unique patterns.

Table 3 Benchmark results for block duration 15 minutes. Time (s), Cost (obj), TTI (Time To Improvement factor vs CBLS).

Instance	RosterLogic		CP-SAT 9.12 1t			CP-SAT 9.12 10t			CP-SAT 9.3 10t			Gecode 1t			Gecode 10t			Gecode-R 1t			Gecode-R 10t		
	Time	Cost	Time	Cost	TTI	Time	Cost	TTI	Time	Cost	TTI	Time	Cost	TTI	Time	Cost	TTI	Time	Cost	TTI	Time	Cost	TTI
L10 W8 - 1	2.1	1.00	60	18.25	-	2.7	0.92	1.21x	16.1	0.92	3.92x	60	15.32	-	60	14.83	-	60	4.57	-	60	3.13	-
L10 W8 - 2	2.1	1.00	60	9.23	-	29.4	0.99	2.43x	60	1.00	12.28x	60	5.01	-	60	4.99	-	60	2.32	-	60	2.51	-
L10 W8 - 3	1.8	1.00	60	-	-	2.3	0.96	1.25x	4.9	0.96	2.71x	60	9.84	-	60	9.93	-	60	6.50	-	60	4.08	-
L10 W8 - 4	2.3	1.00	60	-	-	3.5	0.77	1.54x	34.9	0.77	15.29x	60	11.80	-	60	10.62	-	60	2.37	-	60	3.28	-
L10 W10 - 1	2.5	1.00	60	20.70	-	3.4	0.92	1.31x	60	0.92	6.85x	60	19.32	-	60	18.94	-	60	11.09	-	60	10.71	-
L10 W10 - 2	2.5	1.00	60	8.91	-	21.7	0.90	2.38x	60	0.93	4.19x	60	10.29	-	60	10.08	-	60	3.86	-	60	3.82	-
L10 W10 - 3	3.0	1.00	60	-	-	60	-	-	60	-	-	60	6.74	-	60	5.66	-	60	5.36	-	60	4.19	-
L10 W10 - 4	3.6	1.00	60	-	-	60	-	-	60	-	-	60	21.35	-	60	18.33	-	60	3.27	-	60	8.04	-
L10 W12 - 1	3.2	1.00	60	12.33	-	4.2	0.84	1.31x	60	0.85	5.05x	60	13.93	-	60	13.48	-	60	3.44	-	60	3.75	-
L10 W12 - 2	3.6	1.00	60	4.97	-	60	0.94	6.87x	60	0.97	14.25x	60	6.60	-	60	6.26	-	60	3.28	-	60	2.87	-
L10 W12 - 3	3.0	1.00	60	-	-	56.0	0.69	18.36x	60	-	-	60	12.49	-	60	11.51	-	60	4.20	-	60	4.01	-
L10 W12 - 4	4.0	1.00	60	-	-	12.7	0.84	3.15x	60	-	-	60	25.74	-	60	21.62	-	60	5.44	-	60	6.26	-
L10 W14 - 1	4.2	1.00	60	13.07	-	60	0.89	2.62x	60	0.91	6.85x	60	7.10	-	60	6.20	-	60	3.18	-	60	2.49	-
L10 W14 - 2	4.7	1.00	60	18.69	-	60	1.02	-	60	1.66	-	60	17.61	-	60	17.63	-	60	5.86	-	60	5.40	-
L10 W14 - 3	4.2	1.00	60	-	-	19.7	0.85	4.64x	60	-	-	60	16.06	-	60	15.31	-	60	3.35	-	60	3.65	-
L10 W14 - 4	6.0	1.00	60	-	-	60	-	-	60	-	-	60	27.09	-	60	23.90	-	60	9.78	-	60	11.05	-
L10 W16 - 1	6.1	1.00	60	15.32	-	60	0.82	1.50x	60	0.95	7.44x	60	21.95	-	60	21.94	-	60	10.72	-	60	8.83	-
L10 W16 - 2	6.2	1.00	60	15.36	-	60	0.92	3.34x	60	1.01	-	60	12.19	-	60	11.79	-	60	2.52	-	60	2.64	-
L10 W16 - 3	5.5	1.00	60	-	-	60	-	-	60	-	-	60	12.80	-	60	12.83	-	60	6.13	-	60	6.04	-
L10 W16 - 4	6.2	1.00	60	-	-	60	-	-	60	-	-	60	10.09	-	60	8.46	-	60	3.63	-	60	4.10	-
L12 W8 - 1	2.5	1.00	60	22.94	-	9.0	0.70	1.87x	39.5	0.70	6.15x	60	26.01	-	60	25.50	-	60	9.22	-	60	10.03	-
L12 W8 - 2	2.5	1.00	60	19.97	-	3.7	0.88	1.44x	60	0.88	5.03x	60	13.63	-	60	13.24	-	60	5.69	-	60	5.50	-
L12 W8 - 3	1.9	1.00	60	-	-	9.2	0.90	4.90x	60	-	-	60	24.17	-	60	23.61	-	60	5.56	-	60	6.80	-
L12 W8 - 4	2.9	1.00	60	-	-	6.9	0.76	2.33x	60	-	-	60	10.87	-	60	10.25	-	60	2.28	-	60	3.14	-
L12 W10 - 1	2.9	1.00	60	17.38	-	8.8	0.97	2.40x	60	0.99	14.19x	60	15.05	-	60	14.67	-	60	4.62	-	60	4.42	-
L12 W10 - 2	2.9	1.00	60	9.07	-	8.4	0.89	1.46x	60	0.93	4.73x	60	13.80	-	60	12.69	-	60	4.99	-	60	3.68	-
L12 W10 - 3	2.6	1.00	60	-	-	41.3	0.72	16.00x	60	-	-	60	25.62	-	60	24.21	-	60	5.51	-	60	3.70	-
L12 W10 - 4	3.5	1.00	60	-	-	60	-	-	60	-	-	60	28.97	-	60	25.18	-	60	7.47	-	60	7.33	-
L12 W12 - 1	4.1	1.00	60	31.29	-	7.6	0.96	1.82x	60	1.06	-	60	24.26	-	60	24.09	-	60	7.73	-	60	7.86	-
L12 W12 - 2	4.1	1.00	60	13.82	-	60	0.98	2.99x	60	1.00	12.73x	60	19.40	-	60	18.64	-	60	4.58	-	60	3.58	-
L12 W12 - 3	3.4	1.00	60	-	-	5.2	0.88	1.53x	60	-	-	60	29.41	-	60	24.46	-	60	15.51	-	60	15.67	-
L12 W12 - 4	4.0	1.00	60	-	-	60	-	-	60	-	-	60	19.15	-	60	17.90	-	60	4.69	-	60	3.65	-
L12 W14 - 1	4.5	1.00	60	25.19	-	60	0.72	2.73x	60	1.60	-	60	28.84	-	60	28.42	-	60	17.07	-	60	10.36	-
L12 W14 - 2	5.1	1.00	60	14.24	-	60	1.02	-	60	1.47	-	60	16.59	-	60	16.42	-	60	5.70	-	60	5.73	-
L12 W14 - 3	3.9	1.00	60	-	-	27.2	0.78	6.88x	60	-	-	60	21.96	-	60	19.29	-	60	12.52	-	60	7.79	-
L12 W14 - 4	5.9	1.00	60	-	-	12.3	0.86	2.09x	60	-	-	60	26.09	-	60	23.81	-	60	8.17	-	60	6.37	-
L12 W16 - 1	6.0	1.00	60	14.72	-	60	0.57	2.12x	60	1.06	-	60	27.43	-	60	21.18	-	60	11.95	-	60	10.49	-
L12 W16 - 2	7.4	1.00	60	14.50	-	60	0.96	2.46x	60	1.05	-	60	18.11	-	60	17.79	-	60	5.24	-	60	6.28	-
L12 W16 - 3	4.7	1.00	60	-	-	12.1	0.44	2.56x	60	-	-	60	19.05	-	60	17.39	-	60	11.93	-	60	8.46	-
L12 W16 - 4	6.0	1.00	60	-	-	60	-	-	60	-	-	60	20.28	-	60	14.53	-	60	5.80	-	60	5.78	-
L14 W8 - 1	2.4	1.00	60	12.45	-	6.7	0.88	1.68x	60	0.89	5.57x	60	9.97	-	60	9.83	-	60	3.88	-	60	4.59	-
L14 W8 - 2	2.2	1.00	60	21.72	-	4.4	0.86	1.73x	51.9	0.86	3.91x	60	9.42	-	60	9.48	-	60	4.87	-	60	3.75	-
L14 W8 - 3	1.5	1.00	60	-	-	2.3	0.57	1.46x	3.1	0.57	2.00x	60	9.80	-	60	9.78	-	60	2.71	-	60	1.98	-
L14 W8 - 4	3.4	1.00	60	-	-	9.0	0.76	2.64x	60	-	-	60	17.46	-	60	16.67	-	60	3.17	-	60	3.19	-
L14 W10 - 1	2.7	1.00	60	41.24	-	7.0	0.66	1.71x	60	0.66	6.22x	60	53.93	-	60	52.95	-	60	7.56	-	60	13.71	-
L14 W10 - 2	3.4	1.00	60	11.65	-	7.1	0.89	1.90x	60	0.90	4.96x	60	11.96	-	60	11.85	-	60	3.40	-	60	2.69	-
L14 W10 - 3	2.6	1.00	60	-	-	3.9	0.94	1.50x	57.9	0.94	22.50x	60	23.20	-	60	20.87	-	60	6.34	-	60	3.99	-
L14 W10 - 4	4.3	1.00	60	-	-	6.9	0.39	1.59x	60	-	-	60	33.45	-	60	32.52	-	60	15.91	-	60	9.11	-
L14 W12 - 1	3.9	1.00	60	32.66	-	60	0.82	3.65x	60	1.10	-	60	42.25	-	60	41.51	-	60	9.08	-	60	6.76	-
L14 W12 - 2	3.5	1.00	60	17.96	-	60	0.91	5.38x	60	1.09	-	60	25.45	-	60	25.45	-	60	7.54	-	60	3.53	-
L14 W12 - 3	3.0	1.00	60	-	-	9.7	0.88	3.21x	60	-	-	60	39.57	-	60	38.52	-	60	13.89	-	60	13.98	-
L14 W12 - 4	4.2	1.00	60	-	-	25.7	0.67	6.14x	60	-	-	60	31.83	-	60	28.47	-	60	8.05	-	60	12.92	-
L14 W14 - 1	5.6	1.00	60	40.53	-	9.1	0.59	1.59x	60	0.86	9.19x	60	74.99	-	60	71.76	-	60	25.43	-	60	24.57	-
L14 W14 - 2	5.5	1.00	60	39.44	-	16.8	0.78	3.02x	60	1.26	-	60	42.42	-	60	41.71	-	60	16.31	-	60	17.33	-
L14 W14 - 3	4.3	1.00	60	-	-	42.3	0.43	9.91x	60	-	-	60	11.83	-	60	11.36	-	60	6.38	-	60	5.82	-
L14 W14 - 4	6.0	1.00	60	-	-	17.6	0.86	2.91x	60	-	-	60	37.25	-	60	35.86	-	60	10.78	-	60	9.30	-
L14 W16 - 1	6.4	1.00	60	46.03	-	5.7	0.51	0.87x	60	1.77	-	60	55.44	-	60	48.57	-	60	12.84	-	60	14.05	-
L14 W16 - 2	7.2	1.00	60	29.76	-	23.7	0.82	2.48x	60	1.23	-	60	34.35	-	60	31.23	-	60	10.47	-	60	13.00	-
L14 W16 - 3	5.1	1.00	60	-	-	60	-	-	60	-	-	60	20.80	-	60	20.77	-	60	8.71	-	60	6.80	-
L14 W16 - 4	6.9	1.00	60	-	-	60	-	-	60	-	-	60	22.92	-	60	18.25	-	60	6.79	-	60	9.39	-
L16 W8 - 1	2.4	1.00	60	21.55	-	4.3	0.43	1.47x	7.7	0.43	2.25x	60	31.02	-	60	28.71	-	60	16.08	-	60	9.24	-
L16 W8 - 2	2.0	1.00	60	31.99	-	3.3	0.87	1.59x	13.2	0.87	4.32x	60	36.22	-	60	34.11	-	60	5.95	-	60	5.74	-
L16 W8 - 3	1.7	1.00	60	-	-	3.9	0.93	2.33x	14.9	0.93	8.92x	60	10.05	-	60	7.96	-	60	3.09	-	60	3.14	-
L16 W8 - 4	2.9	1.00	60	-	-	9.6	0.92	3.32x	60	-	-	60	28.85	-	60	28.82	-	60	7.16	-	60	6.32	-
L16 W10 - 1	2.9	1.00	60	28.52	-	4.0	0.60	1.13x	32.9	0.60	3.05x	60	35.48	-	60	33.19	-	60	12.26	-	60	9.14	-
L16 W10 - 2	3.0	1.00	60	9.89	-	60	0.92	13.40x	60	1.01	-	60	12.25	-	60	12.03	-	60	3.08	-	60	3.10	-
L16 W10 - 3	2.4	1.00	60	-	-	5.7	0.72	2.34x	60	-	-	60	46.83	-	60	45.35	-	60	20.38	-	60	10.11	-
L16 W10 - 4	3.7	1.00	60	-	-	11.2	0.75	2.99x	60	-	-	60	28.59	-	60	27.88	-	60	10.67	-	60	11.06	-
L16 W12 - 1	4.3	1.00	60	30.46	-	5.3	0.69	0.97x	60	0.75	6.03x	60	50.89	-	60	50.28	-	60	10.31	-	60	10.14	-
L16 W12 - 2	3.7	1.00	60	19.01	-	60	0.98	3.58x	60	1.00	16.08x	60	19.77	-	60	19.59	-	60	3.91	-	60	3.49	-
L16 W12 - 3	3.1	1.00	60	-	-	8.3	0.91	2.68x	60	-	-	60	27.99	-	60	27.99	-	60	10.32	-	60	7.12	-
L16 W12 - 4	4.5	1.00	60	-	-	60	-	-	60	-	-	60	45.27	-	60	40.02	-	60	15.63	-	60	12.91	-
L16 W14 - 1	4.7	1.00	60	31.25	-	47.6	0.44	2.75x	60	1.17	-	60	49.41	-	60	47.04	-	60	15.92	-	60	14.14	-
L16 W14 - 2	5.5	1.00	60	15.67	-	7.5	0.88	1.35x	60	1.01	-	60	10.90	-	60	11.30	-	60	4.18	-	60	3.62	-
L16 W14 - 3	4.5	1.00	60	-	-	48.6	0.72	10.89x	60	-	-	60	35.10	-	60	35.71	-	60	9.85	-	60	8.48	-
L16 W14 - 4	5.7	1.00	60	-	-	60	-	-	60	-	-	60	15.37	-	60	12.76	-	60	7.91	-	60	7.13	-
L16 W16 - 1	5.7	1.00	60	46.80	-	60	1.17	-	60	1.69	-	60	51.67	-	60	47.37	-	60	11.80	-	60	14.91	-
L16 W16 - 2	7.2	1.00	60	73.40	-	6.9	0.55	0.94x	43.0	0.55	5.33x	60	65.55	-	60	64.76	-	60	31.07	-	60	33.96	-
L16 W16 - 3	5.4	1.00	60	-	-	7.2	0.49	1.33x	60	-	-	60	95.42	-	60	89.10	-	60	27.90	-	60	28.22	-
L16 W16 - 4	6.3	1.00	60	-	-	60	-	-	60	-	-	60	41.74	-	60	37.15	-	60	23.02	-	60	21.41	-

Table 4 Benchmark results for block duration 5 minutes. Time (s), Cost (obj), TTI (Time To Improvement factor vs CBLS).

Instance	RosterLogic		CP-SAT 9.12 1t			CP-SAT 9.12 10t			CP-SAT 9.3 10t			Gecode 1t			Gecode 10t			Gecode-R 1t			Gecode-R 10t		
	Time	Cost	Time	Cost	TTI	Time	Cost	TTI	Time	Cost	TTI	Time	Cost	TTI	Time	Cost	TTI	Time	Cost	TTI	Time	Cost	TTI
L10 W8 - 1	3.7	1.00	60	11.01	-	34.0	0.31	6.32x	60	1.16	-	60	9.44	-	60	9.35	-	60	4.16	-	60	1.85	-
L10 W8 - 2	3.8	1.00	60	10.65	-	58.8	0.47	5.11x	60	0.91	10.31x	60	14.91	-	60	14.91	-	60	4.97	-	60	2.59	-
L10 W8 - 3	3.0	1.00	60	-	-	60	-	-	60	-	-	60	11.18	-	60	11.15	-	60	3.42	-	60	3.73	-
L10 W8 - 4	4.0	1.00	60	-	-	28.8	0.39	7.14x	60	-	-	60	14.34	-	60	13.50	-	60	7.62	-	60	14.17	-
L10 W10 - 1	5.1	1.00	60	14.09	-	60	0.86	7.22x	60	3.08	-	60	18.36	-	60	18.45	-	60	13.48	-	60	7.55	-
L10 W10 - 2	5.4	1.00	60	8.04	-	60	0.47	5.67x	60	3.18	-	60	12.00	-	60	11.27	-	60	3.70	-	60	3.87	-
L10 W10 - 3	5.4	1.00	60	-	-	60	-	-	60	-	-	60	14.44	-	60	14.24	-	60	7.75	-	60	13.12	-
L10 W10 - 4	5.3	1.00	60	-	-	60	-	-	60	-	-	60	14.86	-	60	14.99	-	60	5.42	-	60	3.45	-
L10 W12 - 1	5.6	1.00	60	25.88	-	33.2	0.33	5.90x	60	7.52	-	60	18.04	-	60	18.06	-	60	7.90	-	60	5.16	-
L10 W12 - 2	7.3	1.00	60	13.34	-	60	2.32	-	60	7.23	-	60	19.83	-	60	20.10	-	60	5.82	-	60	6.04	-
L10 W12 - 3	6.8	1.00	60	-	-	60	-	-	60	-	-	60	13.31	-	60	13.95	-	60	8.58	-	60	7.60	-
L10 W12 - 4	7.4	1.00	60	-	-	52.1	0.37	6.99x	60	-	-	60	14.45	-	60	14.76	-	60	5.37	-	60	5.44	-
L10 W14 - 1	8.1	1.00	60	15.75	-	60	1.33	-	60	8.55	-	60	11.74	-	60	11.38	-	60	8.57	-	60	7.40	-
L10 W14 - 2	9.3	1.00	60	10.86	-	60	4.48	-	60	21.03	-	60	16.99	-	60	16.40	-	60	6.32	-	60	5.81	-
L10 W14 - 3	8.9	1.00	60	-	-	60	-	-	60	-	-	60	9.69	-	60	10.11	-	60	2.37	-	60	2.20	-
L10 W14 - 4	9.1	1.00	60	-	-	60	-	-	60	-	-	60	13.42	-	60	13.05	-	60	8.47	-	60	6.28	-
L10 W16 - 1	10.5	1.00	60	18.93	-	60	5.17	-	60	10.92	-	60	13.69	-	60	13.82	-	60	11.86	-	60	8.83	-
L10 W16 - 2	12.9	1.00	60	17.22	-	60	7.00	-	60	11.58	-	60	15.33	-	60	15.21	-	60	8.01	-	60	7.17	-
L10 W16 - 3	10.2	1.00	60	-	-	60	-	-	60	-	-	60	18.79	-	60	17.89	-	60	9.48	-	60	9.93	-
L10 W16 - 4	11.8	1.00	60	-	-	60	-	-	60	-	-	60	17.10	-	60	17.44	-	60	10.52	-	60	9.67	-
L12 W8 - 1	5.1	1.00	60	10.47	-	60	0.61	7.92x	60	1.78	-	60	12.73	-	60	11.78	-	60	6.41	-	60	7.36	-
L12 W8 - 2	4.1	1.00	60	32.82	-	60	0.73	11.89x	60	3.14	-	60	25.43	-	60	28.70	-	60	19.33	-	60	13.30	-
L12 W8 - 3	3.5	1.00	60	-	-	60	-	-	60	-	-	60	16.69	-	60	16.21	-	60	3.51	-	60	3.01	-
L12 W8 - 4	5.4	1.00	60	-	-	60	-	-	60	-	-	60	20.05	-	60	19.51	-	60	11.98	-	60	7.50	-
L12 W10 - 1	5.7	1.00	60	21.35	-	60	1.62	-	60	4.60	-	60	31.18	-	60	29.92	-	60	15.11	-	60	13.03	-
L12 W10 - 2	5.8	1.00	60	18.93	-	60	1.26	-	60	2.29	-	60	18.08	-	60	16.80	-	60	4.89	-	60	4.13	-
L12 W10 - 3	5.2	1.00	60	-	-	28.0	0.40	5.40x	60	-	-	60	21.60	-	60	21.34	-	60	7.18	-	60	4.29	-
L12 W10 - 4	7.7	1.00	60	-	-	60	-	-	60	-	-	60	16.19	-	60	15.92	-	60	4.43	-	60	4.54	-
L12 W12 - 1	7.7	1.00	60	12.71	-	60	1.26	-	60	7.48	-	60	13.95	-	60	13.84	-	60	11.12	-	60	7.49	-
L12 W12 - 2	7.8	1.00	60	22.00	-	60	2.02	-	60	14.14	-	60	25.26	-	60	22.92	-	60	11.55	-	60	8.61	-
L12 W12 - 3	6.7	1.00	60	-	-	60	-	-	60	-	-	60	25.75	-	60	24.76	-	60	10.64	-	60	11.23	-
L12 W12 - 4	9.5	1.00	60	-	-	60	-	-	60	-	-	60	12.89	-	60	12.54	-	60	6.27	-	60	12.69	-
L12 W14 - 1	10.7	1.00	60	18.99	-	60	7.66	-	60	7.27	-	60	27.38	-	60	27.45	-	60	15.21	-	60	14.51	-
L12 W14 - 2	11.3	1.00	60	31.38	-	60	7.18	-	60	13.65	-	60	29.28	-	60	26.76	-	60	15.71	-	60	17.68	-
L12 W14 - 3	8.9	1.00	60	-	-	60	-	-	60	-	-	60	16.83	-	60	16.72	-	60	6.88	-	60	6.74	-
L12 W14 - 4	12.7	1.00	60	-	-	60	-	-	60	-	-	60	27.52	-	60	26.34	-	60	8.23	-	60	7.75	-
L12 W16 - 1	15.4	1.00	60	33.05	-	60	14.88	-	60	18.93	-	60	31.58	-	60	31.66	-	60	20.33	-	60	22.42	-
L12 W16 - 2	13.8	1.00	60	19.88	-	60	15.31	-	60	73.68	-	60	25.58	-	60	22.86	-	60	15.77	-	60	12.87	-
L12 W16 - 3	11.6	1.00	60	-	-	60	-	-	60	-	-	60	43.82	-	60	41.61	-	60	15.72	-	60	16.51	-
L12 W16 - 4	15.1	1.00	60	-	-	60	-	-	60	-	-	60	32.77	-	60	32.59	-	60	18.53	-	60	17.01	-
L14 W8 - 1	4.8	1.00	60	20.76	-	60	1.63	-	60	4.18	-	60	28.56	-	60	28.35	-	60	8.72	-	60	9.79	-
L14 W8 - 2	4.0	1.00	60	17.14	-	60	0.78	7.11x	60	1.27	-	60	12.29	-	60	12.59	-	60	2.67	-	60	4.30	-
L14 W8 - 3	3.3	1.00	60	-	-	60	-	-	60	-	-	60	10.32	-	60	10.31	-	60	3.96	-	60	5.35	-
L14 W8 - 4	6.0	1.00	60	-	-	60	-	-	60	-	-	60	15.97	-	60	16.13	-	60	7.24	-	60	7.11	-
L14 W10 - 1	6.0	1.00	60	30.10	-	60	2.90	-	60	2.86	-	60	34.19	-	60	31.50	-	60	10.52	-	60	9.24	-
L14 W10 - 2	6.9	1.00	60	20.78	-	60	1.47	-	60	4.55	-	60	23.89	-	60	22.10	-	60	5.94	-	60	3.92	-
L14 W10 - 3	4.7	1.00	60	-	-	60	-	-	60	-	-	60	20.01	-	60	18.47	-	60	5.51	-	60	7.49	-
L14 W10 - 4	8.2	1.00	60	-	-	60	-	-	60	-	-	60	15.26	-	60	15.81	-	60	9.02	-	60	7.90	-
L14 W12 - 1	8.2	1.00	60	28.86	-	60	3.29	-	60	10.85	-	60	31.69	-	60	31.06	-	60	17.41	-	60	13.01	-
L14 W12 - 2	7.2	1.00	60	15.81	-	60	2.17	-	60	8.93	-	60	21.98	-	60	21.98	-	60	7.14	-	60	7.91	-
L14 W12 - 3	6.9	1.00	60	-	-	60	-	-	60	-	-	60	35.29	-	60	35.33	-	60	17.73	-	60	16.67	-
L14 W12 - 4	11.3	1.00	60	-	-	60	-	-	60	-	-	60	35.79	-	60	31.72	-	60	17.81	-	60	18.93	-
L14 W14 - 1	11.1	1.00	60	56.10	-	60	13.08	-	60	31.00	-	60	95.87	-	60	100.07	-	60	44.16	-	60	47.80	-
L14 W14 - 2	11.4	1.00	60	251.84	-	56.4	1.04	-	60	124.31	-	60	298.87	-	60	279.27	-	60	105.18	-	60	63.73	-
L14 W14 - 3	9.6	1.00	60	-	-	60	-	-	60	-	-	60	30.05	-	60	31.09	-	60	20.99	-	60	21.33	-
L14 W14 - 4	12.4	1.00	60	-	-	60	-	-	60	-	-	60	22.69	-	60	22.81	-	60	10.83	-	60	10.94	-
L14 W16 - 1	14.5	1.00	60	13.85	-	60	5.62	-	60	12.47	-	60	25.19	-	60	24.45	-	60	10.53	-	60	8.94	-
L14 W16 - 2	15.2	1.00	60	31.58	-	60	31.85	-	60	119.49	-	60	30.60	-	60	30.72	-	60	15.34	-	60	18.60	-
L14 W16 - 3	11.7	1.00	60	-	-	60	-	-	60	-	-	60	23.80	-	60	24.45	-	60	12.61	-	60	12.25	-
L14 W16 - 4	14.6	1.00	60	-	-	60	-	-	60	-	-	60	35.06	-	60	36.46	-	60	19.70	-	60	20.05	-
L16 W8 - 1	4.6	1.00	60	44.53	-	43.2	0.50	6.24x	60	1.38	-	60	46.55	-	60	41.41	-	60	10.43	-	60	9.38	-
L16 W8 - 2	4.4	1.00	60	8.83	-	60	0.43	5.94x	60	0.89	11.61x	60	10.61	-	60	9.69	-	60	7.06	-	60	5.73	-
L16 W8 - 3	3.5	1.00	60	-	-	60	-	-	60	-	-	60	2.76	-	60	2.61	-	60	0.28	17.26x	60	0.69	17.26x
L16 W8 - 4	5.7	1.00	60	-	-	60	-	-	60	-	-	60	19.38	-	60	16.51	-	60	8.88	-	60	5.00	-
L16 W10 - 1	6.2	1.00	60	21.09	-	60	2.18	-	60	5.77	-	60	40.03	-	60	38.22	-	60	14.40	-	60	13.91	-
L16 W10 - 2	7.4	1.00	60	20.71	-	60	1.28	-	60	4.94	-	60	22.13	-	60	21.69	-	60	5.81	-	60	5.22	-
L16 W10 - 3	5.1	1.00	60	-	-	46.8	0.35	9.09x	60	-	-	60	36.41	-	60	34.80	-	60	16.33	-	60	11.65	-
L16 W10 - 4	7.9	1.00	60	-	-	60	-	-	60	-	-	60	29.91	-	60	29.80	-	60	13.12	-	60	12.58	-
L16 W12 - 1	8.9	1.00	60	41.72	-	60	3.19	-	60	19.23	-	60	48.93	-	60	45.33	-	60	10.23	-	60	11.44	-
L16 W12 - 2	8.2	1.00	60	6.20	-	60	2.51	-	60	4.77	-	60	11.51	-	60	11.29	-	60	3.15	-	60	3.28	-
L16 W12 - 3	7.3	1.00	60	-	-	60	-	-	60	-	-	60	33.30	-	60	33.30	-	60	18.58	-	60	17.54	-
L16 W12 - 4	10.4	1.00	60	-	-	60	-	-	60	-	-	60	24.91	-	60	23.92	-	60	14.65	-	60	14.36	-
L16 W14 - 1	10.9	1.00	60	39.54	-	60	9.48	-	60	13.69	-	60	47.65	-	60	47.66	-	60	15.34	-	60	12.26	-
L16 W14 - 2	12.9	1.00	60	28.62	-	60	15.41	-	60	18.53	-	60	36.33	-	60	36.39	-	60	17.28	-	60	18.49	-
L16 W14 - 3	10.1	1.00	60	-	-	60	-	-	60	-	-	60	27.60	-	60	26.76	-	60	10.44	-	60	7.00	-
L16 W14 - 4	13.0	1.00	60	-	-	60	-	-	60	-	-	60	28.35	-	60	27.92	-	60	9.68	-	60	8.82	-
L16 W16 - 1	16.2	1.00	60	12.75	-	60	5.99	-	60	6.93	-	60	19.68	-	60	19.78	-	60	8.88	-	60	9.62	-
L16 W16 - 2	15.7	1.00	60	82.95	-	60	38.01	-	60	55.44	-	60	99.29	-	60	102.26	-	60	42.53	-	60	35.53	-
L16 W16 - 3	11.6	1.00	60	-	-	60	-	-	60	-	-	60	35.34	-	60	35.56	-	60	16.06	-	60	15.59	-
L16 W16 - 4	15.9	1.00	60	-	-	60	-	-	60	-	-	60	26.90	-	60	25.05	-	60	11.47	-	60	9.24	-

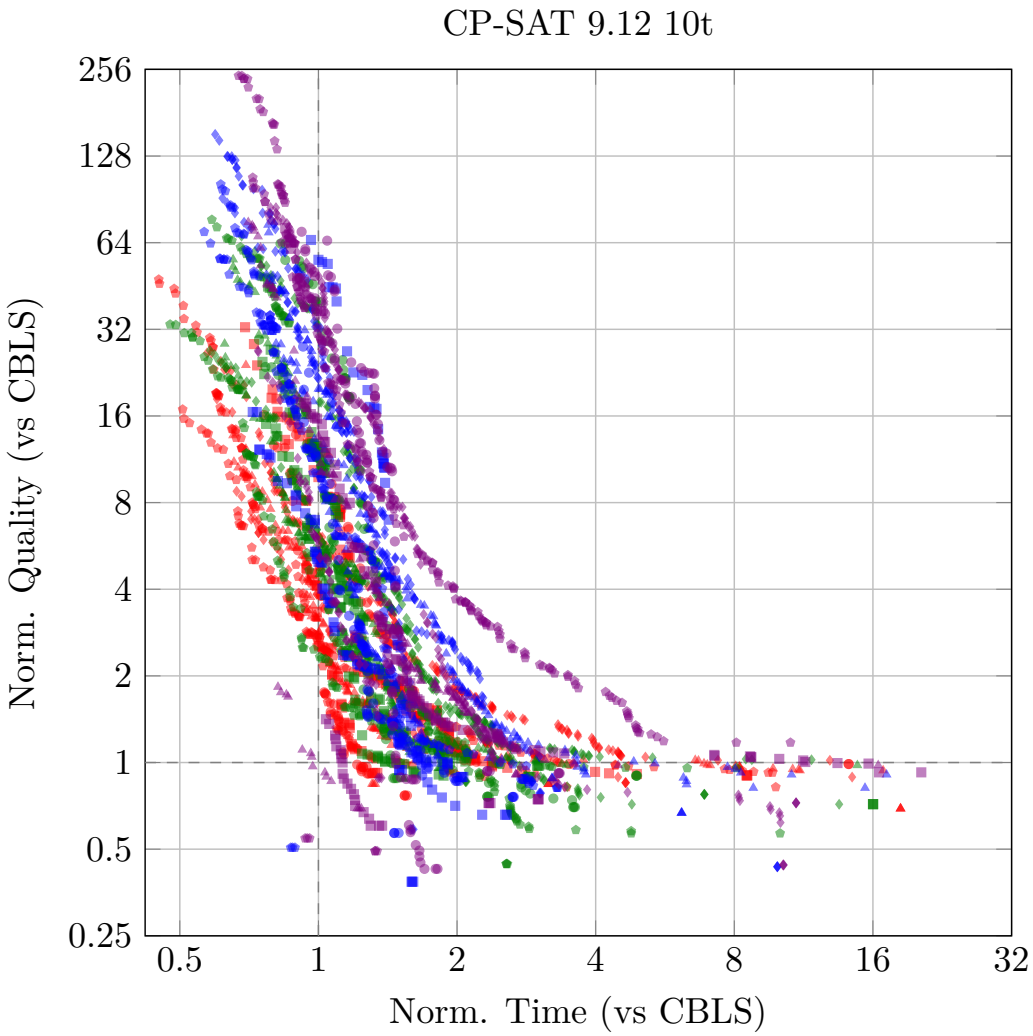


Figure 6 Normalized Quality vs. Time CP-SAT 9.12 10 threads vs RosterLogic Variation for block size 15. Lower values are better. Dashed lines at 1.0 mark CBLS performance. Shapes denote number of workers W8 ●, W10 ■, W12 ▲, W14 ♦, and W16 as ⬟. Colors denote length of schedule: L10 ●, L12 ●, L14 ●, and L16 as ●.

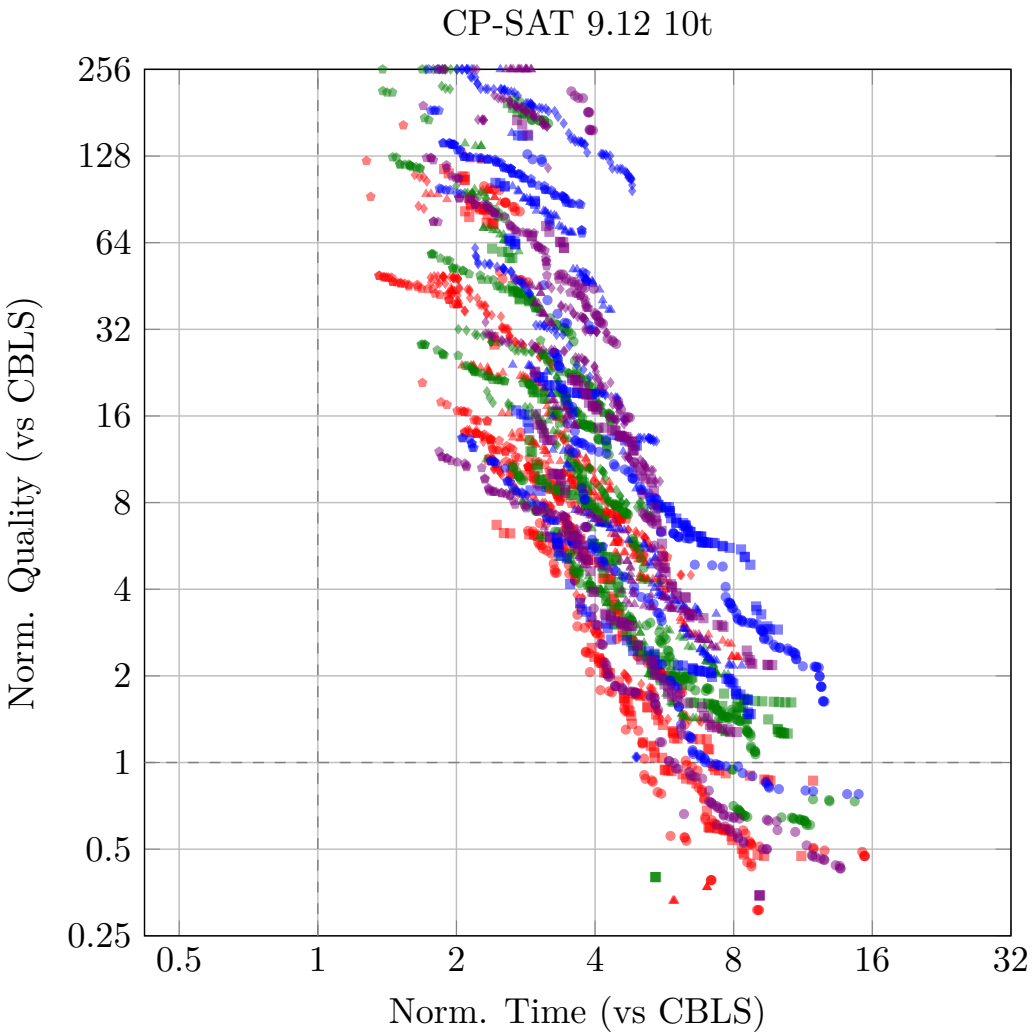


Figure 7 Normalized Quality vs. Time CP-SAT 9.12 10 threads vs RosterLogic Variation for block size 5. Lower values are better. Dashed lines at 1.0 mark CBLS performance. Shapes denote number of workers W8 ●, W10 ■, W12 ▲, W14 ♦, and W16 as ⬟. Colors denote length of schedule: L10 ●, L12 ●, L14 ●, and L16 as ●.

Table 5 The 10 common patterns used for the pattern swap move generator are shown to the left. For each pattern, the reversed version is also shown to the right.

Pattern	Reversed Pattern
S S S S	S S S S
S S S T T	T T S S S
S S T T T	T T T S S
S S S S T T	T T S S S S
S S S T T T	T T T S S S
S S T T T T	T T T T S S
S S S S T T T	T T T S S S S
S S S T T T T	T T T T S S S
S S S S T T T T	T T T T S S S S
S S S T T T S S S	S S S T T T S S S

Aircraft Resource-Constrained Assembly Line Balancing with Learning Effect: A Constraint Programming Approach

Duc Anh Le[1]
DTIS, ONERA, Université de Toulouse, France

Stéphanie Roussel
DTIS, ONERA, Université de Toulouse, France

Christophe Lecoutre
CRIL, Université d'Artois & CNRS, France

Abstract

Balancing aeronautical assembly lines is a major challenge in modern aerospace manufacturing. Aircraft manufacturing plants typically have a predetermined production rate, but the production system requires a period of adaptation at start-up. This phenomenon, known as the learning effect, refers to the gradual improvement in efficiency through task repetition, thereby reducing task duration. However, the stability of an assembly line is also a critical factor, as any change in the production process incurs costs. In this study, Constraint Programming (CP) is used to optimise assembly line balancing, taking into account the learning effect to address the trade-off between achieving target production rates and minimising adjustments to the line.

2012 ACM Subject Classification Applied computing → Decision analysis

Keywords and phrases Assembly Line Balancing, Resource-Constrained, Learning Effect, Ramp-Up, Aeronautic, Constraint Programming, Dominance-Breaking

Digital Object Identifier 10.4230/LIPIcs.CP.2025.25

Supplementary Material *Dataset*: `https://doi.org/10.57745/EBDN5W` [25]

Funding *Stéphanie Roussel*: This work has benefitted from 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.
Christophe Lecoutre: This work has benefited from the support of the National Research Agency under France 2030, MAIA Project ANR-22-EXES-0009.

1 Introduction

The Assembly Line Balancing Problem (ALBP) is a fundamental optimisation challenge in manufacturing, focused on efficiently assigning tasks to workstations along an assembly line. The goal is to balance workloads across stations while minimising idle time and ensuring that the production rate meets demand. Assembly lines are categorised into three types: single-model, mixed-model, and multi-model, depending on the number of different products that can be fabricated at the same time on the line. There are different layouts of assembly lines, including straight lines, U-shaped lines, lines with parallel workstations, and two-sided lines, but the most common is the straight one. The production rate of an assembly line is represented by its cycle time, which is the time between the completion of two consecutive assembled units. Without parallel workstations, the cycle time is also the time required for

[1] Corresponding author

© Duc Anh Le, Stéphanie Roussel, and Christophe Lecoutre;
licensed under Creative Commons License CC-BY 4.0

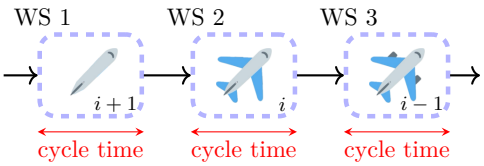


Figure 1 A straight single-model aircraft assembly line with three workstations. $i+1$, i, and $i-1$ are successive units being assembled.

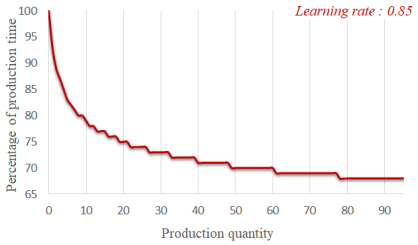


Figure 2 A log-linear learning curve: the production time decreases by 15% recursively each time the production quantity doubles.

all workstations to complete their assigned tasks before passing the unit to the next station. Simple aeronautical assembly lines are usually single-model, straight-line systems, where the aircraft being assembled enter the assembly line, one by one, at the first workstation, and pass successively through each station until they reach the last, where they are completed and leave the line (Figure 1). Given the complexity of assembling aircraft components, aerospace companies must carefully distribute and organise workloads to maximise the production rate and resource utilisation, reduce costs, and ensure the stability of the production system while maintaining quality standards. Effective line balancing ensures a smooth workflow, prevents bottlenecks, and maximises resource utilisation. By continually refining assembly processes, aerospace companies can improve productivity, meet the industry's demanding deadlines and remain competitive in the marketplace.

Aircraft manufacturing plants are designed to produce at a certain rate. However, once production begins, it takes time for the system to reach this production rate. Workers need to gain experience, improve their skills, improve the supply chain management and optimise the production process. Therefore, the initial production rate of plants is always lower than the target rate, and the difference between the two can be significant. As production progresses, continuous improvements in workflow optimisation, automation integration and experience help to narrow this gap. This phenomenon, known as the *learning effect*, refers to the gradual improvement in production system efficiency through task repetition, resulting in reduced task durations [38]. The variation in duration based on the repetition of the same task is called a *learning curve* (Figure 2), and the period in which the task duration continuously improves until it reaches a steady state is referred to as the *learning stage*. The phase required for the production system to reach the target production rate is called the *ramp-up stage*, which is currently a challenging topic of great interest in manufacturing [12, 21].

Since the time required to perform a task in a manufacturing environment usually fluctuates over time due to the learning effect, the design of the assembly line and the assembly process is continually refined. However, the stability of an assembly line is also a critical factor, as any change in the production process incurs costs. There are a number of potential causes for this issue, including the time taken to install or deinstall the heavy machines between workstations, errors made by workers when the assembly process changes and becomes unfamiliar, or changes in inventory management and the supply chain, etc. This creates a clear trade-off decision for aircraft manufacturers: pursuit of a continuously optimised assembly line must be weighed against the cost of making changes to the production process. Indeed, if the assembly line is continuously optimised, the production time will consistently decrease, allowing the system to reach the desired production rate more quickly. However, this process also carries an increased risk and higher associated costs due to the greater number of changes being implemented on the line.

In this work, we investigate the use of Constraint Programming (CP) to address the Resource-Constrained Assembly Line Balancing Problem considering the learning effect. We denote this problem as RC-ALBP/L. The decision, made under resource-constrained conditions, involves a trade-off between the ramp-up time – the time required to reach the target production rate or cycle time – and the number of adjustments to the line. This decision includes determining the task and resource allocation to each workstation, as well as setting the cycle time for each production period. Note that while industrial partners have shown strong interest in the challenge of ramping up production on an assembly line, this research has been conducted in an academic setting. Meetings are planned with industrial partners in the near future on this topic.

The paper is structured as follows. After presenting some related works in Section 2, RC-ALBP/L and its encoding in Optimization Programming Language (OPL) [36] are formally introduced in Section 3 and Section 4. The results of the experiments we have performed are presented in Section 5. Finally, we conclude and discuss some perspectives in Section 6.

2 Related Works

The Assembly Line Balancing Problem (ALBP) is a well-known problem in the literature. Starting from the Simple ALBP (SALBP) [20], numerous extensions have been proposed over the years to address its various dimensions [7, 8, 10]. In this study, we specifically focus on Resource-Constrained ALBP (RC-ALBP), scheduling in the presence of the learning effect, and ALBP with the learning effect.

Resource-Constrained ALBP. This issue has been referred to by a number of names in the past [11, 37], but it was not formally defined until 2005 in [2]. The definition of RC-ALBP was further refined in [16], where the authors pointed out that the resources required by tasks can be simple or multiple, alternative and/or concurrent. The complexity of RC-ALBP has been proven to be strongly NP-hard in [29], even without considering scheduling aspects. When task parallelism is allowed at the workstation level, the decision making process extends beyond task and resource allocation to include the determination of task start times. This particular variant of RC-ALBP has been widely studied under the name Multi-Manned ALBP, which was first introduced in [17]. However, most of these studies still did not consider cumulative resource scheduling, treating each worker as a separate capacity-one resource, and assuming that each task required only one worker. A number of solution approaches have been developed to tackle RC-ALBP, both with and without scheduling considerations, including techniques based on CP [3, 14, 18, 33, 34, 35, 40]. In [14, 35, 40], the authors compared the performance of CP with that of Mixed-Integer Linear Programming (MILP) and some incomplete solution approaches (e.g. Simulated Annealing, Genetic Algorithms). The results showed that CP outperforms the other approaches in this context.

Learning Effect and Scheduling. The learning effect was first documented in [38], where the author introduced a log-linear learning curve (Figure 2) that we incorporate in our experiments. Since then, numerous types of learning curves have been developed. A historical review and a meta-analysis of various learning curves are available in [19] and [39]. In scheduling environments, learning can be approached in two ways: (a) the *position-based approach*, where the extent of learning depends on the number of times a task has been carried out; and (b) the *sum-of-processing-time approach*, which accounts for the total processing

time of all similar tasks completed up to that point [9]. There are notable works on the scheduling problem that consider the learning effect. The multi-mode Resource-Constrained Project Scheduling Problem (RCPSP) with learning effect is addressed in [31, 32], where the duration of each task depends on the amount of resources performed on it. In [22], four CP formulations for the RCPSP with the learning effect are introduced, along with an empirical comparison of their scheduling and lower bounding performance. In [26], the authors consider the learning effect in the high-multiplicity RCPSP, where multiple repetitive projects need to be scheduled, using CP and prove the existence of symmetrical solutions in this problem.

ALBP with Learning Effect. Although ALBP has been widely studied with many extensions, only a few have considered the learning effect and most of them apply the position-based approach. A review of works conducted before 2023 that account for the learning effect in ALBP can be found in [1]. A relaxed version of SALBP with a learning effect (SALBP/L) was first addressed in [15]. The authors proved that the optimal task allocation, while minimising the throughput time of a batch product, is imbalanced in the presence of homogeneous learning. This work has been extended to SALBP/L in [23]. In [30], a modified branch-and-bound algorithm and a fast heuristic based on a task priority rule for SALBP/L, while minimising the number of workstations and the learning stage duration, have been proposed. However, the learning effect is relaxed since the line is considered pre-filled, all tasks have the same number of repetitions in each cycle, and the cycle time can only be changed in predetermined periods. Integer Linear Programming (ILP) and a matheuristic approach, which is based on variable neighborhood search and dynamic programming, have been used to address the same problem in [6]. In [27], the authors provided two rebalancing procedure for SALBP/L to reduce the number of workstations on the line. Nevertheless, this work still considers a relaxed view of the learning effect when counting the number of task repetitions for the position-based learning approach. In fact, a task from a later-entered unit can be processed earlier than one from an earlier unit, and more than one instance of the same task might be processed simultaneously for different units. More recently, mixed-model ALBP incorporating the learning effect has been addressed using ILP and matheuristic approaches, further expanding research in this area [4, 28].

To the best of our knowledge, no work in the literature simultaneously addresses RC-ALBP considering scheduling aspects with task durations dependent on the learning effect. As this is a novel problem, we adopt a complete approach (i.e. an approach able to prove optimality and to provide optimality gaps) in order to establish a baseline. In the literature, it has been shown that CP outperforms MILP in similar problem contexts, which motivates our choice for CP. Our contributions through this work are:

- Formally define RC-ALBP/L with an individual learning curve associated with each task.
- Propose an OPL encoding for RC-ALBP/L.
- Introduce a dominance-breaking method, while proving its correctness.
- Identify three solving approaches based on the constraint programming model.
- Conduct an experiment on industrial aircraft assembly line benchmarks, allowing us to compare CP-based solving approaches, along with an ILP approach.

3 Problem Description

The assembly line under consideration is single-model and straight. Moreover, due to the nature of the aircraft assembly line, we have included the notion of a *zone* that represents a subpart of the aircraft in which assembly tasks are performed. Each workstation contains

the same set of different zones, either inside or outside the aircraft. Each zone has a capacity and can be treated as a cumulative, renewable resource for task execution. Furthermore, for safety reasons, certain tasks may disable specific zones during their execution. When a zone is disabled, any task that requires it will be unable to proceed. However, tasks that disable the same zone can run simultaneously. In this section, we formally describe the components required to define an instance of the problem we consider, namely RC-ALBP/L, aiming to minimise the ramp-up time and the number of adjustments to the line. We do not consider rebalancing or dynamic task allocation in this work.

3.1 Problem Inputs

An instance of RC-ALBP/L is defined by a tuple consisting of the following elements.

- The target cycle time c_{tgt}, which is the cycle time we want to reach.
- The maximum value of cycle time c_{max}, which is the longest allowable time between the completion of two consecutive assembled units.
- The number of workstations W. We denote $\mathcal{W} = \{1, \ldots, W\}$, or equivalently $[\![1, W]\!]$, as the set of workstations. Workstations are ordered from 1 to W on the line.
- The number of periods P. A period is the time interval during which a unit stays at a workstation, and the cycle time in each period corresponds to the duration of this interval. We denote $\mathcal{P} = [\![1, P]\!]$ as the set of periods.
- The set of tasks $\mathcal{T}$ ($|\mathcal{T}| = T$), with for each task $t \in \mathcal{T}$, a monotonically decreasing duration function, denoted as $dur_t : \mathbb{Z} \to \mathbb{N}$, which returns the duration of the task based on the number of times it has been completely executed before, or zero if the input is negative. Note that in period number p, the number of complete executions of a task allocated to station number w is $p - w$. For instance, the duration of a task t allocated to workstation 3 in period 2 is $dur_t(2-3) = dur_t(-1) = 0$, since workstation 3 is not yet in service during period 2. During period 5, its duration will be $dur_t(5-3) = dur_t(2)$, because t has already been executed twice (in periods 3 and 4).
- The graph $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ represents the precedence relationship between tasks in $\mathcal{T}$. An arc $(t, u) \in \mathcal{A}$ indicates that task t must be finished before the start of task u.
- The set of resources $\mathcal{R}$, where, for each resource $r \in \mathcal{R}$, $capa_r$ denotes the capacity of r, and $cons_{r,t}$ denotes the amount of resource r consumed by task $t \in \mathcal{T}$. Resources can typically represent a set of machines or a set of workers that are assigned to a workstation for performing tasks in this workstation.
- The set of zones $\mathcal{Z}$, where, for each zone $z \in \mathcal{Z}$, $capa_z$ denotes the capacity of z, $cons_{z,t}$ denotes the share of zone z occupied by task $t \in \mathcal{T}$, and $\mathcal{D}_t \subseteq \mathcal{Z}$ is the set of zones that are disabled during the execution of task t.
- The workstation-relative time horizon, for each task during each period $\mathcal{H} = [\![0, c_{max}]\!]$.

Assumptions. An instance of RC-ALBP/L is said to be well-formed if and only if the following assumptions hold. Note that unless said otherwise, all instances considered in the paper are supposed to be well-formed.

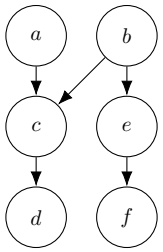
1. The number of periods P is greater than the number of workstations W. Under this assumption, the set of periods $\mathcal{P}$ can be partitioned into two subsets:
   - $\mathcal{P}^U = [\![1, W]\!]$ representing the periods when the line is being filled, called unstable periods.
   - $\mathcal{P}^S = [\![W+1, P]\!]$ the stable periods.

2. The cycle time cannot be changed during the unstable fill-up periods (the first W periods). This assumption is made to prevent an increase in cycle time during these periods, particularly when more tasks are assigned to a downstream workstation. Under this assumption, the cycle time during these periods is equal to the maximum cycle time among them.
3. The cycle time of the unstable periods is greater than the target cycle time.
4. Any task can be allocated to any workstation. The allocation of a task is the same for all periods (no dynamic task allocation or rebalancing allowed).
5. A task cannot be split among workstations (non-preemptive tasks).
6. In each period, all tasks must be completed.
7. The precedence graph $\mathcal{G}$ is acyclic.
8. Any resource can be allocated to any workstation. The amount of each resource allocated to each workstation is the same for all periods (no dynamic resource allocation allowed).

▶ **Example 1.** We consider a toy example, instance of RC-ALBP/L, consisting of 2 workstations, 2 resources $\{r_1, r_2\}$, each with a capacity of 6, and a zone z with a capacity of 1. The target cycle time is 12 and the maximum number of periods for the ramp-up is 6. The maximum value for the cycle time is 20. There are 6 tasks, each with an individual learning curve, i.e. durations based on the number of complete executions of the task. The consumption of resources and zone by each task is represented in Figure 3a. Task c is the only one that disables zone z during its execution. The precedence relationship graph between tasks is represented in Figure 3b.

Task	Consume			Disable	Duration after n complete executions					
	r_1	r_2	z		0	1	2	3	4	5
a	3	1	1	–	9	8	7	6	5	5
b	2	2	1	–	10	7	4	4	4	4
c	1	1	0	z	4	4	4	4	4	4
d	0	3	0	–	8	5	3	3	2	2
e	2	1	1	–	7	6	6	5	5	5
f	1	2	0	–	5	5	4	4	3	3

(a) Tasks involved in the toy example.



(b) Precedence relationship.

Figure 3 Tasks features (a) and precedence relationship (b) for the toy example.

3.2 Assignment, Solution and Optimality

Assignment. An *assignment* (i.e. a solution attempt) of a RC-ALBP/L instance is defined by a tuple $(wks, raw, start, cyc)$, where:

- $wks : \mathcal{T} \to \mathcal{W}$ is a function that maps, for each task $t \in \mathcal{T}$, the workstation $w \in \mathcal{W}$ to which it is allocated;
- $raw : \mathcal{R} \times \mathcal{W} \to \mathbb{N}$ is a function that maps, for each pair composed of a resource $r \in \mathcal{R}$ and a workstation $w \in \mathcal{W}$, the amount of resource r allocated to workstation w;
- $start : \mathcal{T} \times \mathcal{P} \to \mathbb{N}$ is a function that maps, for each task $t \in \mathcal{T}$ and each period $p \in \mathcal{P}$, the start time of t in the workstation to which t is allocated during that period p;
- $cyc : \mathcal{P} \to \mathbb{N}$ is a function that maps, for each period $p \in \mathcal{P}$, its cycle time.

We define an additional function $end : \mathcal{T} \times \mathcal{P} \to \mathbb{N}$ that maps, for each task and each period, the end time in the workstation to which the task is allocated during that period. For each task $t \in \mathcal{T}$ and each period $p \in \mathcal{P}$, the value of this function can be calculated as follows:

$$end(t,p) = start(t,p) + dur_t(p - wks(t))$$

We also have $\mathcal{E}_{t,p} = [\![start(t,p), end(t,p)[\![$ representing the time interval during which the task t is executed in period p.

Solution. An assignment to a RC-ALBP/L instance is a *solution* if and only if: it satisfies the precedence constraints (Equations 1 and 2); respects the resources' capacity (Equations 3 and 4); respects the zones' capacity (Equation 5); satisfies the zone disabling constraints (Equation 6); and respects the characteristics of the cycle time (Equations 7 and 8).

$$\forall (t,u) \in \mathcal{A}, \quad wks(t) \leq wks(u) \tag{1}$$

$$\forall (t,u) \in \mathcal{A}, \forall p \in \mathcal{P}, \quad \big(wks(t) = wks(u)\big) \Rightarrow \big(end(t,p) \leq start(u,p)\big) \tag{2}$$

$$\forall r \in \mathcal{R}, \quad \sum_{w \in \mathcal{W}} raw(r,w) \leq capa_r \tag{3}$$

$$\forall r \in \mathcal{R}, \forall w \in \mathcal{W}, \forall p \in \mathcal{P}, \forall \tau \in \mathcal{H}, \quad \sum_{t \in \mathcal{T} : wks(t) = w \wedge \tau \in \mathcal{E}_{t,p}} cons_{r,t} \leq raw(r,w) \tag{4}$$

$$\forall z \in \mathcal{Z}, \forall w \in \mathcal{W}, \forall p \in \mathcal{P}, \forall \tau \in \mathcal{H}, \quad \sum_{t \in \mathcal{T} : wks(t) = w \wedge \tau \in \mathcal{E}_{t,p}} cons_{z,t} \leq capa_z \tag{5}$$

$$\forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \forall \tau \in \mathcal{H}, \quad (\tau \in \mathcal{E}_{t,p}) \Rightarrow (\mathcal{O}_{u,\tau} = \emptyset) \tag{6}$$

$$\text{with } \mathcal{O}_{u,\tau} = \{u \in \mathcal{T} \setminus \{t\} \mid wks(u) = wks(t), \tau \in \mathcal{E}_{u,p}, \exists z \in \mathcal{D}_t : cons_{z,u} > 0\}$$

$$\forall p \in \mathcal{P}, \quad \max_{t \in \mathcal{T}}(end(t,p)) \leq cyc(p) \tag{7}$$

$$\forall p \in \mathcal{P}, \quad c_{tgt} \leq cyc(p) \leq c_{max} \tag{8}$$

Optimality. A solution to a RC-ALBP/L instance is said to be *optimal* (or Pareto-optimal) if and only if no other solution improves one of the following two optimization criteria without worsening the other: (i) the ramp-up duration (Equation 9), and (ii) the number of adjustments to the line (Equation 10). In this work, we consider that an adjustment to the line is a modification of the cycle time.

$$\sum_{p \in \mathcal{P} : c_{tgt} < cyc(p)} cyc(p) \tag{9}$$

$$\sum_{p \in \mathcal{P}^S} \big(cyc(p-1) \neq cyc(p)\big) \tag{10}$$

▶ **Example 2.** An optimal solution for the example previously introduced is shown in Figure 4. Tasks $\{a, b\}$ are allocated to workstation 1, while tasks $\{c, d, e, f\}$ are allocated to workstation 2. In workstation 1, the order of tasks remains unchanged throughout the production process. In workstation 2, during period 1, no tasks are performed as the first assembled unit is still in workstation 1. There is no adjustment during period 2 due to Assumption 2 (*i.e.* no adjustment in unstable periods). The production rate adjustments are made in periods 3 and 5. In period 3, the task schedule (or the task execution order) remains the same as in period 2, but a decision is made to reduce the cycle time, since the

station time during this period is reduced due to the learning effect. In period 5, a decision is made to change the order of the tasks and further reduce the cycle time to the target cycle time. The ramp-up time in this example is $(19 \times 2) + (15 \times 2) = 68$ and the number of adjustments to the line is 2.
The complete resource profile for this example is available in the Appendix A (Table 4). Note that in this example the available amount of resource r_1 is 6, but only a total of 5 is allocated to both workstations. The reason for this is that the unavailability of zone z (either in use or disabled) has prevented the tasks from running in parallel.

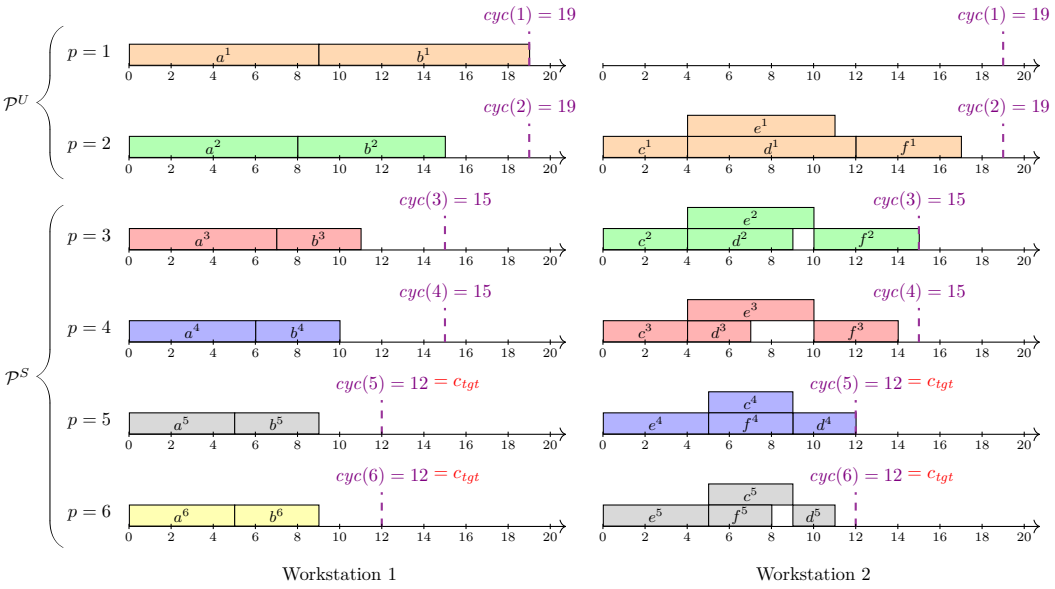


Figure 4 Allocation and scheduling of tasks for the toy example.

4 Encoding RC-ALBP/L in OPL

In this section, we transcribe in OPL the general mathematical model given in Subsection 3.2 for RC-ALBP/L, and identify some dominance-breaking constraints.

4.1 A Base Model for RC-ALBP/L

To start, we propose a base model for RC-ALBP/L. The construction of this model involves both *interval variables* and *state functions* from OPL. An *interval variable* represents a period of time during which a task is carried out, including the start and end times, its duration, and its nature (mandatory or optional). An optional task may be omitted in the computed solution to the problem. Note that although there are no optional tasks in RC-ALBP/L, this feature is still used in the modelling step. The start time, end time and duration of an interval variable are respectively accessible through functions startOf, endOf and lengthOf. A *state function* is a function describing the evolution of a given feature of the environment. In scheduling context, this function can be used to describe the status of a specific resource (in this case, the status of a zone). The possible evolution of this function is constrained by interval variables of the problem using OPL functions (e.g. alwaysEqual).

In order to encode in OPL the general mathematical model described in Subsection 3.2, we define the corresponding decision variables for each of the assignment functions as follows:

- for each task $t \in \mathcal{T}$, $wks(t)$ is represented by an integer variable $\mathbf{wks}_t \in \mathcal{W}$, which indicates the workstation to which t is allocated.
- for each allocated resource $r \in \mathcal{R}$ and each workstation $w \in \mathcal{W}$, $raw(r, w)$ is represented by an integer variable $\mathbf{raw}_{r,w} \in [\![0, capa_r]\!]$, which indicates the amount of r allocated to w.
- for each task $t \in \mathcal{T}$, each period $p \in \mathcal{P}$, and each workstation $w \in \mathcal{W}$, the optional interval variable $\mathbf{itv}_{t,p,w} \in \mathcal{H}$ represents the execution of task t during period p at workstation w. This variable is present only if t is assigned to w, in which case it can express the value returned by $start(t, p)$.
- for each period $p \in \mathcal{P}$, $cyc(p)$ is represented by an integer variable $\mathbf{cyc}_p^{\mathbf{dec}} \in [\![c_{tgt}, c_{max}]\!]$, which indicates the decided cycle time value of p.

In addition, we also introduce the following additional variables that are useful for stating some constraints:

- for each period $p \in \mathcal{P}$, the integer variable $\mathbf{cyc}_p^{\mathbf{req}} \in \mathcal{H}$ represents the minimum required cycle time value of p (or the maximum end time among tasks during p). The value of this variable is always less than or equal to $\mathbf{cyc}_p^{\mathbf{dec}}$.
- for each period $p \in \mathcal{P}$, the integer variable $\mathbf{cost}_p \in \mathcal{H}$ represents the amount of time contributed by p to the ramp-up duration minimisation objective. This variable has the same value as $\mathbf{cyc}_p^{\mathbf{dec}}$ if $\mathbf{cyc}_p^{\mathbf{dec}} > c_{tgt}$, and 0 otherwise.

There are some other additional variables used to encode the zone constraints (or, more specifically, the state function that evolves the status of each zone over time), but these are not described in this section (because they are not related to our main purpose here, which is to express ramp-up cycle time constraints and to express dominance constraints). A complete description of the model in OPL is provided in the Appendix B. The criteria to be minimised are encoded as follows:

$$\sum_{p \in \mathcal{P}} \mathbf{cost}_p \tag{11}$$

$$\sum_{p \in \mathcal{P}^S} (\mathbf{cost}_{p-1} \neq \mathbf{cost}_p) \tag{12}$$

The first objective (Equation 11) is to minimise the time required for the assembly line to reach the target cycle time (or the ramp-up duration). The second objective (Equation 12) is to minimise the number of adjustments to the cycle time. The addressed problem involves a trade-off: to achieve the target cycle time more rapidly, it is necessary to reduce the cycle time of the line as quickly and as extensively as possible, which results in an increase in the number of adjustments to the cycle time.

In this section, we do not elaborate on how each constraint is encoded using the decision variables mentioned earlier. In fact, most of these constraints are quite standard in scheduling problems or have already been discussed in the literature on assembly lines with CP [26, 33, 34]. However, we provide the constraints associated with the ramp-up cycle time variables, which are novel in this work.

$$\forall p \in \mathcal{P}, \quad \mathbf{cyc}_p^{\mathbf{req}} = \max_{t \in \mathcal{T}, w \in \mathcal{W}} \mathsf{endOf}(\mathbf{itv}_{t,p,w}, 0) \tag{13}$$

$$\forall p \in \mathcal{P}^U, \quad \mathbf{cyc}_p^{\mathbf{dec}} = \max_{q \in \mathcal{P}^U} \mathbf{cyc}_q^{\mathbf{req}} \tag{14}$$

$$\forall p \in \mathcal{P}^S, \quad \mathbf{cyc}_p^{\mathbf{dec}} \geq \mathbf{cyc}_p^{\mathbf{req}} \tag{15}$$

$$\forall p \in \mathcal{P}^U, \quad \mathbf{cost}_p = \mathbf{cyc}_p^{\mathbf{dec}} \tag{16}$$

$$\forall p \in \mathcal{P}^S, \quad (\mathbf{cyc}_p^{\mathbf{dec}} > c_{tgt}) \Longrightarrow (\mathbf{cost}_p = \mathbf{cyc}_p^{\mathbf{dec}}) \tag{17}$$

$$\forall p \in \mathcal{P}^S, \quad (\mathbf{cyc}_p^{\mathbf{dec}} = c_{tgt}) \Longrightarrow (\mathbf{cost}_p = 0) \tag{18}$$

The constraints (13) compute the required cycle time for each period. Constraints (14) ensure that the decided cycle time during the unstable periods is equal to the maximum required cycle time among them and cannot be altered during these periods. Constraints (15) express that the decided cycle time during the stable periods is greater than or equal to the required cycle time. Next, the constraints (16), (17) and (18) calculate, for each period, the time contributed to the ramp-up stage. A comprehensive description of this base model with all the constraints is provided in the Appendix B.

4.2 Dominance-Breaking Constraints

We now introduce some dominance-breaking constraints [13] for RC-ALBP/L. With $\mathcal{S}$ representing the solution space of RC-ALBP/L, we can define two functions $obj_{ru} : \mathcal{S} \to \mathbb{N}$ and $obj_{na} : \mathcal{S} \to \mathbb{N}$ that return the value of the objective calculated by Equation 9 and Equation 10, respectively.

▶ **Definition 3** (Dominance relation). *For any two solutions σ and π, a dominance relation $\preceq$ is an incomplete, reflexive and transitive relation between σ and π, which expresses the following conditions:*

$$\pi \preceq \sigma \Longleftrightarrow (obj_{ru}(\pi) \leq obj_{ru}(\sigma)) \wedge (obj_{na}(\pi) \leq obj_{na}(\sigma))$$

The aim of the dominance-breaking constraints is to reduce the solution space from $\mathcal{S}$ to a solution space $\mathcal{S}^{dom} \subseteq \mathcal{S}$ without changing the quality of the optimal solutions. In brief, for every solution σ removed by the dominance-breaking constraints, there exists a solution π remaining that is at least as good as σ in terms of both objectives. Formally:

$$\forall \sigma \in \mathcal{S} \setminus \mathcal{S}^{dom}, \exists \pi \in \mathcal{S}^{dom} : \pi \preceq \sigma$$

Next, we provide a detailed description of each dominance-breaking constraint. For convenience, in the remainder of this paper, for any assignment σ (see Subsection 3.2) we denote by $x(\sigma)$ the value of a decision variable x corresponding to σ (e.g. $\mathbf{cyc}_p^{\mathbf{dec}}(\sigma)$). Similarly, we denote by f_σ the value returned by an assignment function f corresponding to σ (e.g. $start_\sigma(t, p)$).

▶ **Definition 4** (p-excluded equivalence). *For any two assignments σ and π, for any period $p \in \mathcal{P}$, a p-excluded equivalence (noted $\overset{p}{\sim}$) is a symmetrical, reflexive and transitive relation between σ and π such that all of the following conditions are satisfied:*

- $\forall t \in \mathcal{T},\ wks_\pi(t) = wks_\sigma(t)$;
- $\forall r \in \mathcal{R},\ \forall w \in \mathcal{W},\ raw_\pi(r, w) = raw_\sigma(r, w)$;
- $\forall t \in \mathcal{T},\ \forall q \in \mathcal{P} \setminus \{p\},\ start_\pi(t, q) = start_\sigma(t, q)$;
- $\forall q \in \mathcal{P} \setminus \{p\},\ cyc_\pi(q) = cyc_\sigma(q)$.

Put simply, Definition 4 means that if $\sigma \overset{p}{\sim} \pi$, then the only decisions that may differ between σ and π are the tasks schedule during period p and the cycle time of p. In the following of this section, we focus on creating dominance-breaking rules that enforce *cyc* to become a monotonically decreasing function, which is not necessarily the case for non-optimal solutions.

▶ **Lemma 5.** *Let σ be a solution and p a period in $\mathcal{P}^S$ such that $\mathbf{cyc}^{\mathbf{req}}_{p-1}(\sigma) < \mathbf{cyc}^{\mathbf{req}}_{p}(\sigma)$. There exists a solution $\pi \preceq \sigma$ such that $\mathbf{cyc}^{\mathbf{req}}_{p-1}(\pi) \geq \mathbf{cyc}^{\mathbf{req}}_{p}(\pi)$.*

Proof. Let π an assignment such that $\sigma \overset{p}{\sim} \pi$ and for any task $t \in \mathcal{T}$, $start_\pi(t, p-1) = start_\pi(t, p)$. As $dur_t(p-1) \geq dur_t(p)$ (remember that dur_t is a monotonically decreasing function), then $end_\pi(t, p-1) \geq end_\pi(t, p)$ and thus $\mathbf{cyc}^{\mathbf{req}}_{p-1}(\pi) \geq \mathbf{cyc}^{\mathbf{req}}_{p}(\pi)$ (see Figure 5). Since σ is a solution and $\sigma \overset{p}{\sim} \pi$, then all constraints are satisfied in π for periods in $\mathcal{P} \setminus \{p\}$. As i) the start times of tasks during period p in π remain the same as during $p-1$, and ii) the tasks during p are finished earlier than during $p-1$, then all precedence, resource and zone constraints will be naturally satisfied (they are relaxed).
Since $\mathbf{cyc}^{\mathbf{req}}_{p}(\pi) \leq \mathbf{cyc}^{\mathbf{req}}_{p-1}(\pi) = \mathbf{cyc}^{\mathbf{req}}_{p-1}(\sigma) < \mathbf{cyc}^{\mathbf{req}}_{p}(\sigma) \leq \mathbf{cyc}^{\mathbf{dec}}_{p}(\sigma)$, then we can select $\mathbf{cyc}^{\mathbf{dec}}_{p}(\pi)$ such that $\mathbf{cyc}^{\mathbf{dec}}_{p}(\pi) \leq \mathbf{cyc}^{\mathbf{dec}}_{p}(\sigma) \leq c_{max}$, which satisfies the cycle time characteristics and makes π a solution. As $\sigma \overset{p}{\sim} \pi$, we have $\mathbf{cyc}^{\mathbf{dec}}_{q}(\pi) = \mathbf{cyc}^{\mathbf{dec}}_{q}(\sigma)$ for all $q \in \mathcal{P} \setminus \{p\}$. Two possibilities exist:

- if p is not an adjustment period in σ, then simply selecting $\mathbf{cyc}^{\mathbf{dec}}_{p}(\pi) = \mathbf{cyc}^{\mathbf{dec}}_{p}(\sigma)$ will make $\pi \preceq \sigma$.
- if p is an adjustment period in σ, then any possible value of $\mathbf{cyc}^{\mathbf{dec}}_{p}(\pi)$ such that $\mathbf{cyc}^{\mathbf{dec}}_{p}(\pi) < \mathbf{cyc}^{\mathbf{dec}}_{p}(\sigma)$ will make $\pi \preceq \sigma$.

We have $\pi \preceq \sigma$ in both cases. Hence, the lemma is proved. ◀

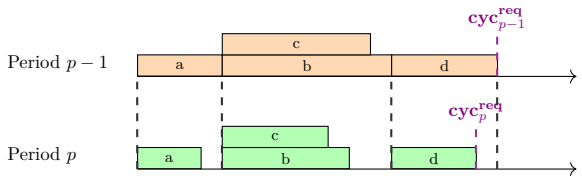


Figure 5 Illustration of Lemma 5's proof: the task intervals during period p in π are covered by those of period $p-1$.

▶ **Lemma 6.** *Let σ be a solution and p an adjustment period in $\mathcal{P}^S$ such that $\mathbf{cyc}^{\mathbf{dec}}_{p-1}(\sigma) < \mathbf{cyc}^{\mathbf{dec}}_{p}(\sigma)$. There exists a solution $\pi \preceq \sigma$ such that $\mathbf{cyc}^{\mathbf{dec}}_{p-1}(\pi) \geq \mathbf{cyc}^{\mathbf{dec}}_{p}(\pi)$.*

Proof. There are two possible cases: (a) $\mathbf{cyc}^{\mathbf{req}}_{p-1}(\sigma) \geq \mathbf{cyc}^{\mathbf{req}}_{p}(\sigma)$; and (b) $\mathbf{cyc}^{\mathbf{req}}_{p-1}(\sigma) < \mathbf{cyc}^{\mathbf{req}}_{p}(\sigma)$.
In case (a), since $\mathbf{cyc}^{\mathbf{req}}_{p}(\sigma) \leq \mathbf{cyc}^{\mathbf{req}}_{p-1}(\sigma) \leq \mathbf{cyc}^{\mathbf{dec}}_{p-1}(\sigma)$, we can construct a solution π from σ by simply selecting any possible value of $\mathbf{cyc}^{\mathbf{dec}}_{p}(\pi)$ within the interval $[\![\mathbf{cyc}^{\mathbf{req}}_{p}(\sigma), \mathbf{cyc}^{\mathbf{dec}}_{p-1}(\sigma)]\!]$, while keeping all other decisions from σ unchanged. The solution π obtained satisfies $\pi \preceq \sigma$.
In case (b), based on Lemma 5 proved above, we can construct a solution σ' such that $\mathbf{cyc}^{\mathbf{req}}_{p-1}(\sigma') \geq \mathbf{cyc}^{\mathbf{req}}_{p}(\sigma')$ and $\sigma' \preceq \sigma$. From here, the situation reverts back to case (a) mentioned earlier. We follow the procedure in case (a) and construct a solution π such that $\pi \preceq \sigma'$, and therefore $\pi \preceq \sigma$.
In both cases, we can construct a solution $\pi \preceq \sigma$. Hence, the lemma is proved. ◀

With Lemma 5 and Lemma 6 provided above, we can solely consider solutions where *cyc* is monotonically decreasing. We conclude this section with another dominance-breaking rule, which ensures that the minimum possible cycle time is selected in each adjustment period.

▶ **Lemma 7.** *Let σ be a solution and p an adjustment period in $\mathcal{P}^S$ such that $\mathbf{cyc}_{p-1}^{\mathbf{dec}}(\sigma) > \mathbf{cyc}_p^{\mathbf{dec}}(\sigma)$ and $\mathbf{cyc}_p^{\mathbf{dec}}(\sigma) > \mathbf{cyc}_p^{\mathbf{req}}(\sigma)$. There exists a solution $\pi \preceq \sigma$ such that $\mathbf{cyc}_p^{\mathbf{dec}}(\pi) = \mathbf{cyc}_p^{\mathbf{req}}(\pi)$.*

Proof. Since $\mathbf{cyc}_p^{\mathbf{dec}}(\sigma) > \mathbf{cyc}_p^{\mathbf{req}}(\sigma)$, we can construct a solution π from σ by simply selecting the value of $\mathbf{cyc}_p^{\mathbf{dec}}(\pi) = \mathbf{cyc}_p^{\mathbf{req}}(\sigma)$, while keeping all other decisions from σ unchanged. As the ramp-up time in π is reduced, we have $\pi \preceq \sigma$. ◀

From all the proved lemma above, we can safely introduce the following dominance-breaking constraints for RC-ALBP/L:

$$\forall p \in \mathcal{P}^S, \qquad \mathbf{cyc}_{p-1}^{\mathbf{req}} \geq \mathbf{cyc}_p^{\mathbf{req}} \tag{19}$$

$$\forall p \in \mathcal{P}^S, \qquad \mathbf{cyc}_{p-1}^{\mathbf{dec}} \geq \mathbf{cyc}_p^{\mathbf{dec}} \tag{20}$$

$$\forall p \in \mathcal{P}^S, \qquad (\mathbf{cyc}_{p-1}^{\mathbf{dec}} > \mathbf{cyc}_p^{\mathbf{dec}}) \Longrightarrow (\mathbf{cyc}_p^{\mathbf{dec}} = \mathbf{cyc}_p^{\mathbf{req}}) \tag{21}$$

The constraints (19) and (20) ensure that both the required and decided cycle times are never increasing during the stable periods, according to Lemma 5 and 6. Finally, constraints (21) guarantee that whenever an adjustment is made, the smallest possible value is selected for the decided cycle time according to Lemma 7.

5 Experimentation

In this section, we present the results of experiments conducted on real-world benchmarks. The details of the benchmarks are provided first, followed by an analysis of the results.

5.1 Benchmarks

Learning Curves. In this work, we use the log-linear learning curve with a steady learning state [38, 39, 19]. Such a log-linear curve is the classical model used in the aeronautical industry. We define for each task $t \in \mathcal{T}$: (a) the duration for the first execution $dur_t^0 = dur_t(0)$; (b) the duration when the learning effect reaches its steady state dur_t^∞; and (c) the learning rate $l_t \in]0, 1]$ which is a parameter that determines the slope of the learning curve. The duration function, based on the number of complete executions n for each task t, is given by the following formula:

$$dur_t(n) = dur_t^\infty + \left\lceil (dur_t^0 - dur_t^\infty) \cdot (n+1)^{\log_2 l_t} \right\rceil, \quad \forall n \in \mathbb{N}$$

In our experiments, the ratio dur_t^0 / dur_t^∞ lies between 2 and 10.

Dataset. The experiments are conducted on three different datasets that come from real aircraft assembly lines [34]. We have adapted these datasets to account for the learning effect (they are also made available online for further research at [25]). Each dataset consists of 187, 199, and 795 tasks, respectively. The learning rate for each task is randomly selected from the interval $[0.45, 1]$, which is considered a realistic range. Note that using the actual values would not affect the applicability of our approach. For each dataset, we create two instances by varying the number of workstations, the number of periods and the cycle time features as showed in Table 1. A total of six instances were considered. In the remainder of this section, we denote by `T-W-P` the instance consisting of T tasks, W workstations and P periods (e.g. `199-2-26`).

Table 1 Features of data instances: number of tasks, number of workstations, number of periods, target cycle time, maximum value of cycle time, number of resources, capacity of resources, number of zones, capacity of zones, and learning rate.

T	W	P	c_{tgt}	c_{max}	$\lvert\mathcal{R}\rvert$	$capa_r$	$\lvert\mathcal{Z}\rvert$	$capa_z$	l_t
$\{187, 199, 795\}$	2	26	2880	28800	6	≤ 40	48	1	$[0.45, 1]$
	4	50	1440	14400					

Solving Approaches. Due to Assumption 2, the value returned by obj_{na} (see Subsection 4.2) for an instance consisting of W workstations and P periods lies in $[\![1, P - W]\!]$. According to Lemma 6, we do not consider an increase in cycle time as part of a possible adjustment to the line, as every solution exhibiting this behaviour is always dominated by a solution with monotonically decreasing cycle time. Therefore, we can transform the original bi-objectives problem into a series of $P - W$ mono-objective sub-problems. For each $m \in [\![1, P - W]\!]$, there is such a sub-problem with constraint (22) and the ramp-up objective (Equation 11).

$$\sum_{p \in \mathcal{P}^S} (\mathbf{cost}_{p-1} > \mathbf{cost}_p) \leq m \tag{22}$$

Such constraints express that the maximum number of adjustments to the cycle time, until the target cycle time is reached, is equal to m. In the experiments, we perform a series of solving attempts (called *runs*) of the above mono-objective sub-problems using three different solving approaches. Each run is defined by a triplet (I, m, A), where I is an instance from the datasets, $m \in [\![1, P - W]\!]$, and A is one of the three following possible solving approaches:

1. CPO (*baseline solving approach*): the base model, as described in Subsection 4.1, is used.
2. CPO^{DB} (*dominance-breaking solving approach*): the *enriched* model, which corresponds to the base model together with the dominance-breaking constraints (Equation 19, Equation 20 and Equation 21), is used.
3. CPO^{WS} (*warm-start solving approach*): the enriched model is used (as for CPO^{DB}), but the solving process involves a kind of incremental mechanism. The aim of CPO^{WS} is to directly generate a set of Pareto-optimal solutions. Actually, it can be easily proved that any solution to an instance I with the upper-bound value $m - 1$ is also a solution for I with the upper-bound value m. Formally, for all $m > 1$, the final solution returned by $(I, m - 1, \text{CPO}^{\text{WS}})$ is used to warm-start $(I, m, \text{CPO}^{\text{WS}})$, which means that CPO^{WS} can only improve the value of the ramp-up objective obj_{ru} when m increases.

The numbers of variables and constraints considered in the base model are given in Table 2. As the number of tasks increases, the numbers of variables and constraints both grow at an approximately proportional rate. The same phenomenon occurs when we increase the number of workstations and periods (doubling them results in nearly quadrupling the number of variables and constraints).

For comparison purposes, we also evaluated an ILP approach. The chosen ILP formulation is a discrete-time, time-indexed variable encoding. Other MILP encodings exist in the context of resource-constrained scheduling, using sequencing or event-based variables [5]. However, in presence of learning effect, task durations become variables, making those encodings non-linear (they require multiplying decision variables by task durations). Our ILP approach is not described in the main paper because of its poor performance. In fact, due to the explosion in the number of variables and constraints (e.g., 560 million variables and 80

Table 2 Number of variables and constraints in the base model (CPO) for each instance.

Instances	187-2-26	199-2-26	795-2-26	187-4-50	199-4-50	795-4-50
# Variables	12,497	13,133	44,721	47,361	49,773	169,569
# Constraints	61,176	75,524	235,839	234,328	288,932	901,831

million constraints for the smallest instance, 187-2-26; see Table 5 in the Appendix for other instances), the ILP approach cannot build a relaxed version of the model after running for 2 hours, and even causes the test computer to run out of memory. Details of this approach are provided in the Appendix C.

Test Setup. The tests have been launched using IBM CP Optimizer 22.1.2 for the CP-based approaches and IBM CPLEX 22.1.2 for the ILP approach [24] through the DOcplex API, on Intel® Xeon® CPU E5-2660v3 2.60-3.30 GHz with 62 GB of RAM. We tried several search strategies for all CP-based approaches (e.g., priority branching on cycle time variables, selecting values of variable domains in increasing order, etc.), however, the results obtained were worse than those of the default strategy. Therefore, we chose to retain the default setting. As the number of variables and constraints increases significantly with the number of workstations and periods, the timeout for each run of a 2-workstation instance was set to 900 seconds (15 minutes), and for a 4-workstation instance, it was set to 7,200 seconds (2 hours).

5.2 Results

The results are presented in Table 3. More precisely, for each instance I, we provide a figure representing the results of runs (I, m, A), for all $m \in [\![1, P - W]\!]$ and for all approaches A. In such figures, each point (circle • for CPO, triangle ▼ for $\mathsf{CPO}^{\mathsf{DB}}$, and star ★ for $\mathsf{CPO}^{\mathsf{WS}}$) corresponds to a solution for which the vertical axis represents the ramp-up duration (objective obj_{ru}) and the horizontal axis represents the *actual* number of cycle time adjustments (objective obj_{na}). Note that it is possible to have multiple ramp-up duration values for a given number of adjustments. This means that several values of m (upper bound for objective obj_{na}) have permitted to reach the same value of obj_{na}. Non-dominated and dominated solutions (defined relative to each solving approach) are respectively represented with big-coloured and small-black points. Note that solutions which are non-dominated within a given approach may become dominated in a competitive cross-approach comparison.

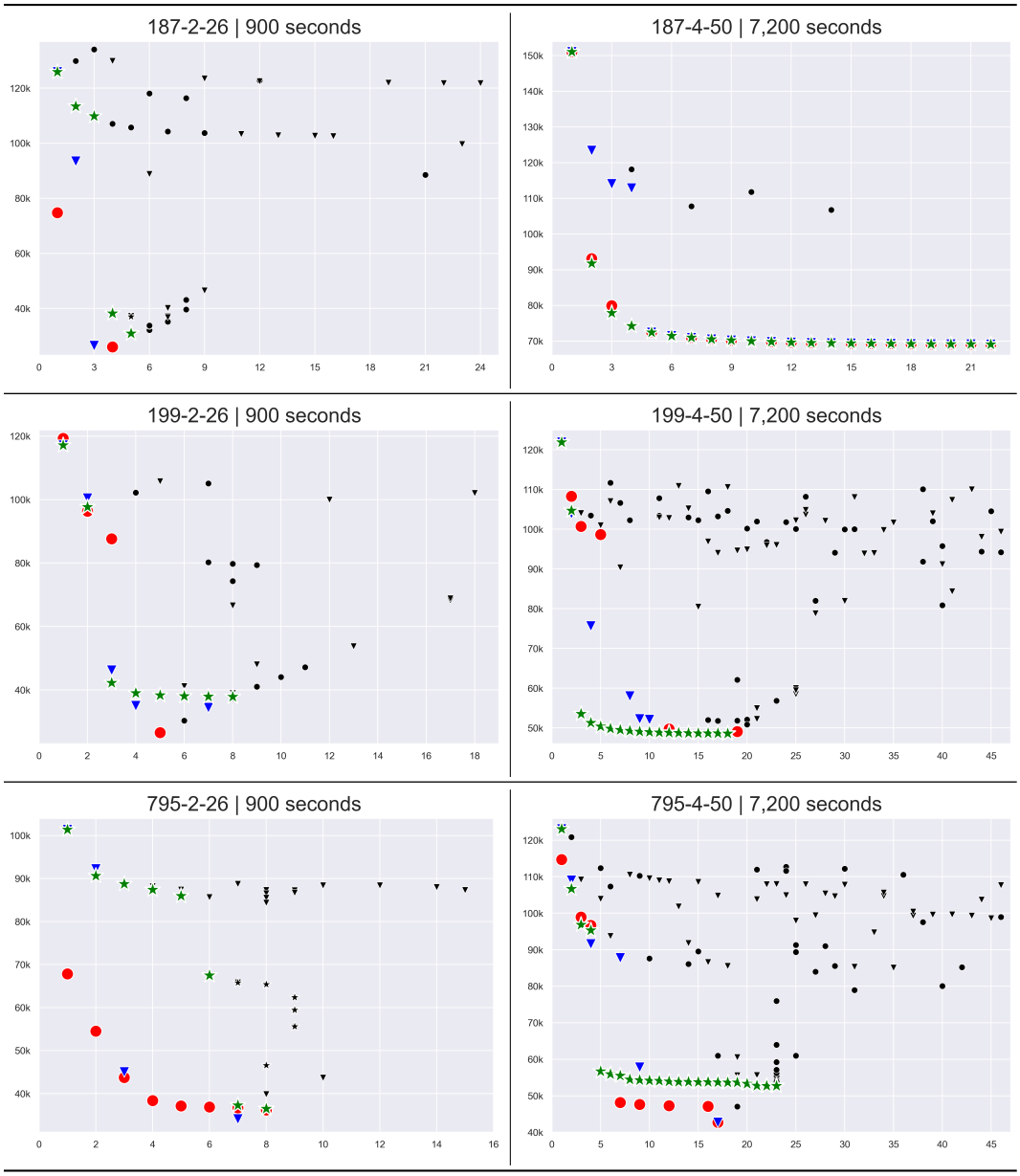
We begin our analysis by comparing CPO and $\mathsf{CPO}^{\mathsf{DB}}$, and we discuss next the results obtained with $\mathsf{CPO}^{\mathsf{WS}}$.

5.2.1 Comparison Between CPO and $\mathsf{CPO}^{\mathsf{DB}}$

The results corresponding to the solving approaches CPO and $\mathsf{CPO}^{\mathsf{DB}}$ are represented by circle • and triangle ▼ in Table 3. There is no clear difference between CPO and $\mathsf{CPO}^{\mathsf{DB}}$ in terms of solution quality. Both yield a GAP[2] value that exceed 60% in the majority of runs. The figures (embedded in the cells of the table) show that, with these two solving approaches,

[2] The difference between the objective value of a given solution σ and a lower bound LB computed by the solver, typically expressed as a percentage. $\text{GAP}(\sigma) = {}^{(obj_{ru}(\sigma) - LB)}/_{obj_{ru}(\sigma)}$

Table 3 Comparison between the results of all solving approaches (circle • for CPO, triangle ▼ for CPO^{DB}, and star ★ for CPO^{WS}). Each bigger coloured point represents a non-dominated solution, whereas each small black point corresponds to a dominated one. The vertical axis represents the ramp-up duration, and the horizontal axis represents the actual number of cycle time adjustments.



a greater upper bound for the number of cycle time adjustments (objective obj_{na}) does not guarantee to get a better ramp-up value (objective obj_{ru}) within the given timeouts, except for instance `187-4-50`.

Note that optimality was proved for both solving approaches on some runs concerning the instance `187-4-50`, using a very small effective tolerance (CPO yielding a slightly greater number of proved optimal solutions, yet). In most cases, CPO and CPO^{DB} yields a similar number of competitive non-dominated solutions. However, an exception is observed for

the 795-tasks dataset, particularly for the instance 795-2-26, where CPO significantly outperforms CPO^{DB}. Unfortunately, it is worthwhile to note that some solutions produced by CPO within the allotted time violate Lemma 6, *i.e.* there are some periods p for which $\mathbf{cyc}^{\mathbf{dec}}_{p-1} < \mathbf{cyc}^{\mathbf{dec}}_{p}$. As such solutions would not be acceptable in an industrial setting, it means that CPO would require a post-processing phase in order to satisfy Lemma 6, whereas CPO^{DB} provides directly acceptable solutions, even with short timeout values.

5.2.2 Results Obtained with CPO^{WS}

We recall that the aim of the solving approach CPO^{WS} is to directly generate a Pareto-front, as neither CPO nor CPO^{DB} are suited to do so within the specified timeouts except for instances 187-4-50 and 795-2-26.

The results show that CPO^{WS} (represented by star ★ in Table 3) is actually capable of generating such a Pareto-front for most instances. Moreover, the Pareto-front obtained with CPO^{WS} contains more points than the other solving approaches, which allows a finer analysis of the results. For example, for instance 187-4-50, CPO^{WS} allows us to evaluate the gain of each additional cycle time adjustment for $m \in [\![1, 6]\!]$, whereas solutions computed with CPO for $m = 4$ or $m = 6$ are dominated. The optimal ramp-up time is also proved only for this instance, under a very small effective tolerance. CPO^{WS} also yields a lower average GAP value compared to the other solving approaches.

That said, CPO^{WS} has some drawbacks. First, warm-starting from the solution of the previous run might guide the solver into a part of the search space with lower quality solutions. This is for example the case for instance 795-2-26 where better solutions are found by CPO and CPO^{DB} for low values of m. Another drawback of CPO^{WS} lies in the requirement for a sequential execution of all runs, which may lead to very significant extended computational time, whereas each run in CPO and CPO^{DB} can be executed in parallel. However, we also observe that most of the descent in the search tree is concentrated in the first few runs, where solutions improve frequently. This suggests that computational time could probably be reduced by executing the first few runs with the full time allowance, and decreasing the timeout for subsequent runs. However, setting a value to these different time-outs might be strongly linked to the instances and we have not explored that option in this work.

6 Conclusion

In this work, we have a) formally defined the RC-ALBP/L, b) introduced an OPL encoding while providing some dominance-breaking rules for this problem, and c) conducted an experimentation on industrial aircraft assembly line benchmarks using three different CP-based solving approaches.

Since RC-ALBP/L involves a trade-off between ramp-up time and the number of line adjustments, the aim of this work is to generate a Pareto-front, which can be achieved using our solution approaches within a reasonable time limit. These results can provide valuable insights for manufacturers in the design and planning of assembly lines – not only in the aircraft industry, but in production more broadly – especially as minimising ramp-up time is receiving growing attention from them.

In future research, we could explore other solving methods, such as MaxSAT or meta-heuristics. We could also investigate scenarios in which the learning effect is transferable across similar tasks, for example installing the left and right wings of an aircraft. Exploring the impact of uncertainty in learning efficiency would allow us to handle robustness issues. Finally, it would be possible to consider that resources are not all available at the beginning of the ramp-up, but are gradually added, therefore adding another level of decision.

References

1 Z.Z. El Abidine and T. Koltai. The effect of learning on assembly line balancing: A review. *Periodica Polytechnica Social and Management Sciences*, 2023.

2 K. Ağpak and H. Gökçen. Assembly line balancing: Two resource constrained cases. *International Journal of Production Economics*, 96(1):129–140, 2005.

3 H.M. Alakaş, M. Pınarbaşı, and M. Yüzükırmızı. Constraint programming model for resource-constrained assembly line balancing problem. *Soft Computing*, 24:5367–5375, 2020. doi:10.1007/S00500-019-04294-8.

4 E. Alhomaidhi. Enhancing efficiency and adaptability in mixed model line balancing through the fusion of learning effects and worker prerequisites. *International Journal of Industrial Engineering Computations*, 15(2):541–552, 2024.

5 C. Artigues, S. Hartmann, and M. Vanhoucke. Fifty years of research on resource-constrained project scheduling explored from different perspectives. *European Journal of Operational Research*, 2025.

6 Z. Bao and L. Chen. A matheuristic approach for the aircraft final assembly line balancing problem considering learning curve. In *2023 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pages 1389–1393. IEEE, 2023. doi:10.1109/IEEM58616.2023.10407009.

7 O. Battaïa and A. Dolgui. A taxonomy of line balancing problems and their solution approaches. *International Journal of Production Economics*, 142(2):259–277, 2013.

8 O. Battaïa and A. Dolgui. Hybridizations in line balancing problems: A comprehensive review on new trends and formulations. *International Journal of Production Economics*, 250:108673, 2022.

9 D. Biskup. A state-of-the-art review on scheduling with learning effects. *European Journal of Operational Research*, 188(2):315–329, 2008. doi:10.1016/J.EJOR.2007.05.040.

10 N. Boysen, P. Schulze, and A. Scholl. Assembly line balancing: What happened in the last fifteen years? *European Journal of Operational Research*, 301(3):797–814, 2022. doi:10.1016/J.EJOR.2021.11.043.

11 J. Bukchin and M. Masin. Multi-objective design of team oriented assembly systems. *European Journal of Operational Research*, 156(2):326–352, 2004. doi:10.1016/S0377-2217(03)00054-7.

12 R. Bultó, E. Viles, and R. Mateo. Overview of ramp-up curves: A literature review and new challenges. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 232(5):755–765, 2018.

13 G. Chu and P. Stuckey. Dominance breaking constraints. *Constraints*, 20:155–182, 2015. doi:10.1007/S10601-014-9173-7.

14 Z.A. Çil and D. Kizilay. Constraint programming model for multi-manned assembly line balancing problem. *Computers & Operations Research*, 124:105069, 2020. doi:10.1016/J.COR.2020.105069.

15 Y. Cohen, G. Vitner, and S.C. Sarin. Optimal allocation of work in assembly lines for lots with homogenous learning. *European Journal of Operational Research*, 168(3):922–931, 2006. Balancing Assembly and Transfer lines. doi:10.1016/J.EJOR.2004.07.037.

16 A. Corominas, L. Ferrer, and R. Pastor. Assembly line balancing: general resource-constrained case. *International Journal of Production Research*, 49(12):3527–3542, 2011.

17 S.G. Dimitriadis. Assembly line balancing and group working: A heuristic procedure for workers' groups operating on the same product and workstation. *Computers & Operations Research*, 33(9):2757–2774, 2006. Part Special Issue: Anniversary Focused Issue of Computers & Operations Research on Tabu Search. doi:10.1016/J.COR.2005.02.027.

18 E.B. Edis. Constraint programming approaches to disassembly line balancing problem with sequencing decisions. *Computers & Operations Research*, 126:105111, 2021. doi:10.1016/J.COR.2020.105111.

19 E.H. Grosse, C.H. Glock, and S. Müller. Production economics and the learning curve: A meta-analysis. *International Journal of Production Economics*, 170:401–412, 2015.

20 W.B. Helgeson and D.P. Birnie. Assembly line balancing using the ranked positional weight technique. *Journal of industrial engineering*, 12(6):394–398, 1961.

21 J. Heraud, K. Medini, and A.L. Andersen. Managing agile ramp-up projects in manufacturing–status quo and recommendations. *CIRP Journal of Manufacturing Science and Technology*, 45:125–137, 2023.

22 A. Hill, J. Ticktin, and T.W.M. Vossen. A computational study of constraint programming approaches for resource-constrained project scheduling with autonomous learning effects. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5–8, 2021, Proceedings 18*, pages 26–44. Springer, 2021. doi:10.1007/978-3-030-78230-6_2.

23 T. Koltai and N. Kalló. Analysis of the effect of learning on the throughput-time in simple assembly lines. *Computers & Industrial Engineering*, 111:507–515, 2017. doi:10.1016/J.CIE.2017.03.034.

24 P. Laborie, J. Rogerie, P. Shaw, and P. Vilím. IBM ILOG CP optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG. *Constraints*, 23:210–250, 2018. doi:10.1007/S10601-018-9281-X.

25 D.A. Le, S. Roussel, and C. Lecoutre. Dataset for the assembly line balancing problem with learning effect, 2025. doi:10.57745/EBDN5W.

26 D.A. Le, S. Roussel, C. Lecoutre, and A. Chan. Learning Effect and Compound Activities in High Multiplicity RCPSP: Application to Satellite Production. In *30th International Conference CP*, 2024. doi:10.4230/LIPIcs.CP.2024.18.

27 Y. Li and T.O. Boucher. Assembly line balancing problem with task learning and dynamic task reassignment. *The International Journal of Advanced Manufacturing Technology*, 88:3089–3097, 2017.

28 Y. Li, D. Liu, and I. Kucukkoc. Mixed-model assembly line balancing problem considering learning effect and uncertain demand. *Journal of Computational and Applied Mathematics*, 422:114823, 2023. doi:10.1016/J.CAM.2022.114823.

29 D. Ogan and M. Azizoglu. A branch and bound method for the line balancing problem in u-shaped assembly lines with equipment requirements. *Journal of Manufacturing Systems*, 36:46–54, 2015.

30 C. Otto and A. Otto. Extending assembly line balancing problem by incorporating learning effects. *International Journal of Production Research*, 52(24):7193–7208, 2014.

31 V. Van Peteghem and M. Vanhoucke. A genetic algorithm for the preemptive and non-preemptive multi-mode resource-constrained project scheduling problem. *European Journal of Operational Research*, 201(2):409–418, 2010. doi:10.1016/J.EJOR.2009.03.034.

32 V. Van Peteghem and M. Vanhoucke. Influence of learning in resource-constrained project scheduling. *Computers & Industrial Engineering*, 87:569–579, 2015. doi:10.1016/J.CIE.2015.06.007.

33 X. Pucel and S. Roussel. Constraint Programming Model for Assembly Line Balancing and Scheduling with Walking Workers and Parallel Stations. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 307, pages 23:1–23:21, 2024. doi:10.4230/LIPICS.CP.2024.23.

34 S. Roussel, T. Polacsek, and A. Chan. Assembly line preliminary design optimization for an aircraft. In *CP 2023 (The 29th International Conference on Principles and Practice of Constraint Programming)*, 2023. doi:10.4230/LIPIcs.CP.2023.32.

35 K.E. Stecke and M. Mokhtarzadeh. Balancing collaborative human–robot assembly lines to optimise cycle time and ergonomic risk. *International Journal of Production Research*, 60(1):25–47, 2022. doi:10.1080/00207543.2021.1989077.

36 P. van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

37 J.M. Wilson. Formulation of a problem involving assembly lines with multiple manning of work stations. *International Journal of Production Research*, 24(1):59–63, 1986.

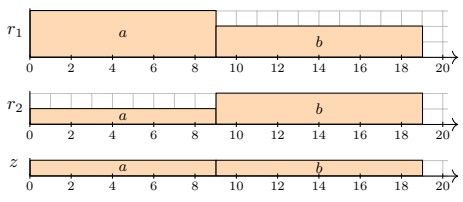
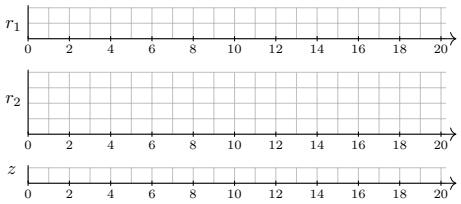
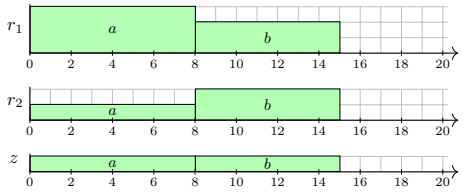
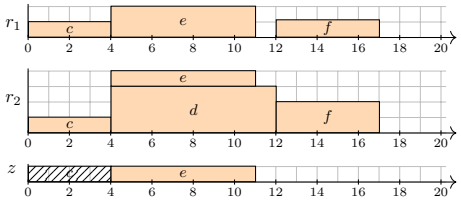
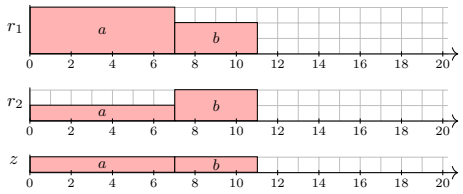
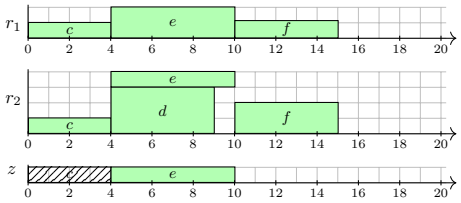
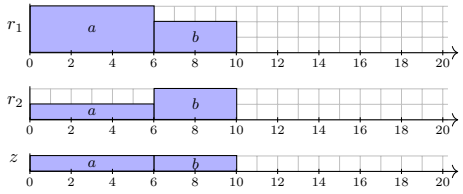
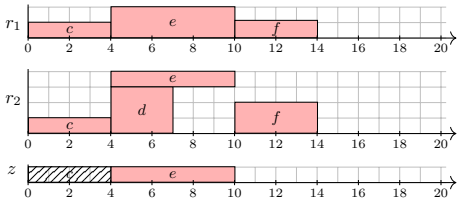
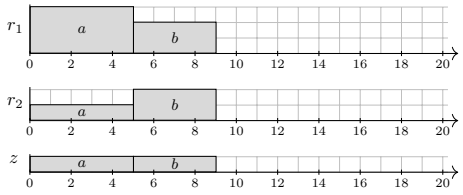
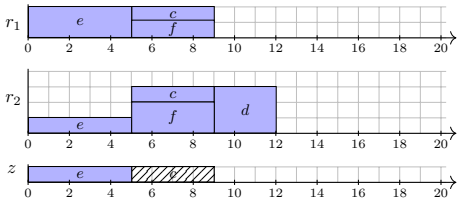
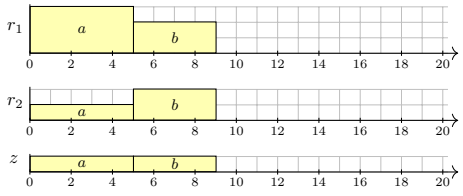
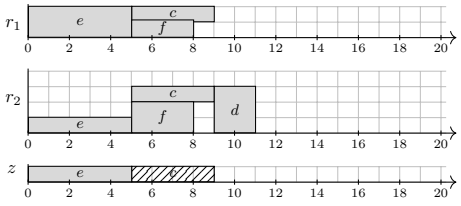
38 T.P. Wright. Factors affecting the cost of airplanes. *Journal of the aeronautical sciences*, 3(4):122–128, 1936.

39 L.E. Yelle. The learning curve: Historical review and comprehensive survey. *Decision sciences*, 10(2):302–328, 1979.

40 Z.A. ÇİL. An exact solution method for multi-manned disassembly line design with and/or precedence relations. *Applied Mathematical Modelling*, 99:785–803, 2021. doi:10.1016/j.apm.2021.07.013.

A Resource Profile of Example 2

Table 4 Resource profiles of workstations 1 and 2 (WS1 and WS2) of Example 2.

Period	Resource profile WS1	Resource profile WS2
1		
2		
3		
4		
5		
6		

B A Base Model in OPL for RC-ALBP/L (Full Version)

The decision variables are the following:

- For each task $t \in \mathcal{T}$, the integer variable $\mathbf{wks}_t \in [\![1, W]\!]$ indicates the workstation to which t is allocated.
- For each allocated resource $r \in \mathcal{R}$ and each workstation $w \in \mathcal{W}$, the integer variable $\mathbf{raw}_{r,w} \in [\![0, capa_r]\!]$ indicates the amount of r allocated to w.
- For each task $t \in \mathcal{T}$, each period $p \in \mathcal{P}$ and each workstation $w \in \mathcal{W}$, the optional interval variable $\mathbf{itv}_{t,p,w} \in \mathcal{H}$ represents the execution of task t during the period p at the workstation w. This variable is present only if t is assigned to w.
- For each period $p \in \mathcal{P}$, the integer variable $\mathbf{cyc}_p^{\mathbf{req}} \in \mathcal{H}$ indicates the minimum required cycle time value of p (or the maximum end times among tasks time during p).
- For each period $p \in \mathcal{P}$, the integer variable $\mathbf{cyc}_p^{\mathbf{dec}} \in [\![c_{tgt}, c_{max}]\!]$ indicates the decided cycle time value of p (or the cycle time applied to the line during p, which is no less than the required cycle time $\mathbf{cyc}_p^{\mathbf{req}}$). Note that the value of this variable is also the value returned by the assignment function $cyc(p)$ (see Subsection 3.2).
- For each period $p \in \mathcal{P}$, the integer variable $\mathbf{cost}_p \in \mathcal{H}$ indicates the amount of time contributed by p to the ramp-up duration minimisation objective. This variable has the same value as $\mathbf{cyc}_p^{\mathbf{dec}}$ if $\mathbf{cyc}_p^{\mathbf{dec}} > c_{tgt}$, and 0 otherwise.
- For each zone $z \in \mathcal{Z}$, each period $p \in \mathcal{P}$ and each workstation $w \in \mathcal{W}$, the state function $\mathbf{zst}_{z,p,w}$ represents the status of zone z during the periods p at the workstation w.

Constraints and criteria are encoded as follows:

minimise

$$\sum_{p \in \mathcal{P}} \mathbf{cost}_p \tag{23}$$

$$\sum_{p \in \mathcal{P}^S} (\mathbf{cost}_{p-1} \neq \mathbf{cost}_p) \tag{24}$$

such that

$$\forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, \quad \mathsf{presenceOf}(\mathbf{itv}_{t,p,w}) = \mathsf{equal}(\mathbf{wks}_t, w) \tag{25}$$

$$\forall p \in \mathcal{P}, \quad \mathbf{cyc}_p^{\mathbf{req}} = \max_{t \in \mathcal{T}, w \in \mathcal{W}} \mathsf{endOf}(\mathbf{itv}_{t,p,w}, 0) \tag{26}$$

$$\forall p \in \mathcal{P}^U, \quad \mathbf{cyc}_p^{\mathbf{dec}} = \max_{q \in \mathcal{P}^U} \mathbf{cyc}_q^{\mathbf{req}} \tag{27}$$

$$\forall p \in \mathcal{P}^S, \quad \mathbf{cyc}_p^{\mathbf{dec}} \geq \mathbf{cyc}_p^{\mathbf{req}} \tag{28}$$

$$\forall p \in \mathcal{P}^U, \quad \mathbf{cost}_p = \mathbf{cyc}_p^{\mathbf{dec}} \tag{29}$$

$$\forall p \in \mathcal{P}^S, \quad (\mathbf{cyc}_p^{\mathbf{dec}} > c_{tgt}) \implies (\mathbf{cost}_p = \mathbf{cyc}_p^{\mathbf{dec}}) \tag{30}$$

$$\forall p \in \mathcal{P}^S, \quad (\mathbf{cyc}_p^{\mathbf{dec}} = c_{tgt}) \implies (\mathbf{cost}_p = 0) \tag{31}$$

$$\forall (t,u) \in \mathcal{A}, \quad \mathbf{wks}_t \leq \mathbf{wks}_u \tag{32}$$

$$\forall (t,u) \in \mathcal{A}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, \quad \mathsf{endBeforeStart}(\mathbf{itv}_{t,p,w}, \mathbf{itv}_{u,p,w}) \tag{33}$$

$$\forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, \quad \mathsf{lengthOf}(\mathbf{itv}_{t,p,w}, dur_t(p-w)) = dur_t(p-w) \tag{34}$$

$$\forall r \in \mathcal{R}, \quad \sum_{w \in \mathcal{W}} \mathbf{raw}_{r,w} \leq capa_r \tag{35}$$

$$\forall r \in \mathcal{R}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, \quad \sum_{t \in \mathcal{T}} \mathsf{pulse}(\mathbf{itv}_{t,p,w}, cons_{r,t}) \leq \mathbf{raw}_{r,w} \tag{36}$$

$$\forall z \in \mathcal{Z}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, \qquad \sum_{t \in \mathcal{T}} \mathsf{pulse}(\mathbf{itv}_{t,p,w}, cons_{z,t}) \leq capa_z \tag{37}$$

$$\forall z \in \mathcal{Z}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, \forall t \in \mathcal{T} : z \in \mathcal{D}_t, \qquad \mathsf{alwaysEqual}(\mathbf{zst}_{z,p,w}, \mathbf{itv}_{t,p,w}, 0) \tag{38}$$

$$\forall z \in \mathcal{Z}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, \forall t \in \mathcal{T} : cons_{z,t} > 0, \quad \mathsf{alwaysEqual}(\mathbf{zst}_{z,p,w}, \mathbf{itv}_{t,p,w}, 1) \tag{39}$$

The first objective (Equation 23) is to minimize the time required for the assembly line to reach the target cycle time (or the ramp-up duration). The second objective (Equation 24) is to minimize the number of adjustments to the cycle time. The addressed problem involves a trade-off: if we want to reach the target cycle time faster, we need to reduce the cycle time of the line as soon as possible and as much as possible. The constraints (25) link the presence of interval variables with the allocation of tasks. Constraints (26) compute the required cycle time for each period. Constraints (27) ensure that the decided cycle time during the unstable periods remains fixed and is equal to the maximum required cycle time among them. Constraints (28) stipulate that the decided cycle time during the stable periods is greater or equal to the required cycle time. Next, the constraints (29), (30) and (31) calculate, for each period, the time contributed to the ramp-up stage. The precedence relations are handled by the constraints (32) and (33). Constraints (34) calculate the duration for each task in each period. Constraints (35) ensure that the total amount of each resource assigned to all workstations does not exceed its capacity, and constraints (36) enforce cumulative resource limitations at the workstation level. Constraints (37) are cumulative constraints on zone capacity. Finally, the status of the zones during task executions are ensured by constraints (38) and (39).

C A ILP Model for RC-ALBP/L

C.1 ILP Formulation

As explained in the main paper, in this work, we use a discrete-time, time-indexed variable ILP encoding for RC-ALBP/L. Other MILP encodings exist in the context of resource-constrained scheduling, using sequencing or event-based variables [5]. However, in presence of learning effect, task durations become variables, making those encodings non-linear (they require multiplying decision variables by task durations).

We first provide a MILP encoding for a relaxed version of the problem. For each task $t \in \mathcal{T}$, each period $p \in \mathcal{P}$, each workstation $w \in \mathcal{W}$ and each time index $i \in \mathcal{H}$, we define the following decision variables:

$$\mathbf{wks}_{t,w} = \begin{cases} 1 & \text{if task } t \text{ is allocated to workstation } w \\ 0 & \text{otherwise} \end{cases}$$

$$\mathbf{start}_{t,p,w,i} = \begin{cases} 1 & \text{if task } t \text{ starts at time } i \text{ during period } p \text{ in workstation } w \\ 0 & \text{otherwise} \end{cases}$$

$$\mathbf{end}_{t,p,w,i} = \begin{cases} 1 & \text{if task } t \text{ ends at time } i \text{ during period } p \text{ in workstation } w \\ 0 & \text{otherwise} \end{cases}$$

$\mathbf{raw}_{r,w} \in [\![0, capa_r]\!]$ the amount of resource r allocated to workstation w

$\mathbf{cyc}_p^{\mathbf{req}} \in \mathcal{H}$ the minimum required cycle time value of period p

$\mathbf{cyc}_p^{\mathbf{dec}} \in [\![c_{tgt}, c_{max}]\!]$ the decided cycle time value of period p

$\mathbf{cost}_p \quad \in \mathcal{H}$ the amount of time contributed by period p to the ramp-up duration

$$\mathbf{target}_p = \begin{cases} 1 & \text{if the target cycle time is not achieved (or } \mathbf{cyc}_p^{\mathbf{dec}} > c_{tgt}) \\ 0 & \text{otherwise} \end{cases}$$

Constraints are encoded as follows:

$$\forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \quad \sum_{w \in \mathcal{W}} \sum_{i \in \mathcal{H}} \mathbf{start}_{t,p,w,i} = 1 \tag{40}$$

$$\forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \quad \sum_{w \in \mathcal{W}} \sum_{i \in \mathcal{H}} \mathbf{end}_{t,p,w,i} = 1 \tag{41}$$

$$\forall t \in \mathcal{T}, \quad \sum_{w \in \mathcal{W}} \mathbf{wks}_{t,w} = 1 \tag{42}$$

$$\forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, \quad \sum_{i \in \mathcal{H}} \mathbf{start}_{t,p,w,i} = \mathbf{wks}_{t,w} \tag{43}$$

$$\forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, \quad \sum_{i \in \mathcal{H}} \mathbf{end}_{t,p,w,i} = \mathbf{wks}_{t,w} \tag{44}$$

$$\forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, \quad \sum_{i \in \mathcal{H}} i \times \mathbf{end}_{t,p,w,i} \leq \mathbf{cyc}_p^{\mathbf{req}} \tag{45}$$

$$\forall p \in \mathcal{P}, \quad \mathbf{cyc}_p^{\mathbf{dec}} \geq \mathbf{cyc}_p^{\mathbf{req}} \tag{46}$$

$$\forall p \in \mathcal{P}^U \setminus \{1\}, \quad \mathbf{cyc}_{p-1}^{\mathbf{dec}} = \mathbf{cyc}_p^{\mathbf{dec}} \tag{47}$$

$$\forall p \in \mathcal{P}^S, \quad \mathbf{cyc}_p^{\mathbf{dec}} \geq (1 + c_{tgt}) \times \mathbf{target}_p \tag{48}$$

$$\forall p \in \mathcal{P}^S, \quad \mathbf{cyc}_p^{\mathbf{dec}} \leq c_{tgt} + c_{max} \times \mathbf{target}_p \tag{49}$$

$$\forall p \in \mathcal{P}^U, \quad \mathbf{cost}_p = \mathbf{cyc}_p^{\mathbf{dec}} \tag{50}$$

$$\forall p \in \mathcal{P}^S, \quad \mathbf{cost}_p \leq c_{max} \times \mathbf{target}_p \tag{51}$$

$$\forall p \in \mathcal{P}^S, \quad \mathbf{cost}_p \geq \mathbf{cyc}_p^{\mathbf{dec}} - c_{max}(1 - \mathbf{target}_p) \tag{52}$$

$$\forall (t,u) \in \mathcal{A}, \forall w \in \mathcal{W}, \quad \sum_{j=0}^{w} \mathbf{wks}_{t,j} \geq \sum_{j=0}^{w} \mathbf{wks}_{u,j} \tag{53}$$

$$\forall (t,u) \in \mathcal{A}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, \quad \sum_{i \in \mathcal{H}} i(\mathbf{end}_{t,p,w,i} - \mathbf{start}_{u,p,w,i}) \leq c_{max}(2 - \mathbf{wks}_{t,w} - \mathbf{wks}_{u,w}) \tag{54}$$

$$\forall t \in \mathcal{T}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, \quad \sum_{i \in \mathcal{H}} i(\mathbf{end}_{t,p,w,i} - \mathbf{start}_{t,p,w,i}) = dur_t(p - w) \times \mathbf{wks}_{t,w} \tag{55}$$

$$\forall r \in \mathcal{R}, \quad \sum_{w \in \mathcal{W}} \mathbf{raw}_{r,w} \leq capa_r \tag{56}$$

$$\forall r \in \mathcal{R}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, i \in \mathcal{H}, \quad \sum_{t \in \mathcal{T}} cons_{r,t} \left(\sum_{j=0}^{i} (\mathbf{start}_{t,p,w,j} - \mathbf{end}_{t,p,w,j}) \right) \leq \mathbf{raw}_{r,w} \tag{57}$$

$$\forall z \in \mathcal{Z}, \forall p \in \mathcal{P}, \forall w \in \mathcal{W}, i \in \mathcal{H}, \quad \sum_{t \in \mathcal{T}} cons_{z,t} \left(\sum_{j=0}^{i} (\mathbf{start}_{t,p,w,j} - \mathbf{end}_{t,p,w,j}) \right) \leq capa_z \tag{58}$$

The constraints in this ILP formulation are basically the linear version of those in the CP-OPL formulation, except for the constraints relating to the neutralized status of zones. These constraints can be linearised by introducing additional binary variables. However, as detailed later on, the relaxed model without those variables is already challenging.

Constraints (40) and (41) ensure that each task is fully executed once per period. Constraints (42) make sure that each task is allocated to only one workstation. Constraints (43) and (44) link the task allocation variables **wks** with the **start** / **end** variables. The required cycle time and the decided cycle time for each period are constrained by constraints (45) and (46), respectively. Constraints (47) ensure that the decided cycle time during the

unstable periods remains fixed and is equal to the maximum required cycle time among them. Constraints (48) and (49) link the two variables $\mathbf{cyc^{dec}}$ and **target**. Next, the time contributing to the ramp-up stage is constrained by constraints (50) for the unstable periods, and by constraints (51) and (52) for the stable periods. The precedence relations are handled by the constraints (53) and (54). Constraints (55) calculate the duration for each task in each period. Constraints (56) ensure that the total amount of each resource assigned to all workstations does not exceed its capacity, and constraints (57) enforce cumulative resource limitations at the workstation level. Finally, constraints (58) are cumulative constraints on zone capacity.

The criteria remain the same as in the CP-OPL formulation (Equation 23 and 24), which are to minimise the ramp-up duration and the number of adjustments to the line. The second criterion is currently non-linear, but can be made linear by adding extra variables. However, we consider here a relaxed mono-objective version of the problem, which is to minimise only the ramp-up duration. Note that, although the variable $\mathbf{cyc^{dec}}$ can be withdrawn from this mono-objective version, we have decided to keep it since it must be present in the complete problem.

C.2 Experiments Results

The number of variables and constraints of the ILP approach for each instance is presented in Table 5. The `toy` instance (Examples 1 and 2) is solved to optimality by this approach in 1 second, resulting in the same objective value as when using the CP-based approach. However, for all other instances, the ILP approach fails to build the model even after 2 hours and causes the test computer to run out of memory due to an explosion in the number of variables and constraints.

Table 5 Approximate number of variables and constraints for each instance in the ILP model.

Instances	`toy`	`187-2-26`	`199-2-26`	`795-2-26`	`187-4-50`	`199-4-50`	`795-4-50`
# Variables	$2,920$	560×10^6	596×10^6	2.3×10^9	1×10^9	1.1×10^9	4.5×10^9
# Constraints	$1,183$	80×10^6	80×10^6	81×10^6	155×10^6	155×10^6	156×10^6

Parallel MIP Solving with Dynamic Task Decomposition

Peng Lin
Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China

Shaowei Cai[1]
Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China

Mengchuan Zou
Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Shengqi Chen
Department of Computer Science and Technology, Tsinghua University, Beijing, China

Abstract

Mixed Integer Programming (MIP) is a foundational model in operations research. Although significant progress has been made in enhancing sequential MIP solvers through sophisticated techniques and heuristics, remarkable developments in computing resources have made parallel solving a promising direction for performance improvement. In this work, we propose a novel parallel MIP solving framework that employs dynamic task decomposition in a divide-and-conquer paradigm. Our framework incorporates a hardness estimate heuristic to identify challenging solving tasks and a reward decaying mechanism to reinforce the task decomposition decision. We apply our framework to two state-of-the-art open-source MIP solvers, SCIP and HiGHS, yielding efficient parallel solvers. Extensive experiments on the full MIPLIB benchmark, using up to 128 cores, demonstrate that our framework yields substantial performance improvements over modern divide-and-conquer parallel solvers. Moreover, our parallel solvers have established new best known solutions for 16 open MIPLIB instances.

2012 ACM Subject Classification Mathematics of computing → Integer programming; Computing methodologies → Parallel algorithms; Applied computing → Operations research

Keywords and phrases Mixed Integer Programming, Parallel Computing, Complete Search, Task Decomposition

Digital Object Identifier 10.4230/LIPIcs.CP.2025.26

Supplementary Material *Software*: `https://github.com/shaowei-cai-group/PartiMIP`

Funding This work is supported by National Key R&D Program of China (2023YFA1009500).

1 Introduction

Mixed Integer Programming (MIP) is both a foundational model in operations research and a central combinatorial optimization problem [40, 2]. Its importance stems from its ability to model a wide range of problems in diverse fields, from practical applications such as planning [31] and scheduling [13] to theoretical problems in propositional logic [11] and graph theory [25].

[1] Corresponding author

© Peng Lin, Shaowei Cai, Mengchuan Zou, and Shengqi Chen;
licensed under Creative Commons License CC-BY 4.0

MIP is formulated to optimize a linear objective function subject to linear constraints, where decision variables may be either integer or real-valued. Because solving general MIP instances is NP-hard [20, 19], practical applications require sophisticated algorithms. Since solver performance often represents a critical bottleneck when addressing large-scale real-world instances, improving the efficiency of MIP solvers remains a primary focus for algorithm developers.

During the past decades, significant advances have been made in the algorithmic efficiency of MIP solvers, with substantial research efforts primarily focused on the refinement of sequential solving techniques and heuristic enhancements [21, 2, 23]. In today's computing landscape, multi-core architectures are ubiquitous, making parallel computing a natural approach for improving solver performance and a widely researched topic in constraint programming [24, 34, 15].

Numerous studies on parallel MIP solving can be categorized into two groups as summarized in [41]: those developed basically from scratch [12, 5, 8, 9, 32] and those that integrate existing MIP solvers [26, 38, 37, 7, 36, 39]. However, only a few approaches in the latter group achieved competitive results compared to state-of-the-art sequential solvers [39], highlighting the persistent challenges in parallel MIP solving. A further obstacle in this field is that the most effective solvers, both parallel and sequential, are closed-source commercial products, such as Gurobi [17], CPLEX [29], Xpress [3], and COPT [14]. Consequently, it is difficult for the academic community to analyze the underlying principles of their parallel approaches, and when these solvers are used as black-box base solvers, the potential for parallel interaction is inherently limited [33].

Current academic research on parallel MIP solving can be broadly classified into two approaches: portfolio-based and divide-and-conquer methods. Portfolio solvers concurrently execute multiple solvers, or different configurations of a single solver, to address identical or slightly perturbed MIP instances. Representative examples include the race ramp-up technique employed in FiberSCIP [36, 39], the DC cutting plane algorithm with multiple random starts [30], and portfolio methods based on local search [22, 10]. However, the performance of portfolio methods is inherently constrained by the best possible sequential performance. In contrast, the divide-and-conquer approach has the potential to outperform the best sequential methods by accelerating the solving process through the parallelization of key algorithmic components. For example, FiberSCIP [39] parallelizes the processing of nodes from sequential branch-and-bound solvers, while HiGHS offers a parallel version that accelerates the solving of internal linear programming and symmetry detection [18]. Similar work in the area of stochastic MIP includes PIPS-PSBB [28], which combines parallel linear programming solving with parallel branch-and-bound tree search. Furthermore, portfolio-based and divide-and-conquer approaches can be seamlessly integrated to develop highly effective hybrid solvers. Extensive investigations on hybrid solvers have been predominantly led by the UG framework of the SCIP team and are well documented in [36, 39].

Our research focuses on the divide-and-conquer strategy for parallel MIP solving. Most of the existing approaches in this category are tightly coupled with their underlying sequential algorithms. For instance, FiberSCIP relies on sequential solvers to generate nodes to be distributed and solved in parallel, thereby forcing the base solver to employ branch-and-bound methods to maintain the branch-and-bound tree. Consequently, the parallel performance of such systems is heavily influenced by the search strategies (e.g., branching and node selection) employed by the sequential solver.

In this work, we propose a novel parallel MIP solving framework that supports search strategies tailored for parallel environments through dynamic problem decomposition. In our approach, solving a MIP problem is treated as a task organized within a global task

tree of the parallel system, which is used to schedule these tasks for parallel execution by the underlying base solvers. The expansion of the task tree is entirely governed by our newly designed decomposition algorithms, which integrate the dynamic information from the parallel solving process to enhance decomposition decisions, and are inherently independent of the underlying base solver. Consequently, our framework is loosely coupled with the base solvers, which require only standard input/output interfaces and are not restricted to branch-and-bound methods. This design enables seamless integration with a wide range of MIP solvers.

To efficiently expand the task tree, we propose a dynamic task decomposition method by partitioning selected tasks into smaller subtasks. The decision on which task to decompose, as well as the specific decomposition action used, is critical to the overall solving process. To optimize this, we propose a hardness estimate heuristic to assess the difficulty of task solving, thus prioritizing the decomposition of more challenging tasks and allocating additional computational resources accordingly. Furthermore, we develop a reward-guided mechanism that leverages historical feedback from the task tree construction to reinforce future decomposition decisions. By incorporating a reward decaying strategy, this mechanism strikes a balance between exploration and exploitation, ultimately enhancing the performance of the parallel solving process.

We apply our framework to state-of-the-art open-source MIP solvers, including SCIP [6] and HiGHS [18], as our base solvers. We conducted experiments on the entire MIPLIB benchmark across a wide range of core configurations from 8 to 128 cores to evaluate the performance of the resulting parallel solvers. To the best of our knowledge, the 128-core configuration reported in our work represents the largest core configuration used to test parallel solvers on the entire MIPLIB benchmark. Overall, the experimental results demonstrate that our techniques significantly enhance the performance of sequential MIP solvers, yielding a substantial improvement in the discovery of high-quality solutions and the number of solved instances. We compare our framework with the default divide-and-conquer strategies implemented in SCIP (i.e., FiberSCIP) and HiGHS. The comparison shows that our strategy consistently outperforms the corresponding default parallel approaches, and the parallel solver developed using our framework achieves state-of-the-art performance among open-source parallel solvers. Moreover, our framework establishes 16 new records for MIPLIB open instances by discovering new best known solutions.

To facilitate reproducibility and further research, we have made our solvers, evaluation scripts, experimental results, and new best known solutions available on GitHub [2].

2 Preliminaries

2.1 Mixed Integer Programming

▶ **Definition 1** (Mixed Integer Programming). *Let* $\mathbf{A} \in \mathbb{R}^{m \times n}$ *be a matrix,* $\mathbf{b} \in \mathbb{R}^m$ *a vector,* $\mathbf{c}, \mathbf{l}, \mathbf{u} \in \mathbb{R}^n$ *vectors,* $I \subseteq N = \{1, \ldots, n\}$ *denote a set of indices, and* $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ *be the set of decision variables. The mixed integer programming problem is defined as follows:*

[2] https://github.com/shaowei-cai-group/PartiMIP

$$\min_{\mathbf{x}} \quad \mathbf{c}^\top \mathbf{x} \quad s.t. \quad \begin{array}{l} \mathbf{A}\mathbf{x} \leq \mathbf{b}, \\ \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \\ x_j \in \mathbb{Z} \quad \forall j \in I, \\ \mathbf{x} \in \mathbb{R}^n. \end{array} \tag{1}$$

In this formulation, $\mathbf{c}^\top \mathbf{x}$ denotes the objective function, $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ represents the general linear constraints, $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$ specifies the global bounds of decision variables, and the integrality constraints require $x_j \in \mathbb{Z}$ for all $j \in I$.

Domain Propagation. Domain propagation is a technique to reformulate a MIP problem into an equivalent formulation that is potentially easier to solve. By iteratively tightening the variable domains (i.e., $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$) through iterative analysis of constraints and neighboring variable domains, this technique reduces the complexity of the MIP model by eliminating redundancies, such as satisfied constraints or fixed variables. Consequently, domain propagation often effectively excludes considerable portions of the search space, occasionally enabling proofs of optimality or infeasibility. In practice, modern solvers efficiently implement domain propagation in their presolving application programming interfaces (APIs) [6, 17, 29]. The fundamental principles of domain propagation are detailed in [35].

2.2 Parallel Divide-and-conquer Strategies in MIP Solving

Here, we review divide-and-conquer-based parallel strategies employed in state-of-the-art solvers. Among today's open-source solvers, SCIP and HiGHS are prominent, each offering a parallel version. For SCIP, the parallel version, known as FiberSCIP [36, 39], adopts a divide-and-conquer strategy to parallelize the processing of nodes in the branch-and-bound tree. In this method, base solvers alternately solve branch-and-bound nodes and transfer the unsolved child nodes to a load coordinator. The load coordinator maintains a pool of unsolved nodes and assigns them to idle solvers as they become available. In FiberSCIP, the generation of nodes is determined by the internal algorithm of the sequential branch-and-bound solver, while the parallel component is primarily responsible for the collection and distribution of nodes from the sequential branch-and-bound tree. In contrast, the parallel version of HiGHS [18] focuses on the parallelization of specific components within its sequential solver, including the dual simplex algorithm, symmetry detection, and querying of clique tables [3]. Notably, parallel HiGHS has been the top-ranked open-source solver in the MIP rankings for several consecutive years [4].

Both these two parallel solvers are tightly coupled with their underlying sequential frameworks, making it challenging to adapt their methods to solvers lacking these specific components. In contrast, our goal is to develop a general parallel framework that does not rely on the base solver's internal algorithms and requires only a standard input/output interface.

[3] `https://ergo-code.github.io/HiGHS/dev/parallel/`
[4] `https://mattmilten.github.io/mittelmann-plots/`

3 Our Parallel Framework for Solving MIP

In this section, we introduce our parallel MIP solving framework with dynamic task decomposition, named PartiMIP. We first present the basic concepts of the main components and how they cooperate together. Afterwards, we dive into the motivation and design of each component. A detailed description of the dynamic task decomposition method is provided in Sect. 4.

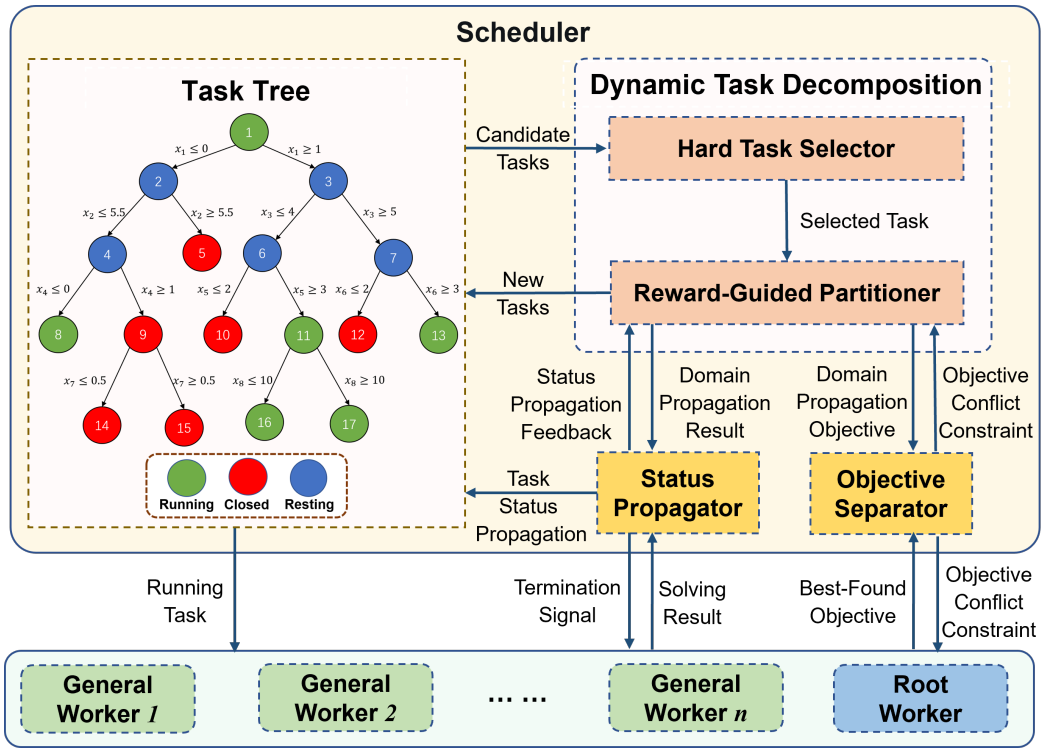


Figure 1 PartiMIP: A parallel MIP solving framework with dynamic task decomposition.

3.1 Basic Concept and Process of Framework

As illustrated in Fig. 1, our framework consists of two roles: the scheduler and the worker. These roles are executed by distinct threads to leverage multiple cores in parallel:

- **Scheduler:** The scheduler maintains a *task tree*, in which the nodes represent the solving tasks for the original problem or subproblems of a given MIP instance. It incorporates a *dynamic task decomposition* to expand the tree, a *status propagator* to deduce task statuses and an *objective separator* to reduce search spaces.
- **Worker:** The workers primarily invoke a base MIP solver to solve the tasks. A *root worker* is designated solely to solve the original problem, while other *general workers* are flexible and can solve any tasks dynamically assigned by the scheduler.

The task tree serves as the central data structure throughout the solving process, where each task can be in one of three possible statuses:

- **Running**: Tasks that are currently being solved by workers.

- **Closed**: Tasks for which the solution result is known (i.e., optimal or infeasible). The solving of closed tasks is completed, and the entire solving process is completed when the root task is closed.
- **Resting**: Tasks that are decomposed into subtasks and remain unassigned to workers. The solution results of resting tasks are inferred from the results of their subtasks.

At a given time, the leaf nodes in the task tree are denoted as *leaf tasks*, each of which contains a distinct search space.

Initially, the scheduler designates the given MIP problem as the root task of the task tree and assigns it to the root worker for solving, enabling quick completion when the instance is simple, and thereby avoiding the overhead of parallel solving.

Subsequently, the scheduler accelerates the solving process in a divide-and-conquer view, decomposing the root task into simpler subtasks. During task decomposition, a leaf task is selected, and the domain of one variable is partitioned into two parts to generate two new subtasks. The search spaces of these subtasks are further reduced through domain propagation, and then the new subtasks are inserted as leaf tasks to update the task tree.

Allowing tasks with overlapping search spaces to be solved simultaneously can lead to the exploration of identical search spaces, which should be avoided as much as possible. In the task tree, each leaf task contains distinct search spaces. To ensure that each general worker explores distinct search spaces at the beginning, task decomposition is divided into two phases:

1. **Initial Decomposition Phase**: Initially, only the root worker is activated to solve the root task, while all general workers remain idle. The scheduler iteratively decomposes the leaf tasks to generate a sufficient number of simpler leaf subtasks to assign to all general workers, during which the statuses of the decomposed tasks transition to resting. The task decomposition process is parallelized to enhance resource utilization and accelerate this phase. Specifically, let $\mathcal{N}$ denote the available CPU cores. $min\{\mathcal{N}/2,\ size(leaf\ tasks)\}$ tasks are selected to be decomposed, and the subtasks are generated and simplified in parallel. Since all selected tasks are leaf tasks, the parallel domain partition and domain propagation processes are independent and consistent. Once sufficient tasks are generated, all general workers are activated to solve the leaf tasks, each with distinct search spaces.
2. **Dynamic Decomposition Phase**: General workers become idle once their assigned tasks are closed. To utilize these idle workers, the scheduler dynamically decomposes ongoing tasks into simpler subtasks and assigns them to idle workers. This strategy ensures that idle workers remain engaged in resolving challenging tasks, thus expediting the whole solving process.

Once a task is solved successfully, the corresponding worker communicates the solution result to the scheduler. Additionally, a worker may receive a termination signal from the scheduler due to task status propagation. In either case, the worker terminates the ongoing task and releases its computational resources to accommodate upcoming tasks.

The simplicity of interaction with the scheduler allows the worker role to be seamlessly integrated into most modern MIP solvers via APIs, without requiring modifications to their internal algorithms. The only requirement for base solvers is the support for a standard input/output interface (e.g., reading problem files and logging solution results), which allows the use of any algorithm, not just those based on branch-and-bound methods. This design offers substantial flexibility in selecting and configuring base solvers, enabling a diverse range of potential solving approaches.

3.2 Task Status Propagation

In the task tree, subtasks are generated by partitioning the domain of a variable in their parent task. Consequently, the union of the search spaces of the subtasks is equivalent to the search space of their parent task. Based on this, the status propagator is designed to exploit the relationship between parent and child tasks to deduce the solution results of related tasks, with the aim of accelerating the solving process. The status propagator monitors signals that signify the solution result of each task, which originate from two sources:

1. During task decomposition, the domain propagation process may directly prove new subtasks to optimality or infeasibility.
2. Workers return the solution results of running tasks.

Once a task is solved, the status propagator tries to deduce the statuses of related tasks through the task tree. The propagation of task status can take place in two directions: upward and downward.

- **Upward Propagation**: This happens when all child tasks are closed. If all child tasks are determined to be infeasible, the parent task is closed as infeasible, and the propagation proceeds upward. If any child task is optimal, the parent task is closed as optimal, and the propagation continues upward.
- **Downward Propagation**: Once the parent task is proven to be optimal or infeasible, all child tasks are closed as optimal or infeasible, and the propagation continues downward.

Furthermore, propagation information reflects the quality of the relationship between parent and subtasks, which is used to improve future task decomposition decisions (Sect. 4.2).

3.3 Objective Conflict Constraint

The differences in search spaces between tasks arise from partitioning variable domains and simplifying through domain propagation. As a result, the original solution for the root task can be reconstructed from any given task solution by reintroducing the eliminated variable assignments. Additionally, the value of the objective function remains unchanged when reconstructing the original root task solution.

Therefore, with respect to the globally updated best-found objective value $\mathcal{O}^*$, finding solutions worse than $\mathcal{O}^*$ for new tasks is meaningless. To eliminate redundant search space that cannot contain a better solution than $\mathcal{O}^*$ for new tasks, we design the objective separator to generate the objective conflict constraint. Specifically, the objective separator dynamically collects the globally updated best-found objective value $\mathcal{O}^*$ in real time. Relying on $\mathcal{O}^*$, for a task $\mathcal{T}$, the objective separator constructs the latest objective conflict constraint:

$$\text{Objective Conflict Constraint: } \mathbf{c}_{\mathcal{T}}\mathbf{x}_{\mathcal{T}} < \mathcal{O}^* - \text{offset}_{\mathcal{T}} \tag{2}$$

where $\mathbf{c}_{\mathcal{T}}\mathbf{x}_{\mathcal{T}}$ represents the product of the variables and their corresponding coefficients in the objective function of $\mathcal{T}$, and $\text{offset}_{\mathcal{T}}$ is the constant term in the objective function of $\mathcal{T}$, which is derived from variable elimination during the domain propagation process.

The objective conflict constraint is in the standard form of a MIP constraint. Incorporating the objective conflict constraint effectively forces the new task to search exclusively for better feasible solutions, significantly reducing the search space. The objective conflict constraint can bring advantages to both the regular solving process and the task decomposition:

- Worker's regular solving: Before solving a newly assigned task, the worker retrieves the latest objective conflict constraint, which is then incorporated into the MIP model to reduce the search space. After task solving begins, the worker synchronizes its best-found objective value in real time with the scheduler, feeding it back to improve the objective conflict constraint.

- Task decomposition: The latest objective conflict constraint is added to the new tasks generated during task decomposition, further enhancing the domain propagation process for greater simplification. Additionally, task decomposition may provide a new best-found objective value once the new task is reduced to optimality via domain propagation.

Providing the best-found objective value as early as possible to the objective separator facilitates the creation of the objective conflict constraint to aid the task decomposition. Therefore, an additional benefit of starting the root worker before the initial decomposition phase is that its regular solving can yield the best-found objective value early in the process.

Additionally, since constructing an objective conflict constraint requires a feasible solution, the application of the objective conflict constraint does not affect the solution result of the original problem, including its feasibility or optimality.

4 Dynamic Task Decomposition

In this section, we propose a dynamic task decomposition method, the key process to maintain the entire task tree. This process decomposes challenging tasks into simpler new tasks for parallel solving. First, we design the *hard task selector*, which selects complex tasks from the candidate tasks to be decomposed. Second, we propose the *reward-guided partitioner* to partition the selected task based on the feedback from historical decomposition actions. These two components both combine static task information with dynamic solving data to enhance decision making. Finally, in Sect. 4.3, we present the complete decomposition algorithm.

4.1 Hard Task Selector

When maintaining the task tree, the elementary operation involves decomposing a complex task into smaller subtasks. The first step in this process is to select a task for decomposition. The strategy used for this selection can influence the structure of the tree by guiding its expansion, such as in a breadth-first search (BFS) or depth-first search (DFS) manner. However, simple strategies like BFS or DFS do not consider the characteristics of the MIP instance in parallel solving, ignoring factors such as load balancing and convergence during the solving process.

Preliminary experiments indicate that distributing relatively easy tasks to workers can incur significant overhead on the scheduler. First, frequent communication between the scheduler and workers is required. Second, continuous decomposition is necessary to generate new tasks for workers that often remain idle. To mitigate this issue, we propose the hard task selector, which prioritizes more challenging tasks for decomposition, aiming to allocate more resources to the complex parts of the task tree.

However, assessing the difficulty of solving a MIP task is NP-hard. To address this, we define a hardness estimate function to heuristically estimate the difficulty of a task. The sparsity of a MIP matrix ($\mathbf{A}$ in $\mathbf{Ax} \leq \mathbf{b}$) is correlated with the difficulty of solving the problem [4]. When the constraint matrix is dense, the solving of large models often becomes computationally intractable, as noted in the HiGHS documentation [5]. Therefore, during the initial decomposition phase, the difficulty of a task is assessed by its non-zero count, defined as follows:

▶ **Definition 2** (nnz). *The **nnz** of a task is the number of non-zero elements in the constraint matrix of the task.*

[5] https://ergo-code.github.io/HiGHS/dev/terminology/#The-constraint-matrix

As the process transitions into the dynamic decomposition phase, tasks are selected among those currently running, and additional runtime information becomes available. At any given moment, tasks that have been running longer are more likely to encounter computational difficulties due to increased elapsed time. Therefore, we define the duration of a task as follows:

▶ **Definition 3** (duration). *Given a specified time point, the* ***duration*** *of a task is the elapsed runtime since the task was assigned to a worker for execution.*

Consequently, the hard task selector incorporates both the non-zero count and the duration. Specifically, for a task $\mathcal{T}$, the hardness estimate function $\textbf{Hardness}(\mathcal{T})$ is defined as:

$$\text{Hardness}(\mathcal{T}) = \begin{cases} nnz \text{ of } \mathcal{T}, & \text{initial decomposition phase,} \\ nnz \text{ of } \mathcal{T} \times duration \text{ of } \mathcal{T}, & \text{dynamic decomposition phase.} \end{cases} \tag{3}$$

4.2 Reward-Guided Partitioner

Given a task selected by the hard task selector, we design a reward-guided partitioner to choose a specific variable for decomposition. The domain of the selected variable is partitioned and subsequently the domain propagation is applied to simplify the resulting subtasks. The partitioner incorporates a reward-guided mechanism for variable selection and employs a reward decaying strategy to balance exploration and exploitation.

Rewarding Variables. The goal of task decomposition is that the resulting subtasks are more easily solvable than the parent task. If a parent task is closed due to upward propagation triggered by the closure of all its child tasks, this indicates that the selected variable for decomposing the parent task is a favorable choice. This is because the time required to close all subtasks is shorter than the time required to solve the parent task directly. To encourage this desirable outcome, we design a variable reward mechanism to reinforce future task decomposition decisions.

Initially, each variable is associated with a global reward value, denoted $\textbf{Reward}(x)$, which is initialized to 0. Once the status propagator receives a solution result of a task, it performs task status propagation to update the task tree. Subsequently, the results of this task status propagation are then used to reward the corresponding variables. For simplicity, we denote the variable selected in the decomposition of a parent task $\mathcal{T}$ as $x_{\mathcal{T}}$. Specifically, for each closed task in a task status propagation, the rewards of corresponding selected variables are updated as follows:

$$\text{Reward}(x_{\mathcal{T}}) := \begin{cases} \text{Reward}(x_{\mathcal{T}}) + 1, & \text{if } \mathcal{T} \text{ is closed by upward propagation} \\ \text{Reward}(x_{\mathcal{T}}), & \text{otherwise} \end{cases} \tag{4}$$

These updates ensure that a higher reward for a variable indicates that it has been selected more frequently in parent tasks that are closed via upward propagation.

Reward-Guided Multi-level Selection. For task decomposition, we prioritize selecting the variable with the highest reward, which is dynamically updated as the solving process progresses. However, when multiple variables share the same highest reward, particularly before any upward propagation occurs, tiebreaking is necessary. In such cases, static information about variables is incorporated to guide the task decomposition process.

First, we consider the number of constraints in which each variable appears. The more constraints a variable appears in, the greater the potential to simplify additional constraints through domain propagation resulting from the partitioning of the variable's domain.

Additionally, based on our preliminary experiments, we observe that partitioning variables with very small domains (e.g., less than 0.1) or excessively large domains (e.g., containing infinity) is often ineffective. Therefore, variables with reasonable domain lengths are prioritized.

Algorithm 1 Reward-Guided Multi-level Selection Heuristic.

Input: $\mathcal{T}$: A task to be decomposed
Output: $x_{\mathcal{T}}$: A variable to decompose the task $\mathcal{T}$

```
candidate_variables ← ∅;
normal_variables ← {variables with normal domain lengths | all variables in T};
if normal_variables ≠ ∅ then
    candidate_variables ← normal_variables;
else candidate_variables ← {all variables in T};
best_variables ← arg max_{v∈candidate_variables} {Reward(v)};
x_T ← arg max_{v∈best_variables} {the number of constraints where v appears in T};
return x_T;
```

Based on these considerations, we design a reward-guided multi-level heuristic to select the most promising variable for decomposing a task, as detailed in Algorithm 1.

Reward Decaying. In the reward-guided multi-level selection algorithm, reward is the primary criterion for variable selection. Since subtasks only contain a subset of the original problem's variables, the higher a variable's reward, the greater its chance of becoming the variable with the highest local reward for a specific task. Therefore, as the dynamic decomposition phase progresses, the likelihood that a variable is selected increases with its reward. Being selected, in turn, is a prerequisite for the increase of reward, creating a positive feedback loop within the algorithm. However, such a mechanism can lead to premature convergence in the selection process. To enhance selection diversity and strike a balance between exploration and exploitation, we propose a reward decaying strategy. When a variable is selected, its reward decays, thereby increasing the likelihood that other variables will be selected in subsequent iterations. In this sense, the reward of a variable becomes a global quota, and each time it is selected, a portion of its reward is consumed.

Specifically, for a task $\mathcal{T}$ to be decomposed, the reward of the selected variable $x_{\mathcal{T}}$ is decayed as follows:

$$\text{Reward}(x_{\mathcal{T}}) := \begin{cases} \text{Reward}(x_{\mathcal{T}}) - 1, & \text{if Reward}(x_{\mathcal{T}}) > 0 \\ \text{Reward}(x_{\mathcal{T}}), & \text{otherwise} \end{cases} \tag{5}$$

4.3 Task Decomposition Algorithm

Here, based on the proposed ideas, we present the dynamic task decomposition in Algorithm 2.

Algorithm 2 Task Decomposition Algorithm.

1 $candidate_tasks \leftarrow$ {the unclosed leaf tasks | leaf tasks from the task tree};
2 $\mathcal{T} \leftarrow \arg\max_{\tau \in candidate_tasks}$ {hardness estimate function Hardness(τ) by Eq. 3};
3 Update the latest objective conflict constraint of $\mathcal{T}$ to reduce search space by Eq. 2;
4 $\mathcal{V} \leftarrow$ a decomposition variable to be partitioned in $\mathcal{T}$ by Algorithm 1;
5 Perform reward decaying on $\mathcal{V}$ by Eq. 5;
6 $\{lb, ub\} \leftarrow$ the lower bound and upper bound of $\mathcal{V}$ within $\mathcal{T}$;
7 $\{\mathcal{T}_{\text{left}}, \mathcal{T}_{\text{right}}\} \leftarrow$ partition the domain of $\mathcal{V}$ on $\mathcal{T}$ into $[lb, \frac{lb+ub}{2}]$ and $[\frac{lb+ub}{2}, ub]$;
8 $\{\hat{\mathcal{T}}_{\text{left}}, \hat{\mathcal{T}}_{\text{right}}\} \leftarrow$ perform domain propagation on $\{\mathcal{T}_{\text{left}}, \mathcal{T}_{\text{right}}\}$ to simplify tasks;
9 Add $\{\hat{\mathcal{T}}_{\text{left}}, \hat{\mathcal{T}}_{\text{right}}\}$ into the task tree as new leaf tasks;

First, the task and the variable for task decomposition are selected using the hardness estimate function and the reward-guided multi-level selection heuristic, respectively (lines 1-4). Next, the reward decaying is applied to the selected variable (line 5). Two subtasks are generated by partitioning the domain of the selected variable at the midpoint (lines 6-7), followed by domain propagation to simplify the new tasks (line 8). Finally, the newly generated tasks are added to the task tree (line 9). Note that the latest objective conflict constraint is integrated into the task decomposition to facilitate domain propagation, thereby simplifying additional search spaces (line 3).

5 Experimental Evaluations

This section systematically evaluates the PartiMIP framework through four complementary dimensions: (1) advantages over state-of-the-art parallel strategies, (2) breakthroughs in challenging open instances, (3) performance gains over sequential solvers, and (4) ablation studies on the effectiveness of the reward-guided multi-level selection heuristic. Collectively, the experimental results demonstrate our framework's ability to leverage parallel computation to improve both solution quality and solving efficiency for general MIP instances.

5.1 Experiment Preliminaries

This subsection introduces the setup of the experiment, including the implementation, benchmarks, base solvers, running environment, and evaluation metrics.

Implementation. Both the scheduler and the workers in our framework are implemented in C++ and compiled using g++ 9.4.0 with the -O3 optimization flag. Parallelization is achieved using the pthread libraries. During task decomposition, domain propagation is executed by invoking the presolving APIs provided by the HiGHS libraries [18] [6].

Benchmarks. Our experiments are conducted on the entire MIPLIB benchmark, MIPLIB2017 [16], which comprises 240 general MIP instances collected from a wide range of real-world applications [7], which also serves as the criterion for Hans Mittelmann's benchmark [8], which is a widely recognized worldwide MIP solver ranking [27].

[6] https://ergo-code.github.io/HiGHS/dev/guide/further/#guide-presolve
[7] https://miplib.zib.de/downloads/benchmark.zip
[8] https://mattmilten.github.io/mittelmann-plots/

Base Solvers. For our research, we selected two state-of-the-art open-source MIP solvers as base solvers in the workers: SCIP (v9.2.0) [1, 6] and HiGHS (v1.9.0) [18]. SCIP is one of the most widely used MIP solvers in both academia and industry; it has been in continuous development for over 20 years and plays a crucial role in solving various optimization problems. Similarly, HiGHS has consistently demonstrated superior performance among open-source solvers in MIP solver rankings over several consecutive years. For comparison, we also evaluated the parallel versions of these solvers.

Experiment Setup. All experiments were carried out on a server equipped with 2 AMD EPYC 9654 CPUs and 2048 GB of RAM, running Ubuntu 20.04.4. We report the following metrics for each solver on the MIPLIB benchmark:

- **SOLVED**: The total number of instances that the solver successfully solved, either by proving optimality or infeasibility.
- **SOLVED Improvement (S-Imp.)**: The percentage increase in the total number of solved instances relative to sequential solvers or comparative parallel solvers, reflecting the overall enhancement in problem resolution capability.
- **PAR-2**: The penalized runtime score commonly used in SAT competitions [9]. The runtime for an unsolved instance is penalized by twice the time limit, yielding a single measure that reflects both the runtime efficiency and the number of instances solved.
- **PAR-2 Improvement (P-Imp.)**: The percentage reduction in the PAR-2 score relative to sequential solvers or comparative parallel solvers, indicating the overall efficiency gain.
- **FEAS**: The number of instances for which the solver finds at least one feasible solution, thereby assessing its ability to identify feasibility.
- **FEAS Improvement (F-Imp.)**: The percentage increase in the number of instances with at least one feasible solution compared to sequential solving or comparative parallel solvers, reflecting improvements in feasibility detection.
- **WIN**: The number of instances in which the solver achieved the best feasible solution among compared solvers, measuring the effectiveness in obtaining high-quality solutions.
- **WIN Improvement (W-Imp.)**: The percentage increase in the number of instances in which the solver found the best feasible solution relative to sequential solving or comparative parallel solvers, highlighting improvements in solution quality.

We compare our framework with sequential solving and with parallel divide-and-conquer strategies employed in state-of-the-art MIP solvers. The parallel solvers are evaluated on core configurations of 8, 16, 32, 64, and 128 cores; to our knowledge, the 128-core configuration reported herein is the largest used to test parallel solvers on the full MIPLIB benchmark. In our experiments, each instance is solved to a relative gap of 0, consistent with the setting in the Hans Mittelmann's benchmark. Each solver is executed with a 300-second time limit per instance, and the total CPU time consumed by the competitive experiments exceeded 2.3 CPU years. To accelerate experimentation, we run one or more instances concurrently to fully utilize 128 cores; for example, we may run 16 instances on 8 threads each, 4 instances on 32 threads each, or a single instance on 128 threads.

Finally, our solver, evaluation scripts, related experimental results, and new best known solutions have been made available on GitHub [10]. Researchers interested in our work are encouraged to access our solver and explore the experimental details further.

[9] `https://satcompetition.github.io/`
[10] `https://github.com/shaowei-cai-group/PartiMIP`

5.2 Comparison to Parallel Divide-and-conquer Strategies

Table 1 Comparison to the parallel divide-and-conquer strategies in state-of-the-art open-source solvers, where "FiberSCIP_X" refers to the parallel version of SCIP implemented in FiberSCIP with X cores. "Parallel-HiGHS_X" stands for the parallel version of HiGHS with X cores. "PartiMIP-[base solver]_X" is the notation for employing base solver within our framework with X cores.

Solver	WIN	W-Imp.	FEAS	F-Imp.	SOLVED	S-Imp.	PAR-2	P-Imp.
FiberSCIP_8	129	0.0%	198	0.0%	79	0.0%	102421.1	0.0%
PartiMIP-SCIP_8	**159**	**23.3%**	**208**	**5.1%**	**81**	**2.5%**	**100615.9**	**1.8%**
FiberSCIP_16	126	0.0%	200	0.0%	83	0.0%	100803.4	0.0%
PartiMIP-SCIP_16	**163**	**29.4%**	**210**	**5.0%**	**86**	**3.6%**	**97747.0**	**3.0%**
FiberSCIP_32	125	0.0%	202	0.0%	87	0.0%	98630.5	0.0%
PartiMIP-SCIP_32	**168**	**34.4%**	**214**	**5.9%**	**88**	**1.1%**	**96887.0**	**1.8%**
FiberSCIP_64	128	0.0%	202	0.0%	93	0.0%	95876.1	0.0%
PartiMIP-SCIP_64	**167**	**30.5%**	**212**	**5.0%**	**94**	**1.1%**	**94113.6**	**1.8%**
FiberSCIP_128	120	0.0%	201	0.0%	92	0.0%	96415.2	0.0%
PartiMIP-SCIP_128	**168**	**40.0%**	**214**	**6.5%**	**98**	**6.5%**	**92223.4**	**4.3%**
Parallel-HiGHS_8	110	0.0%	192	0.0%	79	0.0%	101955.1	0.0%
PartiMIP-HiGHS_8	**179**	**62.7%**	**200**	**4.2%**	**89**	**12.7%**	**96903.0**	**5.0%**
Parallel-HiGHS_16	107	0.0%	192	0.0%	79	0.0%	101945.6	0.0%
PartiMIP-HiGHS_16	**184**	**72.0%**	**206**	**7.3%**	**89**	**12.7%**	**96480.1**	**5.4%**
Parallel-HiGHS_32	111	0.0%	192	0.0%	79	0.0%	101956.3	0.0%
PartiMIP-HiGHS_32	**186**	**67.6%**	**209**	**8.9%**	**96**	**21.5%**	**93368.3**	**8.4%**
Parallel-HiGHS_64	101	0.0%	192	0.0%	78	0.0%	102273.5	0.0%
PartiMIP-HiGHS_64	**190**	**88.1%**	**209**	**8.9%**	**97**	**24.4%**	**92603.9**	**9.5%**
Parallel-HiGHS_128	101	0.0%	192	0.0%	78	0.0%	102322.3	0.0%
PartiMIP-HiGHS_128	**190**	**88.1%**	**209**	**8.9%**	**100**	**28.2%**	**90516.2**	**11.5%**

We compare our PartiMIP framework with the default parallel divide-and-conquer strategies in state-of-the-art open-source solvers.

- **FiberSCIP** [39]: A famous parallel version of SCIP employing a normal ramp-up phase process, developed by the SCIP team. We use its latest version 1.0.0, which is based on SCIP 9.2.0 as its internal base solver.
- **Parallel HiGHS** (v1.9.0) [18]: The parallel version of HiGHS, featuring parallel dual simplex, symmetry detection, and clique table querying [11]. According to Hans-Mittermann's benchmark, the 8-core HiGHS is the top-ranked open-source solver in the MIP rankings.

Table 1 reports the performance metrics on the MIPLIB benchmark for various configurations. Our experimental results clearly demonstrate the advantages of the PartiMIP framework over conventional parallel strategies. For example, with an 8-core configuration, PartiMIP-SCIP achieves a 23.3% improvement in the WIN metric, a 5.1% increase in the FEAS metric, a 2.5% increase in the SOLVED metric, and a 1.8% reduction in the PAR-2 score compared to FiberSCIP. These performance gains not only persist but also improve with higher core counts; At 128 cores, PartiMIP-SCIP shows a 40.0% improvement in WIN and a 6.5% enhancement in SOLVED relative to FiberSCIP.

Similarly, PartiMIP-HiGHS consistently outperforms the parallel version of HiGHS across all configurations. With 8 cores, PartiMIP-HiGHS records a 62.7% improvement in WIN and a 12.7% in SOLVED. At 128 cores, it attains a 88.1% improvement in WIN and an 28.2% improvement in SOLVED compared to its default counterpart.

[11] `https://ergo-code.github.io/HiGHS/dev/parallel/`

Notably, the superior performance of PartiMIP is not confined to a single solver or configuration. Our framework consistently enhances all metrics, including the total number of solved instances (SOLVED), the quality of feasible solutions (WIN and FEAS), and overall runtime efficiency (PAR-2), across both SCIP and HiGHS. These findings robustly validate our dynamic task decomposition and scheduling strategies, demonstrating that PartiMIP not only improves the efficiency of parallel MIP solving but also facilitates a robust, scalable integration with a wide range of MIP solvers.

5.3 New Best Known Solutions to Open Instances

Table 2 PartiMIP establishes new best known solutions for 16 open instances.

Instance name	#Variable	#Constraint	Previous Best	PartiMIP
dlr1	9142907	1735470	2708148.95990256	2708064.1369803
neos-5151569-mologa	108116	45671	686759699	686750731.344582
bmocbd3	403771	152791	-372986719.737107	-373286017.205902
gmut-76-40	24338	2586	-14169441.78	-14169460.9675000
eva1aprime6x6opt	3514	34872	-16.31528287738903	-18.100995280293
dws012-02	51108	26382	122074.2013795086	121112.055928511
neos-4232544-orira	87060	180600	5557371.400000357	5553207.1245239
neos-4292145-piako	32950	75834	29160.50026450142	28122.4999807616
polygonpack5-15	48163	163429	-55494653.8357854	-55494686.5559904
sct5	37265	13304	-228.1172303718	-228.119492755556
cmflsp40-36-2-10	28152	4266	66452235.08297937	66452234.49456009
adult-regularized	32674	32709	7022.953543477999	7022.953543474559
supportcase23	24275	40502	-12160.6593559088	-12160.6593571676
neos-5045105-creuse	3848	252	20.57142909929996	20.5714105876044
gsvm2rl9	801	600	7438.181167768	7438.181021170049
s82	1690631	87878	-33.78523764658873	-33.7970576238223

In the MIPLIB dataset, open instances are those for which the optimal solution remains unproven. The current best known solutions for these instances are published on the MIPLIB website, and these open instances pose significant research challenges that drive progress in the field of MIP solving. Notably, PartiMIP has established new best known solutions for 16 open instances, and these solutions have been submitted and accepted by MIPLIB.

As shown in Table 2, these 16 instances vary widely in the number of variables and constraints, reflecting a wide range of problem structures. This diversity demonstrates the extensive applicability and robust solveability of our framework.

Overall, these experimental results validate the potential of large-scale parallel solving to overcome difficult MIP instances. By dynamically decomposing tasks and using parallel computation, PartiMIP achieves significant gains in both solution quality and efficiency, providing compelling evidence of its superior performance and extensive applicability on challenging optimization problems.

5.4 Comparison to Sequential Solving

Here, we evaluate the effectiveness of our dynamic task decomposition framework in enhancing and accelerating the solving capabilities of sequential solvers on the MIPLIB benchmark. Table 3 compares sequential solvers with their PartiMIP-enhanced counterparts across various core configurations, clearly demonstrating that our parallel framework consistently improves performance for both SCIP and HiGHS.

Table 3 Comparison to sequential solving in MIPLIB benchmarks. "[base solver]_Sequential" denotes sequential solving of the corresponding base solver, and solvers using our framework are labeled "PartiMIP-[base solver]" with the number of cores indicated.

Solver	WIN	W-Imp.	FEAS	F-Imp.	SOLVED	S-Imp.	PAR-2	P-Imp.
SCIP_Sequential	85	0.0%	198	0.0%	73	0.0%	105616.9	0.0%
PartiMIP-SCIP_8	110	29.4%	208	5.1%	81	11.0%	100615.9	4.7%
PartiMIP-SCIP_16	128	50.6%	210	6.1%	86	17.8%	97747.0	7.5%
PartiMIP-SCIP_32	136	60.0%	214	8.1%	88	20.5%	96887.0	8.3%
PartiMIP-SCIP_64	142	67.1%	212	7.1%	94	28.8%	94113.6	10.9%
PartiMIP-SCIP_128	149	75.3%	214	8.1%	98	34.2%	92223.4	12.7%
HiGHS_Sequential	91	0.0%	191	0.0%	76	0.0%	103461.3	0.0%
PartiMIP-HiGHS_8	108	18.7%	200	4.7%	89	17.1%	96903.0	6.3%
PartiMIP-HiGHS_16	118	29.7%	206	7.9%	89	17.1%	96480.2	6.7%
PartiMIP-HiGHS_32	120	31.9%	209	9.4%	96	26.3%	93368.3	9.8%
PartiMIP-HiGHS_64	138	51.6%	209	9.4%	97	27.6%	92603.9	10.5%
PartiMIP-HiGHS_128	148	62.6%	209	9.4%	100	31.6%	90516.2	12.5%

PartiMIP markedly improves solution quality, as evidenced by substantial improvements in the WIN metric for both solvers[12]. It also improves feasibility detection, demonstrating that our parallelized decomposition strategy facilitates the rapid identification of feasible solutions. Moreover, compared to their sequential counterparts, both PartiMIP-SCIP and PartiMIP-HiGHS achieve a notable increase in the total number of solved instances, accompanied by a consistent reduction in the PAR-2 score, indicating that PartiMIP accelerates the overall solving process.

Across all configurations, from 8 to 128 cores, PartiMIP consistently enhances all performance metrics, demonstrating its robust scalability in leveraging parallel computing resources. The consistent improvements observed across different base solvers and core configurations further validate the robustness and broad applicability of our approach.

5.5 Ablation Study

To evaluate the effectiveness of our proposed reward-guided multi-level selection heuristic (Algorithm 1), we conducted comparative experiments with a modified variant. Specifically, we modify PartiMIP by implementing random variable selection for partition tasks, yielding a modified version designated PartiMIP-R.

As shown in Table 4, PartiMIP consistently outperforms PartiMIP-R in almost all evaluation metrics. These results confirm that the reward-guided selection is effective and essential for improving variable selection during task decomposition.

6 Conclusions

In this work, we propose a novel parallel MIP solving framework with dynamic task decomposition. Our approach is built on two key ideas: a hardness estimate heuristic that identifies challenging subproblems and a reward decaying mechanism that reinforces decomposition decisions. Using our framework, we developed parallel solvers based on two leading open-source

[12] The WIN values here are computed by comparing solvers with the same base solver; thus WIN values are lower than those in Table 1, where only solvers with identical core counts are compared.

Table 4 Comparison between PartiMIP and its modified version PartiMIP-R.

Solver	WIN	W-Imp.	FEAS	F-Imp.	SOLVED	S-Imp.	PAR-2	P-Imp.
PartiMIP-R-SCIP_8	158	0.0%	203	0.0%	78	0.0%	102534.5	0.0%
PartiMIP-SCIP_8	**170**	**7.6%**	**208**	**2.5%**	**81**	**3.8%**	**100615.9**	**1.9%**
PartiMIP-R-SCIP_16	154	0.0%	208	0.0%	82	0.0%	101123.1	0.0%
PartiMIP-SCIP_16	**178**	**15.6%**	**210**	**1.0%**	**86**	**4.9%**	**97747.0**	**3.3%**
PartiMIP-R-SCIP_32	169	0.0%	212	0.0%	79	0.0%	101974.5	0.0%
PartiMIP-SCIP_32	**176**	**4.1%**	**214**	**0.9%**	**88**	**11.4%**	**96887.0**	**5.0%**
PartiMIP-R-SCIP_64	166	0.0%	**213**	0.0%	81	0.0%	101101.9	0.0%
PartiMIP-SCIP_64	**181**	**9.0%**	212	-0.5%	**94**	**16.0%**	**94113.6**	**6.9%**
PartiMIP-R-SCIP_128	162	0.0%	**215**	0.0%	86	0.0%	98563.1	0.0%
PartiMIP-SCIP_128	**181**	**11.7%**	214	-0.5%	**98**	**14.0%**	**92223.4**	**6.4%**
PartiMIP-R-HiGHS_8	154	0.0%	199	0.0%	86	0.0%	98939.8	0.0%
PartiMIP-HiGHS_8	**164**	**6.5%**	**200**	**0.5%**	**89**	**3.5%**	**96903.0**	**2.1%**
PartiMIP-R-HiGHS_16	148	0.0%	204	0.0%	83	0.0%	99885.4	0.0%
PartiMIP-HiGHS_16	**180**	**21.6%**	**206**	**1.0%**	**89**	**7.2%**	**96480.1**	**3.4%**
PartiMIP-R-HiGHS_32	162	0.0%	205	0.0%	86	0.0%	98660.3	0.0%
PartiMIP-HiGHS_32	**177**	**9.3%**	**209**	**2.0%**	**96**	**11.6%**	**93368.2**	**5.4%**
PartiMIP-R-HiGHS_64	155	0.0%	208	0.0%	87	0.0%	98003.5	0.0%
PartiMIP-HiGHS_64	**178**	**14.8%**	**209**	**0.5%**	**97**	**11.5%**	**92603.9**	**5.5%**
PartiMIP-R-HiGHS_128	151	0.0%	206	0.0%	89	0.0%	97166.4	0.0%
PartiMIP-HiGHS_128	**174**	**15.2%**	**209**	**1.5%**	**100**	**12.4%**	**90516.2**	**6.8%**

MIP solvers. Extensive experiments demonstrate that our parallel strategy outperforms the divide-and-conquer approaches employed by existing state-of-the-art MIP solvers. Moreover, our parallel solvers have established new best known solutions for 16 open MIPLIB instances.

For future work, our goal is to generalize our framework to address a broader range of constraint programming problems.

References

1 Tobias Achterberg. SCIP: solving constraint integer programs. *Math. Program. Comput.*, 1(1):1–41, 2009. doi:10.1007/S12532-008-0001-1.

2 Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In *Facets of combinatorial optimization: Festschrift for martin grötschel*, pages 449–481. Springer, 2013.

3 Timo Berthold, James Farmer, Stefan Heinz, and Michael Perregaard. Parallelization of the FICO xpress-optimizer. *Optim. Methods Softw.*, 33(3):518–529, 2018. doi:10.1080/10556788.2017.1333612.

4 Robert E. Bixby. Solving real-world linear programs: A decade and more of progress. *Oper. Res.*, 50(1):3–15, 2002. doi:10.1287/OPRE.50.1.3.17780.

5 Robert E. Bixby, William J. Cook, Alan Cox, and Eva K. Lee. Computational experience with parallel mixed integerprogramming in a distributed environment. *Ann. Oper. Res.*, 90:19–43, 1999. doi:10.1023/A\%3A1018960631213.

6 Suresh Bolusani, Mathieu Besanc$_c$on, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, et al. The scip optimization suite 9.0. *arXiv preprint arXiv:2402.17702*, 2024.

7 Michael R. Bussieck, Michael C. Ferris, and Alexander Meeraus. Grid-enabled optimization with GAMS. *INFORMS J. Comput.*, 21(3):349–362, 2009. doi:10.1287/IJOC.1090.0340.

8 Qun Chen and Michael C. Ferris. FATCOP: A fault tolerant condor-pvm mixed integer programming solver. *SIAM J. Optim.*, 11(4):1019–1036, 2001. doi:10.1137/S1052623499353911.

9 Qun Chen, Michael C. Ferris, and Jeff T. Linderoth. FATCOP 2.0: Advanced features in an opportunistic mixed integer programming solver. *Ann. Oper. Res.*, 103(1-4):17–32, 2001. doi:10.1023/A:1012982400848.

10 Zhihan Chen, Peng Lin, Hao Hu, and Shaowei Cai. Parls-pbo: A parallel local search solver for pseudo boolean optimization. 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 5:1–5:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CP.2024.5.

11 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.

12 Jonathan Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. *SIAM J. Optim.*, 4(4):794–814, 1994. doi:10.1137/0804046.

13 Christodoulos A. Floudas and Xiaoxia Lin. Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Ann. Oper. Res.*, 139(1):131–162, 2005. doi:10.1007/S10479-005-3446-X.

14 Dongdong Ge, Qi Huangfu, Zizhuo Wang, Jian Wu, and Yinyu Ye. Cardinal optimizer (COPT) user guide. *CoRR*, abs/2208.14314, 2022. doi:10.48550/arXiv.2208.14314.

15 Ian P. Gent, Ian Miguel, Peter Nightingale, Ciaran McCreesh, Patrick Prosser, Neil C. A. Moore, and Chris Unsworth. A review of literature on parallel constraint solving. *Theory Pract. Log. Program.*, 18(5-6):725–758, 2018. doi:10.1017/S1471068418000340.

16 Ambros M. Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff T. Linderoth, Marco E. Lübbecke, Hans D. Mittelmann, Derya B. Özyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. *Math. Program. Comput.*, 13(3):443–490, 2021. doi:10.1007/S12532-020-00194-3.

17 LLC Gurobi Optimization. Gurobi optimizer ref. manual, 2024.

18 Qi Huangfu and J. A. J. Hall. Parallelizing the dual revised simplex method. *Math. Program. Comput.*, 10(1):119–142, 2018. doi:10.1007/S12532-017-0130-5.

19 Ravindran Kannan and Clyde L Monma. On the computational complexity of integer programming problems. In *Optimization and Operations Research: Proceedings of a Workshop Held at the University of Bonn, October 2–8, 1977*, pages 161–172. Springer, 1978.

20 Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.

21 Ailsa H. Land and Alison G. Doig. An automatic method for solving discrete programming problems. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 – From the Early Years to the State-of-the-Art*, pages 105–132. Springer, 2010. doi:10.1007/978-3-540-68279-0_5.

22 Peng Lin, Mengchuan Zou, Zhihan Chen, and Shaowei Cai. Parailp: A parallel local search framework for integer linear programming with cooperative evolution mechanism. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*, pages 6949–6957. ijcai.org, 2024. URL: https://www.ijcai.org/proceedings/2024/768.

23 Andrea Lodi. Mixed integer programming computation. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 – From the Early Years to the State-of-the-Art*, pages 619–645. Springer, 2010. doi:10.1007/978-3-540-68279-0_16.

24 Arnaud Malapert, Jean-Charles Régin, and Mohamed Rezgui. Embarrassingly parallel search in constraint programming. *J. Artif. Intell. Res.*, 57:421–464, 2016. doi:10.1613/JAIR.5247.

25 Rafael A. Melo and Celso C. Ribeiro. MIP formulations for induced graph optimization problems: a tutorial. *Int. Trans. Oper. Res.*, 30(6):3159–3200, 2023. doi:10.1111/ITOR.13299.

26 Gautam Mitra, I. Hai, and M. T. Hajian. A distributed processing algorithm for solving integer programs using a cluster of workstations. *Parallel Comput.*, 23(6):733–753, 1997. doi:10.1016/S0167-8191(97)00016-1.

27 Hans Mittelmann. Latest benchmark results. In *Proceedings of the INFORMS Annual Conference, Phoenix, AZ, USA*, pages 4–7, 2018.

28 Lluís-Miquel Munguía, Geoffrey Oxberry, Deepak Rajan, and Yuji Shinano. Parallel PIPS-SBB: multi-level parallelism for stochastic mixed-integer programs. *Comput. Optim. Appl.*, 73(2):575–601, 2019. doi:10.1007/S10589-019-00074-0.

29 Stefan Nickel, Claudius Steinhardt, Hans Schlenker, and Wolfgang Burkart. Ibm ilog cplex optimization studio—a primer. In *Decision Optimization with IBM ILOG CPLEX Optimization Studio: A Hands-On Introduction to Modeling with the Optimization Programming Language (OPL)*, pages 9–21. Springer, 2022.

30 Yi-Shuai Niu, Yu You, and Wen-Zhuo Liu. Parallel DC cutting plane algorithms for mixed binary linear program. In Hoai An Le Thi, Hoai Minh Le, and Tao Pham Dinh, editors, *Optimization of Complex Systems: Theory, Models, Algorithms and Applications, WCGO 2019, World Congress on Global Optimization, Metz, France, 8-10 July, 2019*, volume 991 of *Advances in Intelligent Systems and Computing*, pages 330–340. Springer, 2019. doi:10.1007/978-3-030-21803-4_34.

31 Yves Pochet and Laurence A Wolsey. *Production planning by mixed integer programming*, volume 149. Springer, 2006.

32 Ted K. Ralphs, Laszlo Ladányi, and Matthew J. Saltzman. Parallel branch, cut, and price for large-scale discrete optimization. *Math. Program.*, 98(1-3):253–280, 2003. doi:10.1007/S10107-003-0404-8.

33 Ted K. Ralphs, Yuji Shinano, Timo Berthold, and Thorsten Koch. Parallel solvers for mixed integer linear optimization. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 283–336. Springer, 2018. doi:10.1007/978-3-319-63516-3_8.

34 Jean-Charles Régin and Arnaud Malapert. Parallel constraint programming. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 337–379. Springer, 2018. doi:10.1007/978-3-319-63516-3_9.

35 Martin W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *INFORMS J. Comput.*, 6(4):445–454, 1994. doi:10.1287/IJOC.6.4.445.

36 Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, and Thorsten Koch. Parascip: A parallel extension of SCIP. In Christian H. Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum, editors, *Competence in High Performance Computing 2010 – Proceedings of an International Conference on Competence in High Performance Computing, Schloss Schwetzingen, Germany, June 2010*, pages 135–148. Springer, 2010. doi:10.1007/978-3-642-24025-6_12.

37 Yuji Shinano and Tetsuya Fujie. Paralex: A parallel extension for the CPLEX mixed integer optimizer. In Franck Cappello, Thomas Hérault, and Jack J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 – October 3, 2007, Proceedings*, volume 4757 of *Lecture Notes in Computer Science*, pages 97–106. Springer, 2007. doi:10.1007/978-3-540-75416-9_19.

38 Yuji Shinano, Tetsuya Fujie, and Yuusuke Kounoike. Effectiveness of parallelizing the ILOG-CPLEX mixed integer optimizer in the PUBB2 framework. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings*, volume 2790 of *Lecture Notes in Computer Science*, pages 451–460. Springer, 2003. doi:10.1007/978-3-540-45209-6_67.

39 Yuji Shinano, Stefan Heinz, Stefan Vigerske, and Michael Winkler. Fiberscip – A shared memory parallelization of SCIP. *INFORMS J. Comput.*, 30(1):11–30, 2018. doi:10.1287/IJOC.2017.0762.

40 Laurence A Wolsey and George L Nemhauser. *Integer and combinatorial optimization*. John Wiley & Sons, 1999.

41 Y. Xu, Ted K. Ralphs, Laszlo Ladányi, and Matthew J. Saltzman. Computational experience with a software framework for parallel integer programming. *INFORMS J. Comput.*, 21(3):383–397, 2009. doi:10.1287/IJOC.1090.0347.

Understanding the Impact of Value Selection Heuristics in Scheduling Problems

Tim Luchterhand
LAAS-CNRS, Toulouse, France
Université de Toulouse, France

Emmanuel Hebrard
LAAS-CNRS, Toulouse, France

Sylvie Thiébaux
LAAS-CNRS, Toulouse, France
Université de Toulouse, France
Australian National University, Canberra, Australia

Abstract

It has been observed that value selection heuristics have less impact than other heuristic choices when solving hard combinatorial optimization (CO) problems. It is often thought that this is because more time is spent on unsatisfiable sub-problems where the value ordering is irrelevant. In this paper we investigate this belief in the scheduling domain and come up with a more detailed explanation. We find that, even though there are less relevant choices to be made on hard instances, each mistake tends to have a bigger impact, to a point where the potential gain from a value heuristic predominates. Moreover, we observe two interesting and relatively surprising phenomena when solving scheduling problems. First, the accuracy of a given value selection heuristic decreases with the optimality gap. Second, the computational penalty of a mistake increases with the accuracy of the heuristic. For the first observation, we argue that on hard problems, constraint propagation removes a large portion of choices that align with the intuition behind the heuristic. This means that the heuristic faces mostly difficult choices. For the second observation, we argue that simple heuristics tend to make more mistakes on intuitive choice points, and the computational cost for refuting these mistakes is smaller than for those made by a more accurate heuristic.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling; Computing methodologies → Discrete space search; Computing methodologies → Heuristic function construction; Computing methodologies → Neural networks

Keywords and phrases Scheduling, Branching Heuristics, Constraint Programming

Digital Object Identifier 10.4230/LIPIcs.CP.2025.27

Supplementary Material *Software (Source Code)*: `https://gitlab.laas.fr/roc/emmanuel-hebrard/tempo`, archived at `swh:1:dir:bbab6415fb8f44fef04878de35e7b50e797596bd`

Funding This work was supported by the Artificial and Natural Intelligence Toulouse Institute (ANITI) under the grant agreement ANR-23-IACL-0002, and received funding from the European Union's Horizon Europe Research and Innovation program under the grant agreement TUPLES No 101070149.

1 Introduction

Solvers for combinatorial optimization (CO) problems often strongly rely on branching heuristics that help explore an exponentially large search space efficiently. Usually, these can be categorized into two subgroups: *variable* selection and *value* selection heuristics. The former type selects the next choice point on which a branching decision should be made. The latter type then selects the actual value the variable will take. It is well known that the variable ordering can have a significant impact on the overall performance of the solver.

© Tim Luchterhand, Emmanuel Hebrard, and Sylvie Thiébaux;
licensed under Creative Commons License CC-BY 4.0

This is a reason why over the last decades this research area has enjoyed great popularity, and many effective variable selection heuristics like Variable State Independent Decaying Sum (VSIDS) [31], weighted degree [6] or Learning Rate Branching (LRB) [27] have been proposed. More recently, there have even been advances using machine learning (ML) and deep learning techniques [16, 22, 24].

While there exists work on value selection heuristics, the field did not receive as much attention as variable ordering [5]. The consensus arising from early works (e.g. [17, 18]) is to choose values that lead to the least constrained branches as opposed to variable selection, where usually the most constrained variable should be visited first. So independently of the problem at hand, designing a good value selection heuristic revolves around estimating the future constraining effect of a particular choice of value. The crucial part with these look-ahead value selection strategies is to find a trade-off between accuracy and computational cost. Later works leverage ideas from variable selection like VSIDS and record statistics about values during search. These include activity and impact based heuristics [30, 34] as well as the survivors-first approach of [42]. While the aforementioned heuristics perform better than a random value selection, the impact of other heuristic choices like variable ordering heuristics, nevertheless, dominates the overall performance of the solver. Still, in theory, a perfect value selection would render tree search completely obsolete [15]. One could therefore legitimately expect the potential gain from clever value selection heuristics to be significant. Additionally, in optimization problems, reaching solutions with better objective values faster strengthens constraint propagation based on the dual bound. This in turn can notably reduce the size of the overall search space. Yet, as opposed to variable selection, hardly any generic value selection heuristic has ever been adopted. Even domain-specific value selection strategies that are shown to clearly outperform random selection are relatively rare. For instance, we conducted comprehensive experiments on value selection heuristics on scheduling problems. The results in Figure 1 all show a clear trend: a clever value selection can pay off on easy problems but only marginally outperforms random choices on tight problems.

It has been argued that the shortfall of value selection heuristics often is a high computational cost in the case of look-ahead strategies [42]. But more generally, the usual explanation for the relatively low impact of value selection heuristics, especially on difficult problems, is the fact that the solver spends a lot of time in UNSAT regions of the search space. In these regions, the order of the values explored does not matter [15][1].

In this paper however, we argue that this cannot be the sole reason for the diminishing impact of value selection. While we confirm that on problems close to the phase transition [9] the solver spends most of the time in UNSAT subtrees, we also found that the remaining relevant choices tend to be crucial in finding a solution. This goes to a point where the penalty for getting these choices wrong dominates the search performance. As a consequence, a consistently accurate value selection heuristic that can avoid those errors would have a noticeable positive impact on the overall performance, especially on difficult instances.

Furthermore, we observed two interesting and rather surprising phenomena when solving scheduling problems as a sequence of satisfaction problems with an increasingly tighter bound. The first concerns the *online accuracy* of value selection heuristics which we define as the ratio between the number of *correct* branches chosen and the number of all *relevant*

[1] It may have an indirect influence via clause learning or weight-based heuristics. Nevertheless, in both cases there is no known positive correlation with some parameter that could help us decide a priori which branch is best.

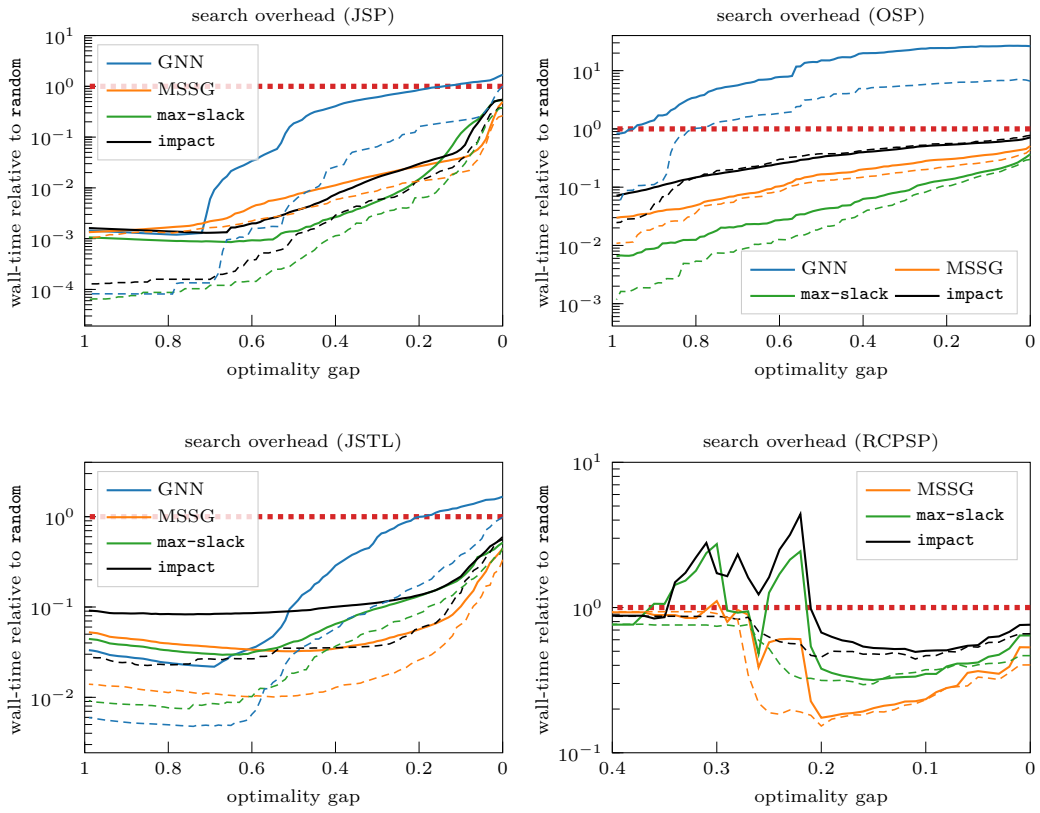


Figure 1 Average / median solve time with respect to `random` value selection on different sets of scheduling problems. Dashed lines represent median values. A point (x, y) is interpreted as: "heuristic h takes y times as much time as `random` to get to a gap of x". The red dotted line marks the performance of the `random` baseline. Details about the heuristics and the problem instances can be found in Section 3.1.

choice points. We say a choice point is relevant when there exists both a value selection for which the resulting subtree contains a solution and one for which it doesn't. A correct choice obviously occurs when the heuristic selects the value that spans a SAT subtree. In our experiments, we found that this accuracy degrades for all tested, informed heuristics as the optimality gap decreases and the problem becomes more constrained. This again means that the heuristics tend to be less accurate when the stakes are highest, i.e. when the gap is small and in turn the cost for making the wrong decision is higher than when the gap is large. Secondly, we noticed that the cost for making a wrong decision does not only depend on the difficulty of the problem but also on the accuracy of the heuristic. Our experiments suggest that, on the one hand, simple heuristics usually make more mistakes than more sophisticated ones, especially on *easy choices*, i.e. choices that follow a clear intuition. Yet, these errors are just as easily refuted, meaning that the cost for failing is low. More intelligent heuristics on the other hand, make less mistakes but when they do, it usually takes disproportionately more effort to notice and correct the error.

To support these claims, we present an in-depth analysis of the behavior of different value selection strategies on a large set of scheduling benchmarks with both disjunctive and cumulative resources. We additionally demonstrate that the mentioned phenomena also arise in experiments with an ML-based heuristic.

Our results show a more complete picture on why developing impactful value selection heuristics is difficult and identify an interplay of three key observations. First, as the optimality gap decreases, the penalty for selecting a wrong value increases. This is because wrong branches lead to increasingly deep UNSAT subtrees. Conversely, when the gap is large, these UNSAT subtrees tend to be a lot more shallow. So in theory, a heuristic with constant value prediction accuracy should have more impact the harder the problem gets because selecting the correct values avoids spending a long time in uninteresting regions. Which brings us to the second observation, namely the fact that the heuristic accuracy does *not* stay constant during the search but rather decreases with the optimality gap. This means that any heuristic will not only face more important choices but also get them wrong more often. And finally, the penalty for a wrong decision positively correlates with the accuracy of the heuristic, i.e. more involved heuristics tend to make less errors in total, but pay a disproportionately higher price for being wrong.

The remainder of this paper is structured as follows. We first introduce the metrics we used in our experiments as well as our procedure for measuring them. We then present the different heuristics we analyzed and the details of our experimental setup. Finally, we conclude the paper with a discussion of the results.

2 Analyzing the Behavior of Value Selection Heuristics

Within the scope of this paper we analyze the impact of value selection strategies on solving different types of scheduling problems. We employ a self-developed constraint programming (CP) solver we call `tempo`[2]. It represents problems using binary and numeric variables. In the case of scheduling problems, numeric variables are used to model events in a simple temporal network (STN) [11]. Each task t is composed of two events: a start event s_t and an end event e_t. Constraints are represented by edges in the form of maximum distances between the events. For example, given two events a, b, the constraint $a \prec b$ (a before b) is represented by an edge starting in b and ending in a annotated with a negative distance. Furthermore, numeric variables are also used to model resource consumption and capacities in the case of cumulative scheduling problems. The objective is modelled as one global schedule interval that must contain all tasks. The objective variable is the duration of said schedule. As many constraint optimization solvers [33, 29, 32, 10], `tempo` minimizes the objective value by solving a sequence of satisfiability problems with an increasingly tighter upper bound on the objective variable.

Binary variables on the other hand represent the existence of edges in the STN and are used as branching points during search. For example, in a disjunctive scheduling problem the binary variable $x_{t,t'}$ may represent the ordering of two tasks t and t'. Concretely, $x_{t,t'} = 1 \Leftrightarrow e_t \prec s_{t'}$, i.e. t must finish before t' starts, and $x_{t,t'} = 0 \Leftrightarrow e_{t'} \prec s_t$. `tempo` makes binary branching decisions which have been shown to be superior to d-way branching in constraint satisfaction problem (CSP) solvers [21]. This means that a binary search tree is spanned where at each node a variable is selected by the *variable* selection strategy and then assigned by the *value* selection heuristic. `tempo` uses the state-of-the-art variable selection strategy VSIDS [31]. It also leverages clause learning or no-good-learning [28] and dynamic restarts [19]. Whenever a literal appears in a conflict clause or in a resolution step leading to said clause, its activity value is updated. When clause learning is switched off, `tempo` uses

[2] The public repository can be found at `https://gitlab.laas.fr/roc/emmanuel-hebrard/tempo`. The experiments were conducted using the commit identified by the tag `cp_2025_val_selection_exp`.

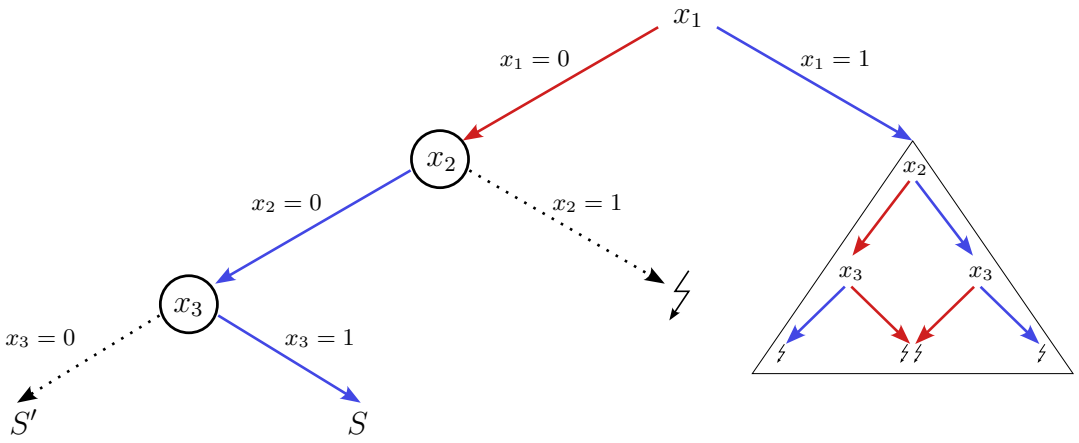


Figure 2 Depiction of the analysis procedure. Blue arrows are heuristic choices, red arrows are refutations. Circled variables need to be classified as either ICS or *relevant*. Only the choice points x_1 and x_2 are relevant. The ↯ symbols signify UNSAT branches. The triangle shows a maximal UNSAT subtree. Note that the dashed arrows are not actually explored by `tempo` but need to be visited for the classification.

explanation based weighted degree [6] instead of VSIDS for variable selection. This is possible since each fail still generates a clause that we use to update literal weights. Finally, we use an edge-finding algorithm based on [40] for propagating the disjunctive resource constraint. On cumulative resources we use an algorithm based on [26]. Without going into detail, it leverages overlaps between tasks. These are usually difficult to detect from numeric variables representing start times of tasks but much easier from ordering variables.

All in all, `tempo` achieves state-of-the-art performance on the problems treated in this paper. Using a custom solver allows us to easily access detailed information that we need for our analysis such as when and where conflicts are encountered and which specific choices were made by the heuristics.

2.1 Relevant and Irrelevant Heuristic Choices

Since `tempo` uses two-way branching, solving each satisfaction problem produces a binary search tree. When a solution is found the search is restarted from the root of the tree so that the new tighter upper bound constraint does not have to be retroactively applied to the search states. The overall search tree is then the concatenation of the trees corresponding to each satisfiability problem. Each of these search trees has the form shown in Figure 2. We measure the performance of a given value selection heuristic in each satisfiability problem by analyzing the corresponding search tree. To that end, we categorize the choice points. The branch chosen by the value selection heuristic is called a *choice* (blue arrows in Figure 2). A *refutation* is the opposite of a choice (marked with red in the illustration). A choice is *correct* when it spans a subtree that contains a solution and incorrect when it doesn't. However, some of these choices, correct or not, may be irrelevant. We say a choice is *relevant* if and only if its refutation would have lead to a different outcome. That is, if the subtree spanned by the choice contains a solution and the one spanned by the refutation does not or vice versa. Consequently, all choices in an UNSAT subtree are irrelevant because neither of the branches leads to a solution. This type of choice is called irrelevant choice in an UNSAT subtree (ICU). Moreover, some correct choices are irrelevant if the refutation also leads to a

solution. These are called irrelevant choices in a SAT subtree (ICS's). In theory, one could further distinguish correct choices, i.e. whether both branches lead to equally good solutions (in terms of objective value) or not. However, we are primarily interested in the solver not getting stuck in UNSAT subtrees and therefore do not make this distinction.

2.2 Value Selection Heuristic Accuracy

Each time `tempo` finds a solution S we get the following information.

1. The total number of heuristic choices T in the current descent. In Figure 2 this is simply the number of blue arrows.
2. The number of correct choices C, i.e. two in the example.
3. The length L of the branch leading to the solution S (three in the example). Note that L is in general less than the number of variables since constraint propagation will fix some variables.
4. The number of ICUs (the three blue arrows in the triangle in Figure 2) which is simply $T - L$.

In order to obtain the number of ICS's we need to further analyze the correct choices (circled variables in Figure 2). To analyze a choice on a variable x, we define a satisfiability problem based on the initial problem description while adding all choices and refutations leading from the root to x as constraints. For example, to classify x_3 in Figure 2 we would add the constraints $x_1 = 0$ and $x_2 = 0$ as well as the previous upper bound on the objective and then call a satisfiability oracle. If the oracle determines that the problem is SAT, i.e. by finding a different improving solution S' (as in the figure), the choice is an ICS. Having obtained the number of ICS's, we can finally calculate the number of *relevant* choices

$$R = T - \#\text{ICU} - \#\text{ICS} = L - \#\text{ICS} \tag{1}$$

and the number of relevant and correct choices

$$R_C = C - \#\text{ICS}. \tag{2}$$

In our running example, we would get $R = 2$ and $R_C = 1$. Based on this information, we then define the *accuracy* of a value selection heuristic h simply as the ratio of relevant and correct choices to the number of relevant choices:

$$\text{accuracy}(h) = \frac{R_C}{R}. \tag{3}$$

In the example shown in Figure 2, the accuracy would be 50%, given that there are two relevant choices of which one is wrong. We can monitor this accuracy as the search progresses since `tempo` logs the required information each time an improving solution is found. We call the progression of the accuracy over the optimality gap *online* accuracy. Furthermore, we are interested in the cost of a wrong relevant choice. We define this *error penalty* P proportional to the average size of a maximal UNSAT subtree which is simply the number of ICUs divided by the number of mistakes. Since we want to compare this number across problems of different size, we normalize it by additionally dividing by the total number of choices. Thus, we get

$$P = \begin{cases} \frac{\#\text{ICU}}{T(R-R_C)}, & \text{if } R_C < R, \\ 0, & \text{otherwise} \end{cases}. \tag{4}$$

In Figure 2, the average size of maximal UNSAT subtrees would be three and $P = 0.5$. This measure gives us an indication of how bad an error by the heuristic is on average, i.e. how much time is wasted in UNSAT regions.

While our procedure gives us detailed information about the behavior of the value selection heuristic, it is also computationally expensive since each oracle call involves solving a SAT problem. Still, the number of oracle calls is only $\mathcal{O}(L)$ and L is typically much smaller than the number of variables due to constraint propagation. Also notice that this procedure is not exact on search trees developed by conflict driven clause learning solvers. This is due to the fact that the refutation of a choice $x = v$, where v is some value, is not necessarily $x \neq v$. More precisely, a backjump from the choice $x = v$ may jump to a choice point on a different variable y. This means that the decision $x = v$ was not necessarily wrong. Also, all the decisions strictly between the backjump level and the decision level are not well categorized. Therefore, when applying this method to a search tree using learning, one should be aware that the results are only approximations.

3 Experiments

In our experiments we analyzed the behavior of different value selection heuristics on a set of scheduling problems. These can be viewed as a sequence of satisfiability problems that get progressively harder as the upper bound on the objective becomes tighter. In accordance with the theory on phase transitions [9] we identify the optimality gap of our objective value as an order parameter and argue that for small values the resulting satisfiability problems are close to the phase transition. This is reasonable given that the number of solutions and thus the probability of finding one decreases with the gap.

The remaining section is structured into five parts. First we present the studied heuristics and our experimental setup. In Sections 3.2 and 3.3 we then continue with a detailed analysis on the progression of the number of irrelevant and relevant choices during search. Next, we move on to our main observations in Section 3.4. And finally, we conclude our experiments by discussing our observations and their implications.

3.1 Setup

In our experiments we analyzed four different heuristics. In the following, we explain their behavior given a binary variable x whose values 1 and 0 correspond to the event precedence constraints $a \prec b$ and $c \prec d$ respectively. As a baseline we use the `random` heuristic that, as the name suggest, chooses values randomly with uniform probability.

First, the `max-slack` heuristic uses the following branching rule:

$$\texttt{max-slack}(x) = \begin{cases} 1, & \text{if } \max(b) - \min(a) \geq \max(d) - \min(c) \\ 0, & \text{otherwise,} \end{cases} \tag{5}$$

where $\min(a)$ and $\max(a)$ are the minimum and maximum values of the domain of the numeric variable a respectively. Intuitively, the heuristic fixes the edge that leaves the largest possible slack in the temporal network. The variable $x_{t,t'}$ in a disjunctive scheduling problem for example represents the ordering of the tasks t, t': $x_{t,t'} = 1 \Leftrightarrow e_t \prec s_{t'}, x_{t,t'} = 0 \Leftrightarrow e_{t'} \prec s_t$. `max-slack` would therefore schedule $t \prec t'$ if $\max(s_{t'}) - \min(e_t) \geq \max(s_t) - \min(e_{t'})$.

Second, we use an impact based heuristic similar to [34]. It keeps a score $i_{\mathtt{x}}$ for each binary literal $\mathtt{x} \Leftrightarrow x = 1, \neg\mathtt{x} \Leftrightarrow x = 0$ in the problem, which is initially zero. Whenever the heuristic assigns a value to x, we count the number of other variables fixed by subsequent constraint propagation and update the score[3]:

$$i_{\mathtt{x}}^{(k+1)} = \frac{i_{\mathtt{x}}^{(k)} + \#\text{ variables fixed by propagation}}{2}. \tag{6}$$

This score is used by the `impact` heuristic to make decisions according to

$$\texttt{impact}(x) = \begin{cases} 1, \text{ if } i_{\mathtt{x}} < i_{\neg\mathtt{x}} \\ 0, \text{ otherwise.} \end{cases} \tag{7}$$

The intuition is somewhat similar to that behind `max-slack`: fix variables in a way that leaves the most freedom for future decisions. The problem with using this rule right from the beginning is that the impact scores are all zero. Our actual implementation of `impact` therefore only follows Equation (7) after a short warm-up period, i.e. after 50% of all literals have been selected at least once, and falls back on `max-slack` otherwise.

Next, we combine `max-slack` and solution-guided search [4] which we call `max-slack` solution guided (MSSG). Initially, MSSG uses `max-slack` until a solution $S = (x_i)_{i=1}^{N}$ ($N \in \mathbb{N}$ the number of binary variables) is found. This solution is then saved. From this point on, MSSG tries to keep the variable assignment of S. Since at some point the search needs to deviate from the path to the last solution due to the tighter upper bound, MSSG is permitted to fall back to `max-slack` if the discrepancy between the S and the current variable assignment $\tilde{S} = (\tilde{x}_i)_{i\in I}, I \subset \{1, \ldots, N\}$ becomes too big. This discrepancy is defined as

$$\text{discrepancy}\left(S, \tilde{S}\right) = \begin{cases} 0, \ I = \emptyset, \\ \frac{1}{|I|}\sum_{i\in I} 1 - \delta_{x_i\tilde{x}_i}, \ I \neq \emptyset \end{cases}, \ \delta \text{ the Kronecker delta,} \tag{8}$$

i.e. the percentage of assigned variables whose values differ from the corresponding ones in S. In our experiments we set this discrepancy threshold to 1%. So in short, MSSG behaves the following way:

$$\text{MSSG}(x) = \begin{cases} \texttt{max-slack}(x), \text{ if no solution found yet or discrepancy } > 1\%, \\ 1, \text{ if } \mathtt{x} \in S, \\ 0, \text{ if } \neg\mathtt{x} \in S. \end{cases} \tag{9}$$

Each time a new improving solution is found, S is updated.

Finally, inspired by the works of [38, 39], we tested a heuristic based on a pre-trained graph neural network (GNN). Without going into too much detail, the GNN exploits a graph representation of the problem, much like the STN. Nodes represent tasks as well as resources, edges between tasks describe precedence relations and are annotated with timing information. Edges between tasks and resources describe resource dependencies and are also annotated with features like resource demand and capacity. From the graph, the GNN calculates a graph embedding using a message passing scheme similar to that in [3]. From this embedding the binary values for each edge are derived using a multi-layer perceptron (MLP). Since running the GNN is costly, the inference is only run at the very beginning of the search. Additionally, it is rerun whenever a solution is found which tightens the upper bound on the objective.

[3] Our implementation differs from that of the original paper for reasons of simplicity.

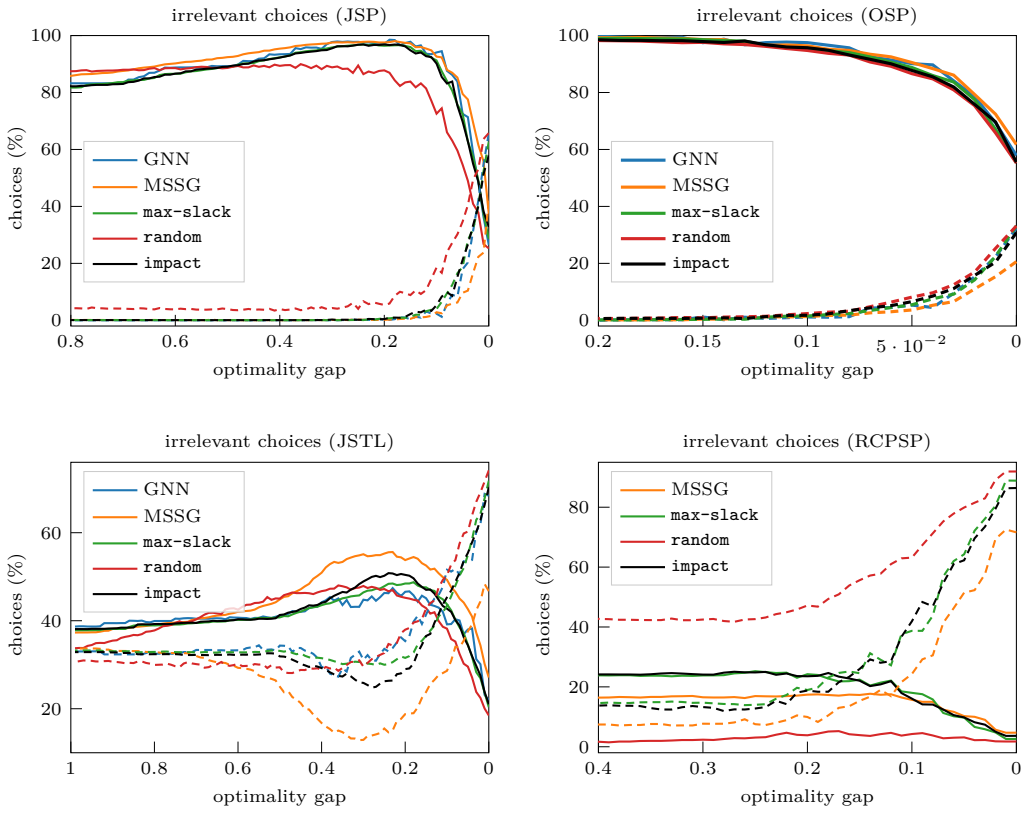


Figure 3 Mean percentage of irrelevant choices distinguished by category. Bold lines represent the percentage of ICS's, dashed lines represent the percentage of ICUs.

We evaluated these heuristics on four sets of different scheduling problem types: 48 Open Shop Problems (OSPs) from [7, 20, 43], 24 Job Shop Problem (JSP) instances from [1, 14, 2, 25, 37, 43, 41], 58 Job Shop instances with time lags (JSTL) from [8] and 20 resource-constrained project scheduling problem (RCPSP) instances from [23]. A complete list of the instances used is given in Table 1. We ran `tempo` with each of the five heuristics on each of those instances with ten different random seeds, totaling in 1500 runs for each heuristic. An exception to this is the GNN based heuristic that, at the time of the experiments, could not handle cumulative resources, which is why we do not present results for the GNN on RCPSPs. Each run was given a timeout of 30 minutes. The results in the next section were produced without using clause learning in order to avoid the problem mentioned Section 2.2. However, results with clause learning can be found in Section A.

3.2 Progression of Irrelevant Choices

We begin our analysis by addressing the belief that the solver spends increasingly more time in UNSAT subtrees as the optimality gap closes. Even though our results seem to align with this hypothesis, they draw a more complete picture as we analyze ICUs and ICS's separately. Figure 3 shows the progression of the share of irrelevant decisions over the course of solving the problem.

First, we can see that, in all four problem types, the percentage of ICUs (dashed lines) increases towards smaller gaps. At the same time, the percentage of ICS's (bold lines) decreases. It is also visible that this effect depends on the type of problem. On less constrained instances like OSPs or JSPs, initially nearly all choices lead to improving solutions and are therefore irrelevant. This is most pronounced on OSPs due to their symmetry. When the gap decreases, the upper bound on the objective variable tightens, making the problem more constrained and ultimately leading to increasingly more failures and therefore more ICUs. Interestingly, on JSP instances, the ratio of ICS's actually appears to increase slowly before falling back off towards the end (not counting `random`). One possible explanation is that propagation removes more and more choices and helps the heuristics guide the solver into *easy regions* where most choices are ICS's, as long as the gap is not too small.

The same overall trend can be observed on the JSTL and RCPSP instances. However, the levels of ICS's and ICUs are initially much closer together. In JSTLs there exist backwards links between tasks in the STN due to the time lags. These can easily cause negative timing cycles and thus failures which explains the higher initial percentage of ICUs and the lower amount of ICS's. For the RCPSP instances, both types of irrelevant choices initially make up a comparatively low share or conversely: there are more relevant choices than in any other problem type in the beginning of the search. This is due to a particularity of the resource model used that only performs very limited forward checking. To give an example, let's say we have a resource with capacity 10 and three tasks t_1, t_2, t_3 that consume quantities of 3, 5 and 6 resource units respectively. Let's also say that t_1 and t_2 are currently assigned to run in parallel. The model does not forbid to also schedule t_3 in parallel, even though it would result in a trivial failure. This means that a lot of relevant but rather trivial choices are created.

Perhaps more interestingly, the amount of irrelevant choices not only depends on the gap but also on the heuristic itself. Looking at `random` for example, it produces the most ICUs and the least ICS's on both JSP and RCPSP instances. This makes sense because `random` is arguably a bad heuristic that makes a lot of mistakes. As a result, the solver will spend comparatively more time in UNSAT regions, leading to more ICUs. Conversely, `random` is less likely to steer the solver into regions where any choice leads to a solution. This effect, while less pronounced, is also present in the JSTL instances, at least towards smaller gaps. This is probably due to the fact that all heuristics have a harder time making good decisions on this type of instance which is why the difference is less noticeable. The same can be seen on the OSP instances, albeit for the opposite reason. As mentioned before, OSPs have a high degree of symmetry, meaning that when the gap is large basically any decision, random or not, leads to a solution. MSSG on the other hand behaves quite the opposite way. As we will see later, it has one of the highest online accuracies. With this in mind, it seems logical that it exhibits high ICS and low ICU percentages on all four problem types.

3.3 Progression of Relevant Choices

Our observations thus far coincide with what is accepted in the literature. Things become more interesting when we combine both the curves for ICS's and ICUs, or put differently, when we look at the number of *relevant* choices. This can be seen in Figure 4. The results on the RCPSP are arguably the most intuitive. The amount of relevant choices starts out rather high and then progressively diminishes as the gap closes. The opposite can be seen on the OSP instances. Again, it is apparent that on these problems, initially virtually everything is a solution and choices only start to matter when the gap becomes small. On both types of Job Shop problems, the results are a bit more complicated to interpret, and the differences

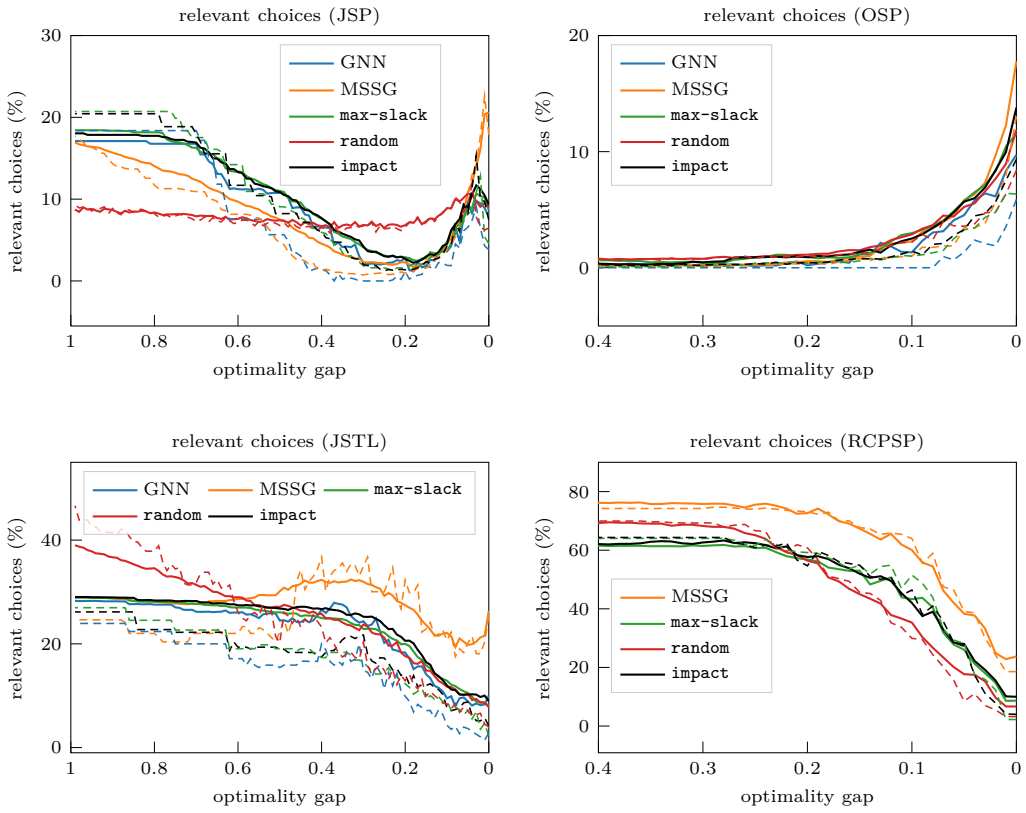


Figure 4 Percentage of relevant choices. Bold lines represent mean values and dashed lines median values.

between the heuristics are more pronounced. On the JSTLs the percentage of relevant choices still follows a clear downward trend except when using solution guided search. Interestingly, `random` has the highest rate of relevant choices in the beginning. Comparing its graph with the one form Figure 3, we can see that this is due to `random` exhibiting both the lowest percentage of ICS's and ICUs. It appears logical that `random`, being a bad heuristic, rarely guides the solver into a subtree with many ICS's. Nevertheless, it seems strange that it initially also exhibits the lowest level of ICUs. One possible explanation could be that `random` makes a lot of trivial mistakes that are easily refuted. Consequently, the solver doesn't spend a long time in an UNSAT region after a wrong decision. MSSG on the other hand, maintains a relatively constant level of relevant choices over the whole range of gap values. There is even a part where the percentage of relevant choices increases over the initial level. Again, comparing the corresponding curves in Figure 3 we can see that this arises from a very low level of ICUs and implies that MSSG provides good guidance. When the gap approaches zero, the number of relevant choices remains at about the same level as in the beginning. At this point, there remain much fewer ICS's in the overall search space. The fact that the number of relevant choices remains high means that MSSG manages to keep the search in regions where it is still possible to find improving solutions, albeit only through clever choices. Finally, on the JSP instances there is a clear difference between `random` and the other heuristics. As on the OSPs, these problems are not very constrained when the gap is small which means that most of the choices are ICS's. The difference between `random` and

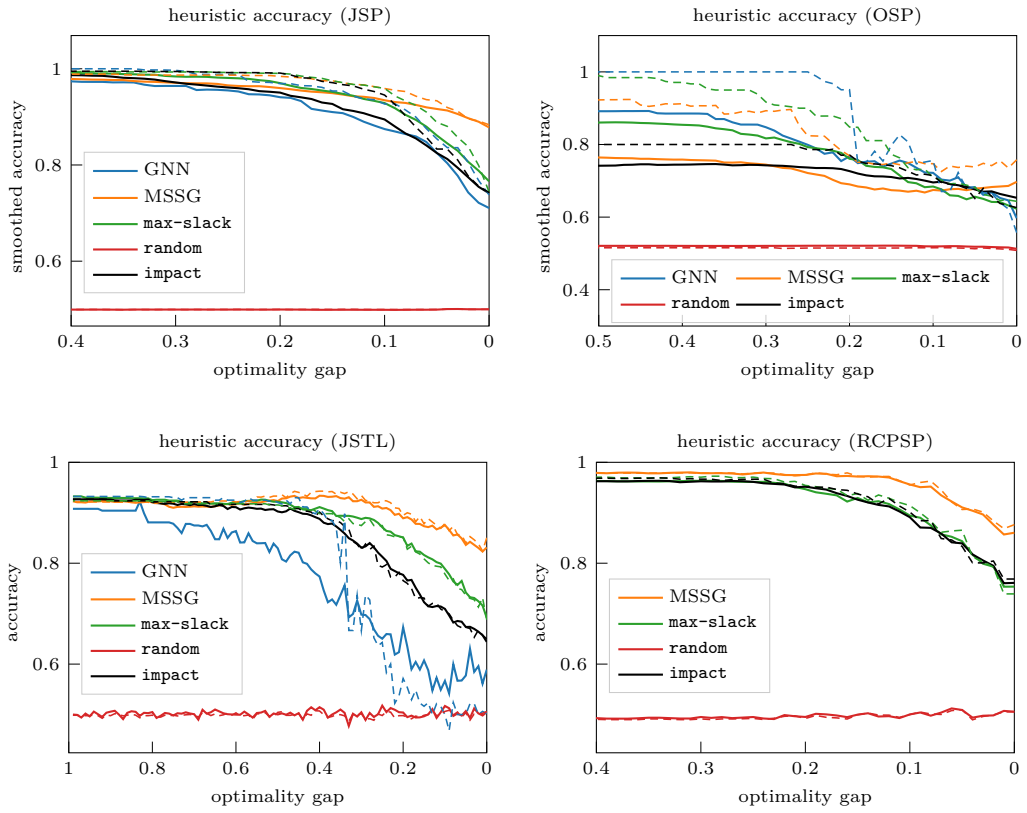


Figure 5 Online accuracy over optimality gap. Bold lines represent mean values and dashed lines median values. In the two top figures, the accuracy has been smoothed using a biased average.

the other heuristics arises from the fact that `random` makes more errors and thus also has a higher level of ICUs right from the beginning. What is worth of notice is that for very small gap values, the percentage of relevant choices actually rises back up, in the case of MSSG even higher than the initial ratio.

To summarize the results this far, we confirm that all heuristics spend more time in UNSAT subtrees the smaller the gap becomes. Still, the distinction between ICS's and ICUs draws a more complete picture and reveals more details about the behavior of the different heuristics across the four instance types. The results also differ somewhat from the intuition in the literature, especially when looking at the number of relevant choices. Only on RCPSP instances we observe a clear downward trend of the heuristic's impact as the gap shrinks. On the other problem types, the trend is either less clear (JSTL), inverses towards the end (JSP) or goes in the complete opposite direction (OSP). On the latter three types, one would expect the heuristics to have a greater impact on the performance of the solver, provided of course that their accuracy does not change. However, the results we will discuss next show that this is precisely not the case.

3.4 Online Accuracy and Error Penalty

In the following part, we only focus on the relevant choices and analyze the actual online accuracy of the heuristics presented in Figure 5. As expected, `random` has a constant accuracy of 50% all the time on all four problem types. All the other heuristics on the other hand

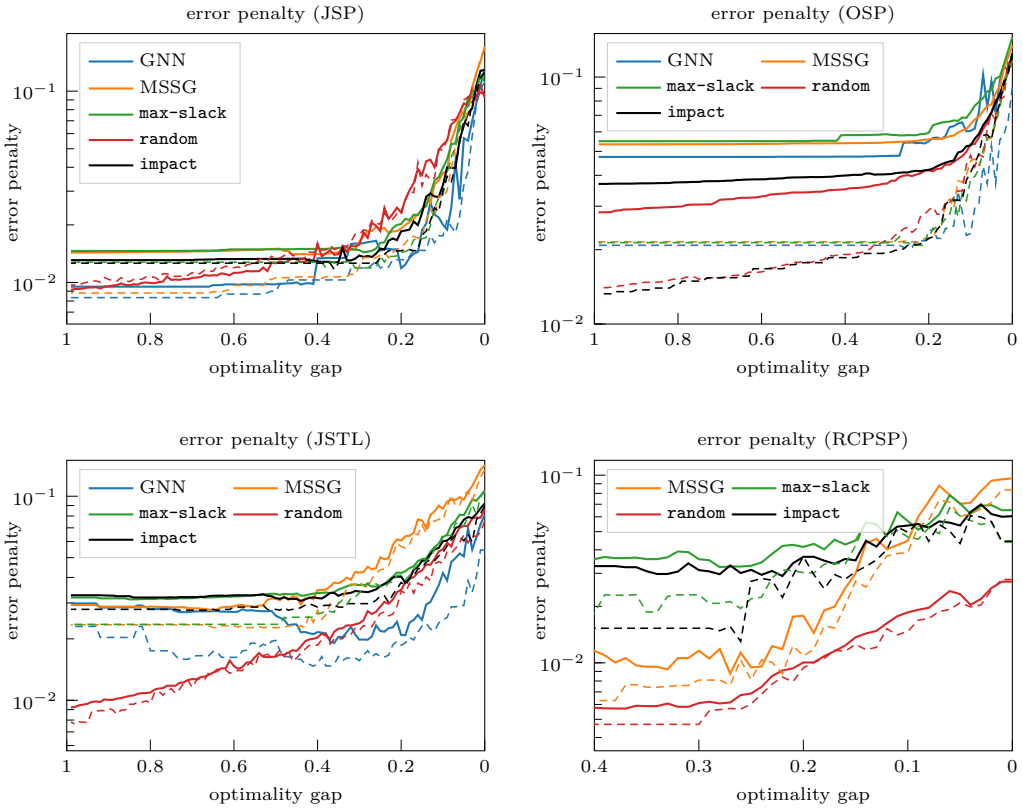


Figure 6 Error penalty over optimality gap. Bold lines represent mean values and dashed lines median values.

show a clear drop in accuracy as the gap becomes smaller. MSSG seems to suffer the least from this problem. So apparently, the relevant choices the heuristics face become more and more difficult over the course of the search. We hypothesize that this happens due to an *alignment* between constraint propagation and value selection heuristics. In short, constraint propagation often prunes decisions that are in line with the heuristic. Thus, when the problem is tight and constraint propagation is strong, fewer intuitive decisions are left to the heuristic which ultimately results in a lower accuracy. We go into more detail in Section 3.5.

Finally, not only does the heuristic accuracy decrease with the gap, it is also lowest when the stakes are highest. Figure 6 shows the progression of the error penalty for the different heuristics. As expected, this penalty increases towards smaller gap values on all instances and for all heuristics. This means that the average size of the UNSAT subtrees becomes bigger and mistakes generally occur earlier in the search. In some cases, we can observe an increase of about a factor of ten (MSSG on JSP or RCPSP). This observation is in line with the theory on phase transitions [9]. There exists work on the notion of exceptionally hard problems [35], i.e. problems that are located to the far left of the phase transition and still are very difficult. However, we did not encounter them in our experiments. A possible explanation for this is that their difficulty is not necessarily a property of the problems but also depends on the algorithm and heuristics used [36].

So even though on some instance types the percentage of relevant choices decreases as we approach the optimum, the remaining relevant choices tend to be far more impactful. Moreover, the error penalty also depends on the heuristics themselves. When comparing Figure 5 and Figure 6, it becomes clear that the most accurate heuristic (MSSG except on OSP instances) also suffers the biggest error penalty. In contrast, while the error penalty for `random` also increases, it usually starts out lowest of all heuristics and ends at smaller values than MSSG.

For the sake of completeness, we want to mention that we also did some experiments with `tempo` with enabled no-good-learning. The results, however, are not accurate as discussed in Section 2.2. We will therefore not analyze them in detail here, but they can be found in Section A. The observations mostly coincide with those made in this chapter.

3.5 Discussion

In summary, our experiments reveal three key observations. First, while the percentage of irrelevant choices in an UNSAT subtree (ICUs) does increase towards smaller gap values, the percentage of relevant choices does not necessarily decrease at the same time for all problem types. We found it to be the case on some instance types like RCPSPs and quite the opposite on OSPs. Moreover, this effect depends on the heuristics themselves: better heuristics tend to face more relevant choices. Second, the accuracy of all informed heuristics, that excludes `random`, becomes less accurate the tighter the problem becomes. And finally, the error penalty, i.e. the time spent in an UNSAT subtree after a wrong heuristic decision, increases as the gap decreases. This effect is most pronounced for accurate heuristics.

These three phenomena and especially their interplay give us a better understanding of why designing impactful generic value selection heuristics is difficult. From our results it seems clear that the fact that the solver spends most of the time in UNSAT subtrees cannot be the only explanation for the limited impact of these heuristics on hard instances. A consistently accurate heuristic should be especially impactful since it would be able to avoid the increasingly large error penalties. This would create somewhat of a positive feedback loop. Fewer errors lead to more relevant choices that the heuristic gets mostly right given its high accuracy which in turn leads to more relevant choices and so on. This is obviously not the case. There are even instance types where the overall percentage of relevant choices does not decrease, yet still the search slows down towards smaller gap values.

We therefore present an additional argument, namely that the branching choices become increasingly non-intuitive the harder the problem gets. This degradation of the heuristic accuracy can be clearly seen in Figure 5. Our explanation for this is that heuristic branching choices often *align* with constraint propagation. For instance, the `max-slack` heuristic suggests to sequence two events in a way that leaves the most slack in the timing network, i.e. it prefers to keep large positive timing values. Notice that constraint propagation of binary disjunctions precisely excludes any ordering decision that would lead to negative slack. In general, the principle behind value branching is to first explore branches that are less likely to be inconsistent and hence less likely to be pruned. Therefore, decisions pruned by constraint propagation are overwhelmingly those that the value selection heuristic would have gotten right anyway. As a result, when the gap decreases and the tightness of the problem increases, constraint propagation gets stronger and is increasingly prevalent in the shape of the search tree which ultimately means that the heuristic will more likely make the wrong decision on the choices points that are left. Or put differently, due to the alignment with constraint propagation, decisions that go against the intuition behind the heuristic become more likely. Conversely, if the problem is not tight (large gap), constraint propagation is less strong and

more easy choices that follow a clear intuition are left to the heuristic. In future work, it would be interesting to see whether this phenomenon can be reproduced in settings where much weaker propagation is used. In our case, it seems to arise for all heuristics, be they relatively simple like `max-slack` or more complex like the GNN. A very accurate heuristic should still get more of the difficult choices right. But given the nature of the problem, a heuristic that accurately models an exponential number of choices is either impossible to design or too costly to run.

The third observation, i.e. the increasing error penalty, only amplifies the effect of the degrading accuracy. What's most noteworthy is the fact that the penalty again depends on the heuristic. As described previously, when the optimality gap is small, the more accurate heuristics pay a disproportionately high price for selecting the wrong value. Our hypothesis is that bad heuristics like `random` make more errors on the one hand that are easily refuted on the other hand. In contrast, more accurate heuristics make decisions that initially seem correct but turn out to be wrong much later in the search. In a way this effect offsets the advantage of a high accuracy and could explain why we have yet to see a superior generic value selection heuristic. It also gives a new perspective on reasoning about the effectiveness of these heuristics. The fact that one heuristic is more accurate than another on paper does not necessarily translate to a better performance in practice.

4 Conclusion and Future Work

In this paper we addressed a common belief in the literature that explains why value selection heuristics generally have a lower impact on the performance of a CO solver than other heuristic choices like for example variable selection. To that end we analyzed different value selection heuristics on a set of scheduling benchmarks. In our experiments, we were able to confirm that, when the optimality gap shrinks, the solver *does* spend more time in unsatisfiable subtrees where the value ordering effectively has no impact. We also argued that this cannot be the sole reason for the diminishing influence of value selection heuristics and presented further interesting phenomena in our results. First, the heuristics become less accurate the smaller the gap, most likely due to the interaction with constraint propagation that prunes away many easy branching choices and leaves the heuristics with mostly unintuitive decisions. What makes this worse is our second observation, namely that the penalty for selecting the wrong value strongly increases for small optimality gaps. This means that the heuristics fail most often when the stakes are highest. And finally, this error penalty also positively correlates with the heuristic's accuracy, meaning that better heuristics pay a higher price for making mistakes. These observations, on the one hand, explain why the impact of value selection heuristics diminishes on harder problems. On the other hand, they explain why the gain in performance from an accurate heuristic is not necessarily as high as expected in practice. This should be kept in mind when employing methods like ML to improve heuristic parts of the solver. While ML has been successfully used for variable selection, using it for value selection seems far less promising. In fact, we found that our GNN based heuristic was unable to provide any significant advantage – or depending on the instance type any advantage at all – over the other much simpler heuristics.

At this point we want to stress that our results neither suggest that further research on value selection heuristics is pointless nor do they contradict works on sophisticated heuristics like [13, 12]. We simply made some interesting observations that hopefully will help to better understand the impact of value selection in CO problems like scheduling. To give

some concrete advice based on our findings: if one is dealing with large industrial problem instances where finding an optimum is futile, a clever value selection heuristic that aims to quickly find a high quality solution may prove very effective. Conversely, if optimality is of interest, "dumb but fast" is probably the way to go.

Finally, we acknowledge that our results are restricted to the scheduling domain and in some parts to the solver implementation we used. Specifically, our solver `tempo` uses 2-way branching and strong propagation mechanisms, and our analysis procedure only obtains approximate results when using no-good-learning and backjumping. It should be possible to develop an algorithm that can obtain exact results even with backjumps in order to confirm our findings. Such a procedure surely exists, even though it might be more computationally demanding. More importantly however, we think it would be interesting to extend our experiments to a broader setting, i.e. general CSPs with arbitrary value domains and different phase transitions. Our explanations for the observations made in this paper are not directly tied to scheduling which is why we believe that similar phenomena might be observed on different problem types as well.

References

1 Joseph Adams, Egon Balas, and Daniel Zawack. The Shifting Bottleneck Procedure for Job Shop Scheduling. *Management Science*, 34(3):391–401, 1988. URL: `http://www.jstor.org/stable/2632051`.

2 David Applegate and William Cook. A Computational Study of the Job-Shop Scheduling Problem. *ORSA Journal on Computing*, 3(2):149–156, May 1991. `doi:10.1287/ijoc.3.2.149`.

3 Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks, June 2018. `arXiv:1806.01261`.

4 J. Christopher Beck. Solution-Guided Multi-Point Constructive Search for Job Shop Scheduling. *Journal of Artificial Intelligence Research*, 29:49–77, May 2007. `doi:10.1613/jair.2169`.

5 Christian Bessière and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Principles and Practice of Constraint Programming — CP96*, pages 61–75, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. `doi:10.1007/3-540-61551-2_66`.

6 Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, pages 146–150, NLD, 2004. IOS Press.

7 Peter Brucker, Johann Hurink, Bernd Jurisch, and Birgit Wöstmann. A Branch & Bound Algorithm for the Open-shop Problem. *Discrete Applied Mathematics*, 76(1):43–59, 1997. Second International Colloquium on Graphs and Optimization. `doi:10.1016/S0166-218X(96)00116-3`.

8 Anthony Caumond, Philippe Lacomme, and Nikolay Tchernev. A memetic algorithm for the job-shop with time-lags. *Computers & Operations Research*, 35(7):2331–2356, July 2008. `doi:10.1016/j.cor.2006.11.007`.

9 Peter C. Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence. Sydney, Australia, August 24-30, 1991*, pages 331–340. Morgan Kaufmann, 1991. URL: `http://ijcai.org/Proceedings/91-1/Papers/052.pdf`.

10 Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, a lazy clause generation solver. `https://github.com/chuffed/chuffed`, 2018.

11 Rina Dechter, Itay Meiri, and Judea Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49(1-3):61–95, 1991. `doi:10.1016/0004-3702(91)90006-6`.

12 Augustin Delecluse and Pierre Schaus. Black-Box Value Heuristics for Solving Optimization Problems with Constraint Programming. In *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 36:1–36:12, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.CP.2024.36`.

13 Jean-Guillaume Fages and Charles Prud'Homme. Making the First Solution Good! In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1073–1077, 2017. `doi:10.1109/ICTAI.2017.00164`.

14 H. Fisher and G.L. Thompson. Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules. In *Industrial Scheduling*, pages 225–251. Prentice-Hall, Englewood Cliffs, 1963.

15 Daniel Frost and Rina Dechter. Look-Ahead Value Ordering for Constraint Satisfaction Problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 572–578. Morgan Kaufmann, 1995. URL: `http://ijcai.org/Proceedings/95-1/Papers/075.pdf`.

16 Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact Combinatorial Optimization with Graph Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems*, pages 15554–15566, 2019. URL: `https://proceedings.neurips.cc/paper/2019/hash/d14c2267d848abeb81fd590f371d39bd-Abstract.html`.

17 Pieter Andreas Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of the 10th European Conference on Artificial Intelligence*, ECAI '92, pages 31–35, USA, 1992. John Wiley & Sons, Inc.

18 Matthew L. Ginsberg, Michael Frank, Michael P. Halpin, and Mark C. Torrance. Search Lessons Learned from Crossword Puzzles. In *Proceedings of the 8th National Conference on Artificial Intelligence. Boston, Massachusetts, USA, July 29 – August 3, 1990, 2 Volumes*, pages 210–215. AAAI Press / The MIT Press, 1990. URL: `http://www.aaai.org/Library/AAAI/1990/aaai90-032.php`.

19 Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 431–437, USA, 1998. American Association for Artificial Intelligence. URL: `http://www.aaai.org/Library/AAAI/1998/aaai98-061.php`.

20 Christelle Guéret and Christian Prins. A new lower bound for the open-shop problem. *Annals of Operations Research*, 92:165–183, 1999. `doi:10.1023/A\%3A1018930613891`.

21 Joey Hwang and David G. Mitchell. 2-Way vs. d-Way Branching for CSP. In *Principles and Practice of Constraint Programming – CP 2005*, pages 343–357, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. `doi:10.1007/11564751_27`.

22 Elias B. Khalil, Pierre Le Bodic, Le Song, George L. Nemhauser, and Bistra Dilkina. Learning to Branch in Mixed Integer Programming. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 724–731. AAAI Press, 2016. `doi:10.1609/AAAI.V30I1.10080`.

23 Rainer Kolisch and Arno Sprecher. PSPLIB – A project scheduling problem library. *European Journal of Operational Research*, 96(1):205–216, January 1997. `doi:10.1016/s0377-2217(96)00170-1`.

24 Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan Catanzaro. Can Q-Learning with Graph Networks Learn a Generalizable Branching Heuristic for a SAT Solver? In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems*, 2020. URL: `https://proceedings.neurips.cc/paper/2020/hash/6d70cb65d15211726dcce4c0e971e21c-Abstract.html`.

25 S. Lawrence. *An Experimental Investigation of Heuristic Scheduling Techniques (Supplement)*. PhD thesis, Carnegie-Mellon University, 1984.

26 Arnaud Letort, Nicolas Beldiceanu, and Mats Carlsson. A Scalable Sweep Algorithm for the cumulative Constraint. In *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 439–454. Springer, 2012. doi:10.1007/978-3-642-33558-7_33.

27 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning Rate Based Branching Heuristic for SAT Solvers. In *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 123–140, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-40970-2_9.

28 J.P. Marques Silva and K.A. Sakallah. GRASP-A new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, ICCAD-96, pages 220–227. IEEE Comput. Soc. Press, 1996. doi:10.1109/iccad.1996.569607.

29 L. Michel, P. Schaus, and P. Van Hentenryck. Minicp: a lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1):133–184, 2021. doi:10.1007/s12532-020-00190-7.

30 Laurent Michel and Pascal Van Hentenryck. Activity-Based Search for Black-Box Constraint Programming Solvers. In *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, pages 228–243, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-29828-8_15.

31 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/378239.379017.

32 Laurent Perron and Frédéric Didier. CP-SAT. URL: https://developers.google.com/optimization/cp/cp_solver/.

33 Charles Prud'homme and Jean-Guillaume Fages. Choco-solver: A java library for constraint programming. *Journal of Open Source Software*, 7(78):4708, 2022. doi:10.21105/joss.04708.

34 Philippe Refalo. *Impact-Based Search Strategies for Constraint Programming*, pages 557–571. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-30201-8_41.

35 Barbara M. Smith and Stuart A. Grant. Sparse constraint graphs and exceptionally hard problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence – Volume 1*, IJCAI'95, pages 646–651, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

36 Barbara M. Smith and Stuart A. Grant. Where the Exceptionally Hard Problems Are. Technical report, University of Leeds, 1995.

37 Robert H. Storer, S. David Wu, and Renzo Vaccari. New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling. *Management Science*, 38(10):1495–1509, 1992. URL: http://www.jstor.org/stable/2632676.

38 Florent Teichteil-Königsbuch, Guillaume Povéda, Guillermo González de Garibay Barba, Tim Luchterhand, and Sylvie Thiébaux. Fast and Robust Resource-Constrained Scheduling with Graph Neural Networks. In *Proc. 33rd International Conference on Automated Planning and Scheduling*, pages 623–633. AAAI Press, 2023. doi:10.1609/ICAPS.V33I1.27244.

39 Hélène Verhaeghe, Quentin Cappart, Gilles Pesant, and Claude-Guy Quimper. Learning Precedences for Scheduling Problems with Graph Neural Networks. In *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:18, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2024.30.

40 Petr Vilím, Roman Barták, and Ondrej Cepek. Extension of $O(n \log n)$ Filtering Algorithms for the Unary Resource Constraint to Optional Activities. *Constraints An Int. J.*, 10(4):403–425, 2005. doi:10.1007/S10601-005-2814-0.

41 Takeshi Yamada and Ryohei Nakano. A Genetic Algorithm Applicable to Large-Scale Job-Shop Problems. In *Parallel Problem Solving from Nature*, 1992. URL: https://api.semanticscholar.org/CorpusID:37171770.

42 Zhijun Zhang and Susan L. Epstein. Learned value-ordering heuristics for constraint satisfaction. In *Proceedings of STAIR-08 Workshop at AAAI-2008*, 2008. URL: https://api.semanticscholar.org/CorpusID:16569060.

43 Éric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993. Project Management and Scheduling. doi:10.1016/0377-2217(93)90182-M.

A Appendix

A.1 Additional Figures

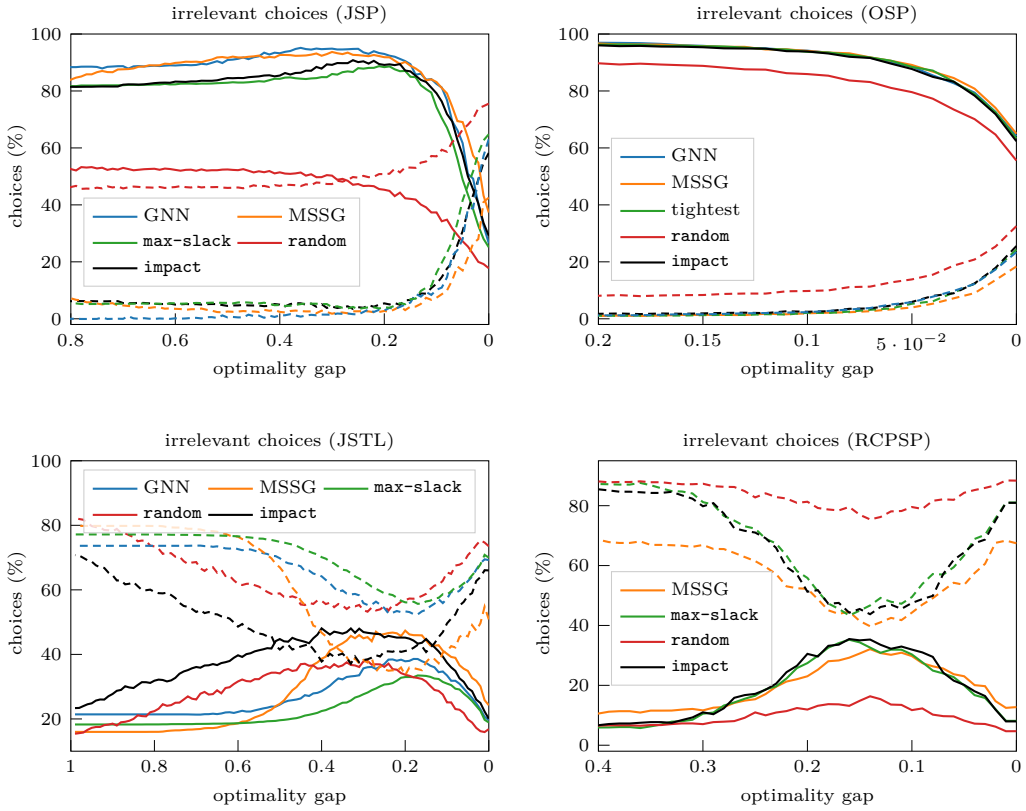


Figure 7 Mean percentage of irrelevant choices distinguished by category with no-good-learning. Bold lines represent the percentage of ICS's, dashed lines represent the percentage of ICUs. These results need to be interpreted with care since the procedure detailed in Section 2.2 is not exact in this case.

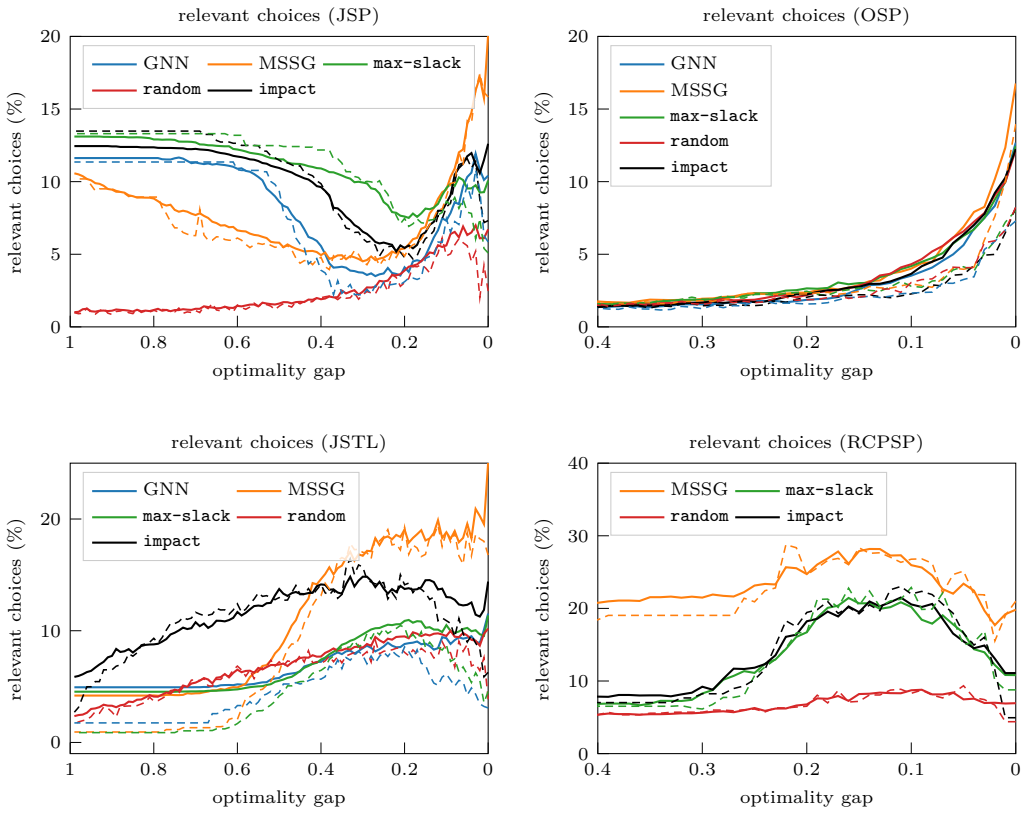


Figure 8 Percentage of relevant choices with no-good-learning. Bold lines represent mean values, dashed lines median values. These results need to be interpreted with care since the procedure detailed in Section 2.2 is not exact in this case.

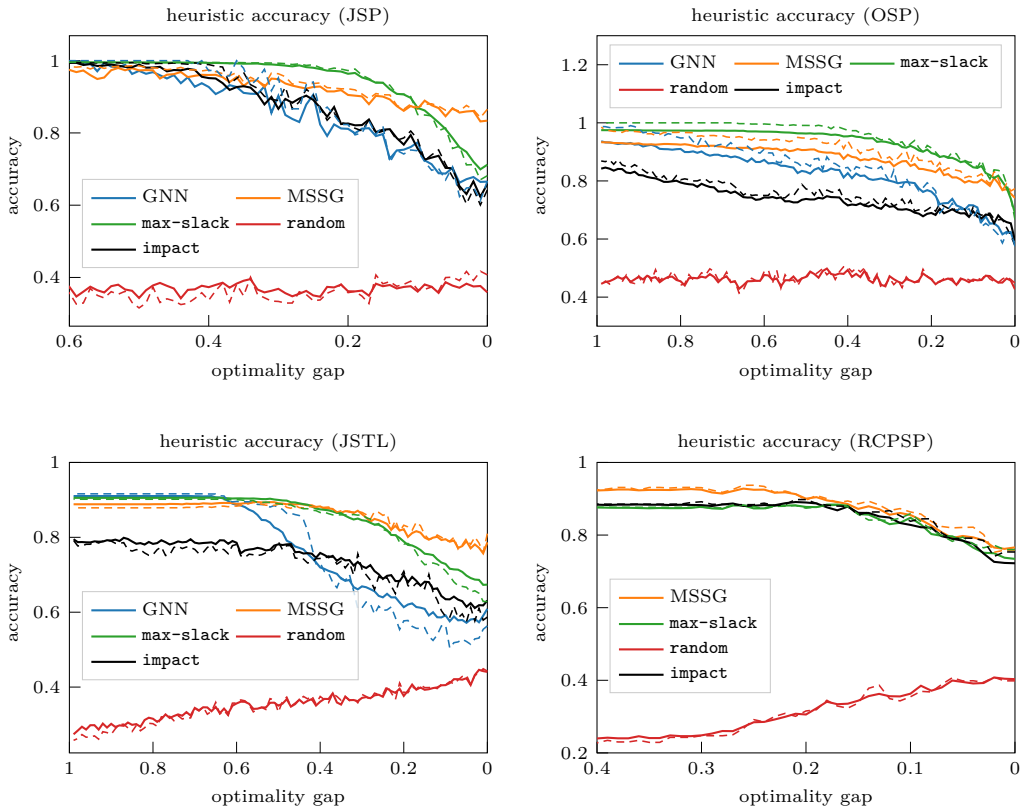


Figure 9 Online accuracy results with no-good-learning. These results need to be interpreted with care since the procedure detailed in Section 2.2 is not exact in this case.

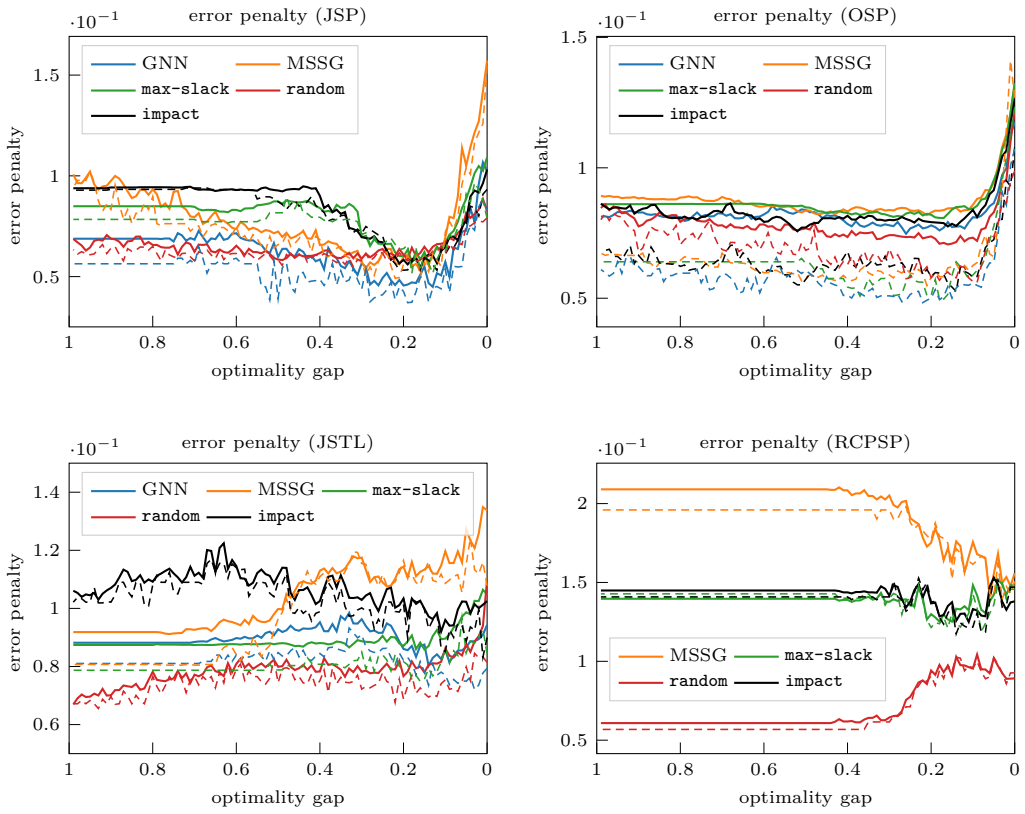


Figure 10 Error penalty results with no-good-learning. Bold lines represent mean values, dashed lines median values. These results need to be interpreted with care since the procedure detailed in Section 2.2 is not exact in this case.

Table 1 Instances used in all experiments.

instance type	size (# tasks)	instance names
JSP	50-500	abz5.txt, abz6.txt, abz8.txt, ft20.txt, la03.txt, la04.txt, la05.txt, la06.txt, la11.txt, la16.txt, la18.txt, la19.txt, la23.txt, la25.txt, orb03.txt, orb04.txt, orb05.txt, orb06.txt, orb07.txt, swv19.txt, swv20.txt, ta02.txt, ta05.txt, yn01.txt
OSP	9-225	GP03-01.txt, GP03-02.txt, GP04-04.txt, GP04-08.txt, GP05-04.txt, GP05-10.txt, GP06-02.txt, GP06-10.txt, GP07-02.txt, GP07-03.txt, GP08-02.txt, GP08-06.txt, GP09-03.txt, GP09-04.txt, GP10-08.txt, GP10-09.txt, j3-per0-2.txt, j3-per10-1.txt, j3-per20-0.txt, j4-per0-1.txt, j4-per20-0.txt, j4-per20-2.txt, j5-per0-1.txt, j5-per20-0.txt, j5-per20-1.txt, j6-per0-0.txt, j6-per10-1.txt, j6-per20-0.txt, j6-per20-1.txt, j7-per0-0.txt, j7-per0-1.txt, j7-per20-0.txt, j8-per0-1.txt, j8-per20-1.txt, j8-per20-2.txt, tai04_04_02.txt, tai04_04_04.txt, tai04_04_06.txt, tai05_05_04.txt, tai05_05_07.txt, tai05_05_08.txt, tai07_07_03.txt, tai07_07_05.txt, tai07_07_06.txt, tai10_10_02.txt, tai10_10_06.txt, tai10_10_07.txt, tai15_15_01.txt
JSTL	36-150	car5_0_0, car5_0_2, car6_0_0,5, car6_0_10, car7_0_10, car7_0_5, car8_0_10, car8_0_inf, ft06_0_1, ft06_0_inf, ft10_0_0, la01_0_0, la01_0_5, la02_0_0, la02_0_3, la03_0_1, la03_0_3, la04_0_0,5, la04_0_1, la05_0_0, 25, la05_0_3, la07_0_0,5, la07_0_10, la08_0_0,5, la08_0_2, la09_0_1, la09_0_10, la10_0_1, la11_0_10, la11_0_inf, la12_0_0, la12_0_1, la13_0_0, la13_0_10, la14_0_0,5, la14_0_1, la15_0_0,5, la15_0_inf, la16_0_0,5, la16_0_inf, la17_0_10, la17_0_3, la18_0_10, la18_0_3, la19_0_1, la19_0_inf, la20_0_1, la20_0_inf, la21_0_0,5, la21_0_10, la22_0_1, la22_0_10, la23_0_0, la23_0_10, la24_0_0,5, la24_0_10, la25_0_0, la25_0_10
RCPSP	30	j309_1.sm, j309_2.sm, j309_3.sm, j309_4.sm, j309_5.sm, j309_6.sm, j309_7.sm, j309_8.sm, j309_9.sm, j309_10.sm, j3013_1.sm, j3013_2.sm, j3013_3.sm, j3013_4.sm, j3013_5.sm, j3013_6.sm, j3013_7.sm, j3013_8.sm, j3013_9.sm, j3013_10.sm

SLS-Enhanced Core-Boosted Linear Search for Anytime Maximum Satisfiability

Ole Lübke[1]
Institute for Software Systems, Hamburg University of Technology, Germany

Jeremias Berg
Department of Computer Science, University of Helsinki, Finland

Abstract

Maximum Satisfiability (MaxSAT), the constraint paradigm of minimizing a linear expression over Boolean (0-1) variables subject to a set of propositional clauses, is today used for solving NP-hard combinatorial optimization problems in various domains. Especially anytime MaxSAT solvers that compute low-cost solutions within a limited available computational time have significantly improved in recent years. Such solvers can be divided into SAT-based methods that use sophisticated reasoning, and stochastic local search (SLS) methods that heuristically explore the search space. The two are complementary; roughly speaking, SLS struggles with finding feasible solutions, and SAT-based methods with minimizing cost. Consequently, most state-of-the-art anytime MaxSAT solvers run SLS before a SAT-based algorithm with minimal communication between the two.

In this paper, we aim to harness the complementary strengths of SAT-based, and SLS approaches in the context of anytime MaxSAT. More precisely, we describe several ways to enhance the performance of the so-called core-boosted linear search algorithm for anytime MaxSAT with SLS techniques. Core-boosted linear search is a three-phase algorithm where each phase uses different types of reasoning. Beyond MaxSAT, core-boosted search has also been successful in the related paradigms of pseudo-boolean optimization and constraint programming. We describe how an SLS approach to MaxSAT can be tightly integrated with all three phases of the algorithm, resulting in non-trivial information exchange in both directions between the SLS algorithm and the reasoning methods. We evaluate our techniques on standard benchmarks from the latest MaxSAT Evaluation and demonstrate that our techniques can noticeably improve on implementations of core-boosted search and SLS.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Theory of computation → Constraint and logic programming

Keywords and phrases Maximum Satisfiability, MaxSAT, SAT, SLS, Anytime Optimization

Digital Object Identifier 10.4230/LIPIcs.CP.2025.28

Supplementary Material *Software (archive)*: `https://doi.org/10.5281/zenodo.15584837`
Software (repository): `https://github.com/jezberg/loandra/tree/boums-cadical-integration` [9], archived at `swh:1:dir:6babae3bfbb34e637ca99fab6b5f3a85f2263476`

Funding The work is supported by Research Council of Finland under grant 362987.

Acknowledgements The authors thank the Finnish Computing Competence Infrastructure (FCCI) for computational and data storage resources.

1 Introduction

Maximum Satisfiability (MaxSAT) has, over the last few years, matured into a thriving optimization paradigm that is today used for solving NP-hard optimization problems across various domains [6]. While especially theoretical work tends to treat MaxSAT as the problem

[1] Corresponding author

© Ole Lübke and Jeremias Berg;
licensed under Creative Commons License CC-BY 4.0

of computing assignments that maximize the (sum-of-weights) of satisfied clauses, most modern MaxSAT solvers treat MaxSAT as the equivalent problem of minimizing a linear cost function over 0-1 variables subject to a set of propositional clauses. Modern MaxSAT solving techniques can roughly be divided into two categories: exact and anytime solvers. Exact solvers aim to compute minimum-cost solutions using as little time and memory as possible. While there has been some recently renewed interest in branch and bound techniques for MaxSAT [33], a clear majority of current exact approaches build on the highly successful conflict-driven clause learning (CDCL) SAT solvers [35] and, essentially, reduce the optimization task into a sequence of decision queries [37, 2].

In this work, we focus on anytime MaxSAT, which aims to compute as low-cost solutions as possible within limited computation time and memory. Existing approaches to anytime MaxSAT can roughly be divided into approaches based on CDCL solvers, called SAT-based approaches [40, 38, 10, 20], and approaches based on stochastic local search (SLS) [30, 31, 32, 18, 17, 28]. SAT-based anytime solvers extensively use CDCL solvers and apply sophisticated reasoning techniques in their search. SLS approaches, in turn, heuristically traverse the search space for solutions as effectively as possible while looking for low-cost solutions. The two approaches are known to exhibit orthogonal performance. Roughly speaking, the reasoning techniques employed by CDCL-based approaches excel at finding feasible solutions or escaping local optima, while SLS approaches excel at quickly finding low-cost solutions in local optima when given a feasible solution from which to start. This orthogonality and the promise of combining reasoning-based CDCL-like and search-based SLS-like methods have previously been realized in related fields. As a notable example related to our work, we note that integrating a local search component in a CDCL solver has been shown to increase performance, especially on satisfiable instances [15].

The orthogonal performance of SAT-based and SLS methods for anytime MaxSAT is well understood by MaxSAT solver developers. Since 2018, practically all anytime solvers participating in the MaxSAT evaluations first apply a SAT solver to obtain any feasible solution, use that solution as a starting point for an SLS algorithm, and finally – if the time limit is not yet reached – attempt to improve the solution returned by the SLS approach through a SAT-based algorithm. In the other direction, some SLS approaches to MaxSAT use the propagation mechanics of a CDCL-type solver to escape local optima [16]. We study further ways to combine the strengths of SAT-based and SLS approaches to anytime MaxSAT. In particular, we work with the so-called core-boosted linear search algorithm to anytime MaxSAT [10] that uses a multitude of CDCL-based reasoning techniques, including preprocessing, lower-bounding core-guided search [41], and upper-bounding solution improving search [42]. Core-boosted linear search has been used in many applications, including interpretable data analysis [24, 44] and scheduling [19]. Similar algorithmic ideas have also successfully been applied in more expressive constraint paradigms like Pseudo-Boolean optimization [21], constraint programming [23] and in more general optimization tasks like multiobjective optimization [26].

More specifically, in this work, we i) identify several novel ways of integrating an SLS approach to all phases of core-boosted search and ii) provide a systematic study on the effect of applying SLS prior to a CDCL-based algorithm in anytime MaxSAT. The main novel integrations of SLS with CDCL-based search we propose are the use of unsatisfiable cores in SLS and the use of SLS interleaved in the so-called increasing precision heuristic (called varying resolution in previous work) [20]. Our experimental evaluation on standard benchmarks demonstrates that sophisticated ways of integrating SLS can result in improved performance of core-boosted search, but the precise effect significantly depends on the specific solver instantiations.

The rest of the paper is structured as follows. In Section 2, we discuss anytime MaxSAT, core-boosted linear search, and SLS for MaxSAT to the extent required for understanding our work. In Section 3, we detail the ways in which SLS can be integrated with all three phases of core-boosted linear search. In Section 4 we report on an experimental evaluation of the effectiveness of the resulting integration of SLS and core-boosted search, before giving concluding remarks in Section 5.

2 Core-Boosted Linear Search and SLS for anytime MaxSAT

In this section, we define Maximum Satisfiability (MaxSAT) and related terms that are used throughout the paper, followed by descriptions of core-boosted linear and stochastic local search (SLS) for MaxSAT.

2.1 Maximum Satisfiability

A literal ℓ is a $\{0,1\}$-valued variable x or its negation $\bar{x}$. A clause $C \equiv \sum_i \ell_i \geq 1$ is an at-least-one constraint and a (CNF) formula F is a set of clauses. We will often treat clauses as the set of literals that appear in them. An objective O is a pseudo-Boolean expression $\sum_i c_i \ell_i$ where each c_i is a positive integer. A (truth) assignment α maps variables to 0 (which we identify with FALSE) or 1 (TRUE). The semantics of assignments are extended to literals, clauses, and formulas in the standard way: $\alpha(\bar{x}) = 1 - \alpha(x)$, $\alpha(C) = \max\{\alpha(\ell) \mid \ell \in C\}$, and $\alpha(F) = \min\{\alpha(C) \mid C \in F\}$. The value (or cost) of O under α is $\alpha(O) = \sum_i c_i \alpha(\ell_i)$. With a slight abuse of notation, we sometimes treat an objective O as a set of its terms, i.e. write $c\ell \in O$ if $O \equiv c\ell + \sum_i c_i \ell_i$. α is a solution of F if $\alpha(F) = 1$. F is satisfiable if it has solutions and unsatisfiable if not.

We adopt two equivalent definitions of Weighted Partial Maximum Satisfiability (WPMS) to present core-boosted linear and stochastic local search (SLS). Specifically, SLS approaches are more naturally described in terms of the so-called *clause-based definition* of MaxSAT in which an instance (F_H, F_S, w) consists of two formulas, the hard clauses F_H, the soft clauses F_S, and a weight function w that associates a positive integer weight $w(C)$ to each soft clause $C \in F_S$. The goal is to compute a solution α of F_H that minimizes the cost $\text{COST}(F_S, w, \alpha) = \sum_{C \in F_S} w(C)(1 - \alpha(C))$, defined as the weights of falsified soft clauses. Core-boosted linear search (and indeed any approach that makes use of a CDCL SAT-solver) for MaxSAT is more natural to describe in terms of the so-called *objective-based* definition of MaxSAT in which an instance (F, O) consists of a formula F and an objective O. The goal is to minimize O subject to F, i.e. compute a solution α of F that minimizes $\alpha(O)$ over all solutions of F. For both definitions of MaxSAT a minimum-cost solution of F is called an optimal solution of F. The *optimum cost* of the instance is the cost of an optimal solution. In *anytime* MaxSAT solving, the goal is to compute a solution of as low cost as possible within some fixed amount of computational resources, typically computation time.

For intuition on the equivalence between the clause-based and the objective-based definitions of MaxSAT, note that a term $c\ell$ of an objective can be viewed as a soft clause $(\bar{\ell})$ with weight c. Thus, any objective-based instance (F, O) can be seen as the clause-based instance $(F, \{(\bar{\ell}) \mid c\ell \in O\}, w)$ where $w((\bar{\ell})) = c$ whenever $c\ell \in O$. In the other direction, a clause-based instance (F_H, F_S, w) can be viewed as an objective-based instance $(F_H \cup \{C \vee b_C \mid C \in F_S\}, \sum_{C \in F_S} w(C) b_C)$ formed by: (i) extending each soft clause C with a fresh variable b_C, (ii) forming a formula that contains the hard clauses and the extended soft clauses, and (iii) forming an objective to be the sum of the weights of soft clauses and their extension variables. For more details on the transformation and the equivalence between the definitions, we refer the reader to [6].

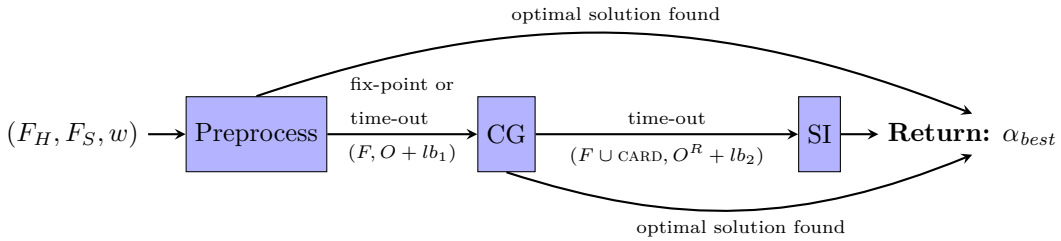


Figure 1 Core-boosted linear search. CG is the core-guided phase, and SI is the solution-improving phase. We assume every step can update the best-known solution α_{best}.

▶ **Example 1.** Consider the clause-based MaxSAT instance (F_H, F_S, w) with: $F_H = \{(x_1 \vee \bar{x}_2), (\bar{x}_1 \vee \bar{x}_2), (x_2 \vee \bar{x}_3)\}$, $F_S = \{(x_1 \vee x_2), (\bar{x}_1 \vee x_2), (x_1 \vee x_2 \vee x_3)\}$, and $w((x_1 \vee x_2)) = 1, w((\bar{x}_1 \vee x_2)) = 2, w((x_1 \vee x_2 \vee x_3)) = 3$. The corresponding objective-based instance (F^o, O) has $F^o = F_H \cup \{(x_1 \vee x_2 \vee b_1), (\bar{x}_1 \vee x_2 \vee b_2), (x_1 \vee x_2 \vee x_3 \vee b_3)\}$ and $O = b_1 + 2b_2 + 3b_3$. An optimal solution α to the instance assigns $\alpha(x_2) = \alpha(x_3) = \alpha(b_1) = \alpha(b_3) = 0$ and $\alpha(x_1) = \alpha(b_2) = 1$ and has $\text{COST}(F_S, w, \alpha) = \alpha(O) = 2$.

2.2 Incremental SAT Solving and Unsatisfiable Cores

Core-boosted linear search makes extensive use of incremental SAT-solving under assumptions and unsatisfiable cores. An *unsatisfiable core* of an objective-based MaxSAT instance (F, O) is a clause C that: (i) only contains variables that appear in the objective, and (ii) is satisfied (i.e. assigned to 1) by any solution to F. Notice that satisfying a core requires assigning at least one objective variable to 1, thus incurring cost in O. In this sense, cores provide information on the lower bound of the optimal cost of the instance.

To see the connection between our clause-based definition of a core and the more classical one in terms of unsatisfiable subsets of constraints, note that requirement (ii) that all solutions to F satisfy C is equivalent to the formula $F \wedge \bigwedge_{\ell \in C} \bar{\ell}$ being unsatisfiable. Thus, our definition of a core could be viewed as the negation of an unsatisfiable subset of F obtained by assigning each literal in C to 0 and simplifying accordingly.

In practice, cores of MaxSAT instances are extracted using incremental SAT solving under assumptions [22, 7]. We abstract the use of an incremental SAT solver (i.e. a decision procedure for propositional logic) into the function `Inc-SAT-solve`$(F, \mathcal{A})$ where F is a CNF-formula and $\mathcal{A}$ is a set of literals, typically called assumptions. The call returns a triplet $(sat?, C, \alpha)$. If *sat?* is true, then α is a solution to F that extends $\mathcal{A}$, i.e. assigns $\alpha(\ell) = 1$ for all $\ell \in \mathcal{A}$. Otherwise (if *sat?* is false), no such solution exists, and C is a clause satisfied by any solution of F containing negations of some of the literals in $\mathcal{A}$. Notice that F has solutions iff `Inc-SAT-solve`$(F, \emptyset)$ returns true. Unsatisfiable cores of MaxSAT instances are then extracted by having $\mathcal{A}$ contain negations of objective variables. Practically all modern SAT solvers offer an assumption interface out of the box [22, 7].

▶ **Example 2.** Consider the objective-based instance (F^o, O) detailed in Example 1. Invoking `Inc-SAT-solve`$(F^o, \{\bar{b}_2\})$ returns true and (for example) the solution α_1 that assigns $\alpha_1(x_1) = \alpha_1(x_2) = \alpha_1(x_3) = \alpha_1(b_2) = 0$ and $\alpha_1(b_1) = \alpha_1(b_3) = 1$. Invoking `Inc-SAT-solve`$(F^o, \{\bar{b}_1, \bar{b}_2\})$ returns false and the core $(b_1 \vee b_2)$.

2.3 Core-Boosted Linear Search for Anytime MaxSAT

Figure 1 overviews core-boosted linear search for computing a low-cost solution to a (clause-based) MaxSAT instance (F_H, F_S, w) [10]. Core-boosted search consists of three separate phases: preprocessing, core-guided search, and solution-improving search. Each phase can – in theory – find and identify an optimal solution to the instance. More importantly, each phase will also modify and simplify the instance, and the following phase is always invoked on the modified instance rather than the original one. Intuitively, this allows core-boosted search never to be much worse than any individual phase, while also allowing subsequent phases to benefit from deductions made in previous ones. We next detail all three phases.

The Preprocessing Phase

The preprocessing phase converts the input clause-based instance (F_H, F_S, w) into an objective-based one using the transformation outlined in Section 2.1. Additionally, the step applies different simplification techniques. These can range from removing tautologies and reusing unit soft clauses as objective terms to more sophisticated preprocessing rules used in SAT and MaxSAT solving, including bounded variable elimination and subsumed label elimination [8, 25]. For the correctness of core-boosted linear search, we assume that the preprocessing techniques are cost-preserving.

▶ **Definition 3.** *Invoke the preprocessing phase on an instance (F_H, F_S, w) and assume it returns the instance $(F, O + lb_1)$ where lb_1 is a constant. The preprocessing phase is cost-preserving if both of the following hold:*

- **(i)** *If α^P is a solution of F then there exists a solution α to F_H for which $\textsc{cost}(F_S, w, \alpha) \leq \alpha^P(O) + lb_1$.*
- **(ii)** *If α is a solution of F_H then there exists a solution α^P to F for which $\alpha^P(O) + lb_1 = \textsc{cost}(F_S, w, \alpha)$.*

Several examples of cost-preserving preprocessing techniques are known. For practically all of them, given a solution α^P to the preprocessed instance $(F, O + lb_1)$, a solution α to the original instance (F_H, F_S, w) for which $\textsc{cost}(F_S, w, \alpha) \leq \alpha^P(O) + lb_1$ can be obtained in linear time with respect to the number of clauses in $F_H \cup F_S$ [8, 25].

The Core-Guided Phase

The next phase of core-boosted linear search applies core-guided search on the preprocessed instance $(F, O + lb_1)$. In the core-boosted setting, core-guided search only runs for a limited time. More precisely, the phase extracts a set $\textsc{cores}$ of cores of $(F, O + lb_1)$ by repeated invocations of an incremental SAT solver in the way described in Section 2.2. When either the time limit for the core-guided phase is reached or no more cores can be found, the core-guided phase forms a new instance $(F \cup \textsc{card}, O^R + lb_2)$ by relaxing all cores in $\textsc{cores}$. For some informal intuition on core-guided search, we have that the difference $lb_2 - lb_1 \geq 0$ is equal to the minimum cost incurred by satisfying the cores in $\textsc{cores}$, and that the objective O^R measures the additional cost on top of satisfying the cores in $\textsc{cores}$ that solutions of $(F \cup \textsc{card})$ incur in O. More precisely, we require that the core relaxation step is solution preserving in the sense of the next definition.

▶ **Definition 4.** *Invoke the core-guided phase on the instance $(F, O + lb_1)$ and let $(F \cup \textsc{card}, O^R + lb_2)$ be the instance returned. The core-guided phase is solution-preserving if the following conditions hold:*

(i) $\alpha^R(O^R) + lb_2 = \alpha^R(O) + lb_1$ *for any solution* α^R *to* $F \cup \textsc{card}$.
(ii) *Any solution* α *to* F *can be extended into a solution* α^R *to* $F \cup \textsc{card}$ *for which* $\alpha^R(O^R) + lb_2 = \alpha(O) + lb_1$.

In practice, all core-guided algorithms for MaxSAT that we are aware of are solution-preserving and can be used for core-boosted linear search. All of them implement the core relaxation step by introducing new variables of form o^k_C to O^R as indicators for at least $k \geq 2$ variables in the core C being assigned to 1 by solutions of $(F \cup \textsc{card})$. Essentially, such variables ensure that a solution to $F \cup \textsc{card}$ can assign one variable from each core to 1 without incurring cost in O^R, but need to assign one of the new variables to 1 (thus incurring cost) for any subsequent ones. The additional clauses in $\textsc{card}$ contain CNF encodings of reified cardinality constraints equivalent to $\sum_{\ell \in C} \ell \geq k \rightarrow o^k_C$. While existing instantiations of core-boosted search for MaxSAT only extract cores over original objective variables, in the general case, due to core relaxation changing the objective function, subsequent cores could also contain the new variables o^k_C.

The Solution-Improving Phase

The solution-improving phase is invoked on $(F \cup \textsc{card}, O^R + lb_2)$, the final working instance of the core-guided phase. The phase looks for low-cost solutions of $(F \cup \textsc{card}, O^R)$ by invoking a SAT solver on the formula $F \cup \textsc{card} \cup \textsc{as-cnf}(O^R < \alpha_{best}(O^R))$ where α_{best} is the current lowest-cost solution, and $\textsc{as-cnf}(O^R \leq \alpha_{best}(O^R))$ is a CNF encoding of the objective-improving constraint $O^R < \alpha_{best}(O^R)$ that is satisfied by any solution α for which $\alpha(O^R) < \alpha_{best}(O^R)$. If the SAT solver returns a solution, it will cost less than α_{best}. Then, the objective-improving constraint is tightened, and the solver is invoked again. If, instead, the SAT solver determines that no such solution exists, the current best-known solution is returned as optimal. The solution-improving phase runs until either an optimal solution of $(F \cup \textsc{card}, O^R)$ is found, or a time-limit is reached, at which point, the solution α of the input clause-based instance (F_H, F_S, w) for which $\textsc{cost}(F_S, w, \alpha) \leq \alpha_{best}(O) + lb_1$ is returned.

An essential intuition of core-boosted linear search is that any solution of $F \cup \textsc{card}$ that has a low-cost wrt $O^R + lb_2$ will correspond to a solution α of (F_H, F_S, w) for which $\textsc{cost}(F_S, w, \alpha)$ is low. This is because the core-guided phase is solution-preserving, and the preprocessing phase is cost-preserving. Next, we detail two additional heuristics designed for any time solution-improving search that are important for our work.

Increasing Precision. Solution-improving search with increasing precision [20] heuristically picks a constant ρ and runs solution-improving search on the objective $O^\rho = \lfloor O^R/\rho \rfloor$ obtained by: (i) dividing each coefficient in O^R by ρ, (ii) rounding down, and (iii) removing all variables with 0 coefficient. For some intuition on the name, notice how O^ρ is a "lower precision" approximation of O^R for which the solution improving constraint $O^\rho < \alpha_{best}(O^\rho)$ can be encoded with fewer clauses. While the division of an objective obviously can change the cost of solutions, in practice it has been observed that solutions that have a low-cost wrt O^ρ also tend to have a low-cost wrt O^R [20, 10]. If a minimum cost solution wrt O^ρ is found, the value of ρ is lowered – thus increasing the precision – and the search restarted. Solution-improving search with increasing precision ends either when a minimum-cost solution of O^ρ is found when $\rho = 1$, or when a time limit is reached.

Phase Saving. Phase saving refers to overriding the default polarity selection of the SAT solver to guide its search toward solutions of low cost. The intuition underlying phase saving is that a SAT solver, in its default setting, aims to find any solution of its internal formula or

prove that one does not exist. In the context of anytime MaxSAT, not all solutions are equal, so phase saving aims to ensure that the first solution that the solver finds is as low-cost as possible. Typical phase-saving schemes employed in anytime MaxSAT [3, 20, 38] involve setting the default polarity of the variables in the objective to not incur cost, and set the polarity of the remaining variables to match the current lowest-cost solution α_{best}.

2.4 Stochastic Local Search for MaxSAT

Algorithm 1 SLS for MaxSAT.

Require: (F_H, F_S, w), α, `max-flips`
Ensure: α_{best} is a solution
1: UPDATE-BEST-KNOWN-SOL
2: $w^{\texttt{sls}} \leftarrow$ INIT-WEIGHTS
3: **while** `flip-count` $\leq$ `max-flips` **do**
4: **if** $\alpha(F_H) = 1$ and COST$(F_S, w, \alpha) <$ `best-cost` **then** UPDATE-BEST-KNOWN-SOL
5: **end if**
6: **if** $\nexists$ decreasing variable **then** $w^{\texttt{sls}} \leftarrow$ UPDATE-WEIGHTS
7: **end if**
8: $\alpha \leftarrow$ SELECT-AND-FLIP-VARIABLE
9: `flip-count` $\leftarrow$ `flip-count` $+ 1$
10: **end while**
11: **return** α_{best}
12:
13: **function** UPDATE-BEST-KNOWN-SOL
14: $\alpha_{best} \leftarrow \alpha$, `best-cost` $\leftarrow$ COST(F_S, w, α), `flip-count` $\leftarrow 0$
15: **end function**

Algorithm 1 details the overall structure of a stochastic local search (SLS) algorithm for computing a low-cost solution to a clause-based MaxSAT instance (F_H, F_S, w). In addition to the instance itself, the algorithm expects a user-specified parameter on how long to search for. In contrast to the algorithms that make use of a SAT solver, an SLS approach is, in general, not able to identify optimal solutions, which makes the limit parameter mandatory. A typical way of limiting the run time of SLS approaches is to enforce a maximum number of flips `max-flips` and terminate the search if no improving solutions have been found within `max-flips` flips. Finally, the algorithm also expects a solution α of F_H as input. In the general setting, an SLS algorithm can be invoked using any assignment of the variables in $F_H \cup F_S$. However, prior work has shown that invoking an SLS approach with a solution of F_H leads to much better performance in terms of the cost of the final solution being found. Since we are working on different ways of integrating SLS with SAT-based approaches to MaxSAT, we can assume that we always have access to some solution of the instance being solved. In our setting, such a solution can be obtained, e.g. by invoking the SAT solver on the hard clauses.

Algorithm 1 begins by initializing its best-known solution α_{best} to equal the input solution α and a flip counter to 0 on Line 1. Afterward, the SLS weights $w^{\texttt{sls}}$ for all clauses in $F_H \cup F_S$ are initialized. Importantly, these are not the same as the weights of the soft clauses in the instance given by w. Instead, $w^{\texttt{sls}}$ will be used to define the score of each variable.

▶ **Definition 5.** *Assume an SLS approach has a current assignment α and SLS weights w^{sls} when solving an instance (F_H, F_S, w). For a variable x let α_x be the assignment obtained by flipping the value of x, i.e. α_x agrees with α on all variables except x. The* score *of x wrt α is then $\textsc{score}(x) = \textsc{cost}(F_H \cup F_S, w^{sls}, \alpha) - \textsc{cost}(F_H \cup F_S, w^{sls}, \alpha_x)$.*

In other words, the score of a variable x is the difference in the sum of weights wrt $w^{\texttt{sls}}$ of falsified clauses (hard or soft) when flipping the variable x. Informally speaking, flipping a variable x with a high positive score will result in the current solution improving. With this intuition, the main loop (Lines 3 to 9) first checks on Line 4 whether the current assignment α' is a solution of lower cost than the best know, and if it is, the best-known solution and `best-cost` are updated. Additionally, the flip counter (`flip-count`) is reset to 0. Note specifically that the flip counter measures the number of flips since the last improvement to the solution was found. Afterward, the existence of so-called decreasing variables is checked. Here a variable is decreasing if its score is positive since flipping variables with a positive score decreases the cost of the assignment wrt $w^{\texttt{sls}}$ (cf. Definition 5). If no such variables exist, the algorithm has encountered a local optimum, and the SLS weights $w^{\texttt{sls}}$ are updated to escape it. Note that practical implementations of SLS algorithms directly modify and store the score of affected variables as well. Finally, a variable is selected, and the current assignment is updated by flipping its value on Line 8, and the flip counter is increased on Line 9. Commonly, variable selection employs the Best from Multiple Selection (BMS) heuristic [14], where a fixed number of variables is drawn randomly with replacement, and the variable with the highest score is selected from those.

Different SLS algorithms differ mainly in how the SLS weights are built and updated, and the specific choice of weighting scheme is known to impact the empirical performance of the approach significantly. In our work, we make use of the so-called NuWLS 2.0 weighting scheme for MaxSAT that was introduced in the NuWLS-c-2023 solver [17] that won all four anytime tracks of the MaxSAT evaluation 2023 and has since been integrated into other, well-performing MaxSAT solvers as well [5, 11, 27, 39]. When solving a MaxSAT instance (F_H, F_S, w) NuWLS 2.0 initializes the weight of each hard clause to 1 and the weight of each soft clause to 0. Informally, this encourages Algorithm 1 to initially flip variables more likely to satisfy the hard clauses. The way the weights are updated on Line 6 depends on if the current assignment α is a solution of F_H. If not, the weights of *all falsified* hard clauses are increased by 1. Otherwise, if $\alpha(F_H) = 1$, the weights of *all* soft clauses are increased by their *tuned weight*.

▶ **Definition 6.** *Given an instance (F_H, F_S, w), the tuned weight of a soft clause $c \in F_S$ is equal to $\frac{w(c)|F_S|}{\sum_{d \in F_S} w(d)}$.*

In other words, the tuned weight of a soft clause c is $w(c)$ divided by the average weight of all soft clauses. Dividing by the average effectively distributes the tuned weights around 1: clauses c with smaller-than-average (respectively larger-than-average) $w(c)$ have a tuned weight $\in (0, 1)$ (respectively > 1).

3 Integrating SLS with Core-boosted Linear Search

In this section, we describe the different ways in which we integrate SLS with all phases of core-boosted linear search. Many of the techniques we describe involve invoking an SLS algorithm (i.e. Algorithm 1) on an objective-based instance (F, O). This should be understood as invoking Algorithm 1 on the clause-based instance obtained by treating each term $c\ell \in O$ as a unit soft clause $(\bar{\ell})$ of weight c. In the following sections, assume that core-boosted linear search is invoked on a clause-based instance (F_H, F_S, w).

3.1 SLS in the Preprocessing Phase

A very natural way of integrating SLS with a core-boosted linear search would be to first run the preprocessing phase on the clause-based input instance to obtain the objective-based instance $(F, O + lb_1)$, then invoke a SAT solver on F to obtain an initial solution α, and finally invoke Algorithm 1 on (F, O) with α as the initial assignment and use the α_{best} returned as an initial assignment for the rest of the core-boosted linear search. This idea is well-studied in the context of anytime MaxSAT and corresponds to the de facto way in which most state-of-the-art anytime MaxSAT solvers integrate SLS with a CDCL-based algorithm [39, 28, 17].

While the effectiveness of applying an SLS algorithm before a CDCL-based algorithm is well understood in anytime MaxSAT (as witnessed e.g. by the results of the latest MaxSAT evaluations), much less is known about the relative effectiveness of running an SLS algorithm on a clause-based MaxSAT instance compared to the objective-based MaxSAT instances obtained after preprocessing. In this work, we study the effect of running Algorithm 1 both before and after the preprocessing phase of the core-boosted search. More precisely, given $(F, O + lb_1)$, we first call `Inc-SAT-solve`$(F, \emptyset)$ in order to obtain a solution α of F which we use as a starting point to invoke Algorithm 1 on the instance (F, O), and obtain a solution α' (potentially equal to α). Then, we invoke Algorithm 1 on the instance (F_H, F_S, w) using the solution α^R of F_H that corresponds to the solution α' of F as the initial assignment.

Notice that, especially when employing more advanced preprocessing rules like bounded variable elimination, the clauses in F might significantly differ from those in $F_H \cup F_S$ and the relative effectiveness of Algorithm 1 in terms of the cost of the solution returned is an open question. Another important intuition here is that the solution β returned by Algorithm 1 when invoked on (F_H, F_S, w) need not be a solution of F, thus while the cost $\text{COST}(F_S, w, \beta)$ is an upper bound on the optimal cost of $(F, O + lb_1)$, β can not be used for phase saving (cf. Section 2.3). In our implementation, we store β throughout the rest of the core-boosted search, and in the end, compare the best solution found by core-boosted search with β and return the better of the two.

▶ **Example 7.** Consider the clause-based instance (F_H, F_S, w) from Example 1. Assume the preprocessing phase (as a consequence of e.g. applying bounded variable elimination) returns the objective-based instance (F, O) with $F = \{(x_1 \vee b_1), (\bar{x_1} \vee b_2)\}$, and $O = 4b_1 + 2b_2$. Invoking `Inc-SAT-solve`$(F, \emptyset)$ could obtain the solution α that assigns $\alpha(x_1) = \alpha(b_1) = \alpha(b_2) = 1$ with $\alpha(O) = 6$ that can be used as the starting point for SLS (Algorithm 1) invoked on (F, O). α corresponds to the solution β of F_H that assigns $\beta(x_1) = 1, \beta(x_2) = \beta(x_3) = 0$ that has $\text{COST}(F_S, w, \beta) = 2$ and can be used as a starting point for running Algorithm 1 on (F_H, F_S, w).

3.2 SLS in the Core-Guided Phase

During the core-guided phase of a core-boosted linear search, most of the SAT solver invocations are expected to return false and a new core. Core extraction essentially boils down to learning new clauses, and since SLS approaches can not learn new clauses nor determine that an instance is unsatisfiable, their applicability during the core-guided phase is limited. For general core-guided algorithms, SLS could conceivably help with heuristics that rely on finding intermediate solutions, most notably the so-called core exhaustion (also called cover optimization) technique [1, 36] that seeks to greedily strengthen the extracted cores with additional SAT calls under relaxed sets of assumptions. Since the PMRES [41] core-guided algorithm that the particular instantiation of core-boosted search we work with

uses can not be extended with core exhaustion, we instead study ways in which the cores learned during this phase can be used to improve the performance of the SLS approach in the solution-improving phase where the goal is to compute new solutions, and SLS is expected to perform better. Additionally, we use the SLS approach at the end of the core-guided phase to find a low-cost solution to the instance obtained after the core-guided algorithm performs the core relaxation steps.

Using Cores in SLS

The core-guided phase is invoked on the objective-based instance $(F, O + lb_1)$ obtained after the preprocessing phase. It iteratively extracts a set of cores CORES of (F, O) in the form of clauses over the variables in O that are satisfied by all solutions in F. An essential intuition for how we use cores in an SLS algorithm is that the subsequent solution-improving phase of a core-boosted algorithm is invoked on the instance $(F \cup \text{CARD}, O^R)$ that also contains all the clauses in F. Thus, all the solutions of interest during the solution-improving phase will also have to satisfy all the cores extracted during the core-guided phase. Furthermore, such cores are obtained by the sophisticated reasoning performed by the SAT solver and can, as such, represent information that the comparatively much simpler SLS algorithm does not have access to, motivating the inclusion of cores in the SLS algorithm.

A very natural idea of how to incorporate cores into Algorithm 1 during the solution-improving phase is to add all cores as (hard) clauses to the instance, i.e. to run Algorithm 1 on the instance $(F \cup \text{CARD} \cup \text{CORES}, O^R)$ rather than $(F \cup \text{CARD}, O^R)$.

▶ **Example 8.** Assume the core-guided phase was executed on the objective-based instance (F, O) from Example 7, and found the core $C = (b_1 \vee b_2)$. As C is satisfied by all solutions of F, it can be added to its clauses without changing the set of solutions. It can, however steer the search of the SLS algorithm more effectively toward feasible solutions by effectively communicating that both of the soft clauses $(\bar{b_1})$ and $(\bar{b_2})$ in the clause-based representation of (F, O) that SLS in invoked on, cannot be satisfied simultaneously by any solution.

While the direct addition of the cores is conceptually simple and an easy-to-implement way of communicating information from the cores to the SLS algorithm, adding a potentially large number of new clauses into the instance risks decreasing the overall effectiveness of Algorithm 1 and slowing down the rate at which flips can be performed. Thus, we also consider a way to integrate information in the cores directly in the weighting scheme of Algorithm 1. Recall that the score of a variable (cf. Definition 5) essentially indicates to the algorithm how important it is to flip that variable to steer toward low-cost solutions, and that the score of a variable represents a cost reduction wrt the solver-internal clause weights $w^{\texttt{sls}}$ (cf. Definition 5). The higher the score, the likelier the variable is to get flipped.

Informally speaking, our "core-aware" extension of the NuWLS weighting scheme described in Section 2.4 increases the scores of variables that appear in cores that are assigned to 0 by the current assignment of the SLS algorithm to guide the algorithm toward feasible solutions and decreases the scores of variables appearing in cores that are already assigned to 1 to discourage the search from setting more than one variable to 1 from each core, as doing so would incur more cost in the objective. Additionally, we note that some of the variables in the cores might have had their coefficients set to 0 in O^R. Conceptually, all the cost of assigning such a variable to 1 is already accounted for in the lower bound derived by the core-guided phase. As such, assigning them to 1 during the solution-improving phase incurs no additional cost. With this intuition, we lower the scores of such variables by a larger amount to encourage the SLS algorithm to satisfy the cores on these variables specifically.

Algorithm 2 Modification of SLS scores to incorporate cores.

Require: $(F \cup \text{CARD}, O^R)$, CORES, `scores`, `core-factor` α

for $C \in$ CORES **do**
  $n \leftarrow$ number of literals in C assigned to 1 by α
  for $\ell \in C$ **do**
    $w \leftarrow$ tuned weight of ℓ in O^R
    if $w = 0$ **then** *change* $\leftarrow$ `core-factor` $\cdot$ maximum tuned weight in O^R
    else *change* $\leftarrow \frac{1}{w}$
    end if
    if $n = 0$ **then** add *change* to `scores`$[\ell]$
    else subtract $n \cdot$ *change* from `scores`$[\ell]$
    end if
  end for
end for

Algorithm 2 details the core-based extension of variable scores that we invoke after the weight initialization phase of SLS (cf. Algorithm 1, Line 2). The procedure alters the score of each literal ℓ that appears in a core found during the core-guided phase based on the tuned weight (cf. Definition 6) of ℓ in O^R where O^R is the objective resulting after the core-relaxation steps performed by the core-guided phase. The change in score is based either on the coefficient of ℓ in O^R, or – if the coefficient of ℓ is 0 – by the maximum tuned weight of any literal in O^R.

During the SLS procedure, whenever a variable is flipped, we determine the affected cores (the implementation maintains a mapping from variables to cores), update the count of satisfied literals, and then modify the variable scores of each affected core accordingly, i.e. the variable scores of falsified cores are increased by the inverse corresponding weight, and those of satisfied cores are decreased by n times the inverse corresponding weight. Note, that initially, $n > 0$ for all cores, since we always start SLS from an initial assignment that is a solution, so it satisfies all cores. After a variable flip we do not run Algorithm 2 again, but update the scores so that the resulting values are equal to the values we would obtain by executing Algorithm 2 if the current assignment were the initial assignment. The idea of using information from cores in the SLS-phase bears some resemblance to encoding the constraints of an optimization problem into the optimization criteria via so-called penalty functions [45]. However, as all cores are logically entailed by the constraints in the instance, including any of them in the weighting function does not relax the instance or change the set of feasible solutions to it.

SLS for Obtaining a Solution to the Relaxed Instance

At the end of the core-guided phase, the cores of $(F, O + lb_1)$ are relaxed and the instance $(F \cup \text{CARD}, O^R + lb_2)$ on which the solution improving phase will be invoked is formed. At this point, the SAT solver used by the core-boosted search only contains the clauses in F. As a consequence, the best-known solution α_{best} will only assign the variables in F and O. To obtain an upper bound on the optimal cost of $(F \cup \text{CARD}, O^R + lb_2)$ and a solution for the solution improving phase to use phase saving with, α_{best} needs to be extended into a solution of $(F \cup \text{CARD}, O^R + lb_2)$.

Before our work, the de facto way in which implementations of core-boosted search algorithms for MaxSAT extended α_{best} into a solution of $F \cup \text{CARD}$ was to invoke a SAT solver on the clauses $F \cup \text{CARD}$ with assumptions that correspond to α_{best}, i.e. invoke

`Inc-SAT-solve`$(F \cup \text{CARD}, \{\ell \mid \alpha_{best}(\ell) = 1\})$ and use the obtained solution α as a starting point for the solution-improving phase. In our preliminary experiments, we observed that the solution α obtained this way often is quite poor, i.e. has a high $\alpha(O^R)$. An intuitive explanation for this is that most core-guided MaxSAT algorithms encode the semantics of the new variables in O^R as implications rather than equivalences. As a consequence, there are several extensions of α_{best} into solutions of $F \cup \text{CARD}$ that all have a cost wrt O^R that is at least $\alpha_{best}(O)$ and often much higher. A SAT solver does not differentiate between these extensions and can return any of them. Our solution is to invoke an SLS algorithm on $(F \cup \text{CARD}, O^R)$ using α as a starting point. This allows the SLS algorithm to search not only over the extensions of α_{best}, but also improve on α_{best} itself.

▶ **Example 9.** Consider the instance (F, O) from Example 7 and assume the best assignment α_{best} found so far assigns $\alpha_{best}(x_1) = \alpha_{best}(b_2) = 0$ and $\alpha_{best}(b_1) = 1$. If the core-guided phase finds the core $(b_1 \vee b_2)$ it will form the new objective $O^R = 2b_1 + 2o_C^1$ and new clauses $\text{CARD} = \text{AS-CNF}((b_1 + b_2 > 1) \rightarrow o_C^1)$. Invoking a SAT solver on $F \cup \text{CARD}$ with the assumptions $\{\bar{x_1}, \bar{b_2}, b_1\}$ can obtain the solution α that assigns $\alpha(o_C^1) = 1$, even though a solution that assigns $o_C^1 = 0$ is also a valid extension.

3.3 SLS During the Solution-Improving Phase

In contrast to the core-guided phase, the solution-improving phase of core-boosted linear search focuses on finding solutions of clauses in $F \cup \text{CARD}$ under different precisions (cf. Section 2.3) of O^R. An important intuition for integrating SLS with the phase is that the number of clauses required for encoding an objective improving constraint $\lfloor O^R/\rho \rfloor < B$ of the objective under a precision ρ depends on the bound B. Thus, a low-cost (wrt $\lfloor O^R/\rho \rfloor$) solution can be used to decrease the number of clauses that need to be added to the SAT solver in the solution-improving phase and thus make the SAT calls faster to solve. To obtain such a solution, we invoke SLS on the instance $(F \cup \text{CARD}, \lfloor O^R/\rho \rfloor)$ once for every precision considered during the phase. If the SLS algorithm finds a solution of cost 0, the SAT solver is never invoked, instead the ρ is lowered and the SLS algorithm invoked again.

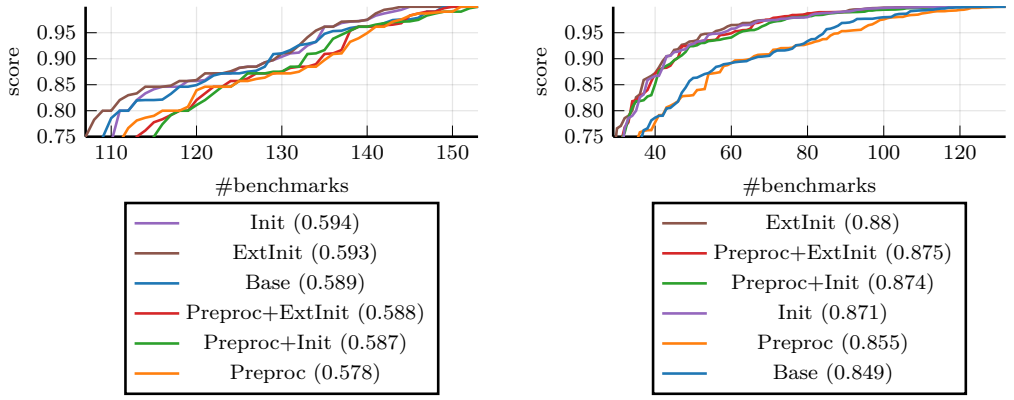
Note, that these more frequent SLS calls can be performed without reinitializing the SLS solver. After the core-guided phase, we initialize the solver once with the instance $(F \cup \text{CARD}, O^R)$. We directly modify the weights inside the SLS solver to apply the increasing precisions.

▶ **Example 10.** Consider the instance (F, O) from Example 7. Assume that the first precision considered in the solution-improving phase is 4. Then $\lfloor O/4 \rfloor = b_1$, initially, only one objective variable is being considered. Executing SLS on $(F, \lfloor O/4 \rfloor)$, could yield an assignment that sets $x_1 = \text{TRUE}$, which is a 0-cost solution for $\lfloor O/4 \rfloor$, and thus trivially an optimal solution for $\lfloor O/4 \rfloor$.

4 Experimental Evaluation

We implemented the techniques outlined in Section 3 in the anytime core-boosted linear search solver Loandra from the MaxSAT Evaluation (MSE) 2024 [10, 11]. This version of Loandra uses the SAT solver Glucose 4.1 [4] and runs the local search component of the NuWLS-c solver [17] on the full objective-based instance at the start of each new precision level during solution improving search.

The baseline solver in our evaluation, **Base**, is the 2024 MSE version of Loandra with NuWLS-c exchanged for BouMS 2.1.0 [34] and Glucose for CaDiCaL 2.0 [13]. Since the current experimental evaluation focuses on the algorithmic ideas rather than specific parameter



(a) unweighted. **(b)** weighted.

Figure 2 Anytime scores for different variants of running SLS before core-boosted linear search.

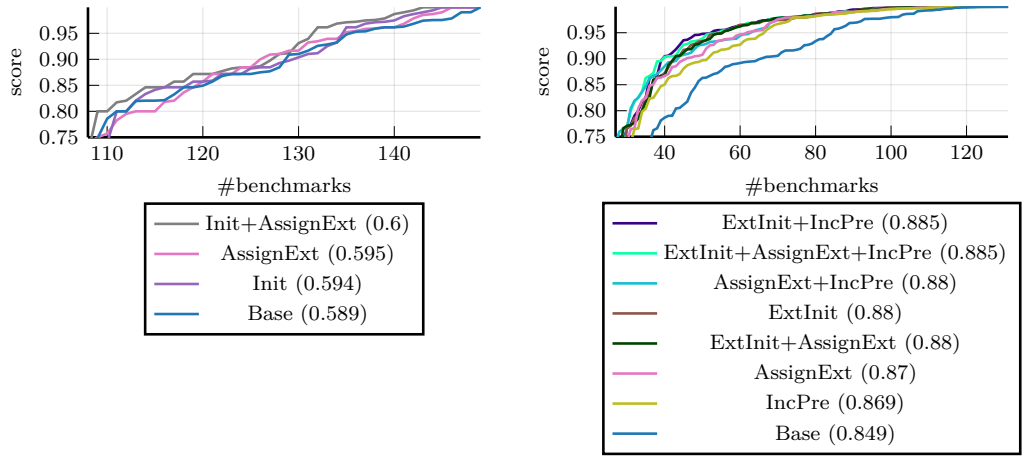
values, we set the parameters of **Base** to equal those used in previous work [17, 11, 42]; the `max-flips` of BouMS was set to 10^7, the preprocessing phase and the core-guided phase are both invoked for a maximum of 30 seconds, the precision for the solution-improving phase was set to 10, and the solution-improving constraints realized with the dynamic polynomial watchdog encoding. The solver implementing all of our techniques is available in open source at: `https://doi.org/10.5281/zenodo.15584837`.

In our experiments, we use the 216 unweighted and 229 weighted instances from the anytime track of the MSE 2024 (`https://maxsat-evaluations.github.io/2024/benchmarks.html`) [12]. We employ the same comparison metric as the MSE, the average (over all instances) of the anytime score that for a solver s on an instance i is calculated as $\frac{1+ \text{ score of best known cost for } i}{1+ \text{ cost of solution for } i \text{ found by } s}$. The score is 1 if the solver finds a solution whose cost matches the best-known, and 0 if no solution is found. All evaluations were performed on 2.40-GHz Intel Xeon Gold 6148 machines in RHEL under a per-instance 32-GiB memory limit and 15-minute time limit. The choice of a larger (compared to the 5 minutes used in the MaxSAT Evaluations) time limit is motivated by previous work [43] demonstrating that involved search techniques in anytime MaxSAT tend to require a slightly larger time limit to be beneficial.

We first assess the impact of running SLS before the core-guided phase (Section 4.1), followed by an evaluation of running SLS for assignment extension and on different precision levels (Section 4.2). Then we examine the impact of using cores with the previous techniques (Section 4.3). Finally, we compare the resulting solver to the leading state-of-the-art solvers from the MSE 2024 (Section 4.4).

4.1 Impact of Running SLS on the Clause Based Instance

Our first experiment investigates different ways of invoking an SLS solver at the start of the search. For this, we consider two variants (described in more detail in Section 3.1) of the **Base** solver. The `Init` solver invokes SLS on the objective-based instance obtained after the initial conversion. This corresponds to the most common way in which modern SAT-based anytime solvers already use SLS in their search. The `ExtInit` solver extends the `Init` configuration by additionally invoking SLS on the original clause-based instance. Remember



(a) unweighted. **(b)** weighted.

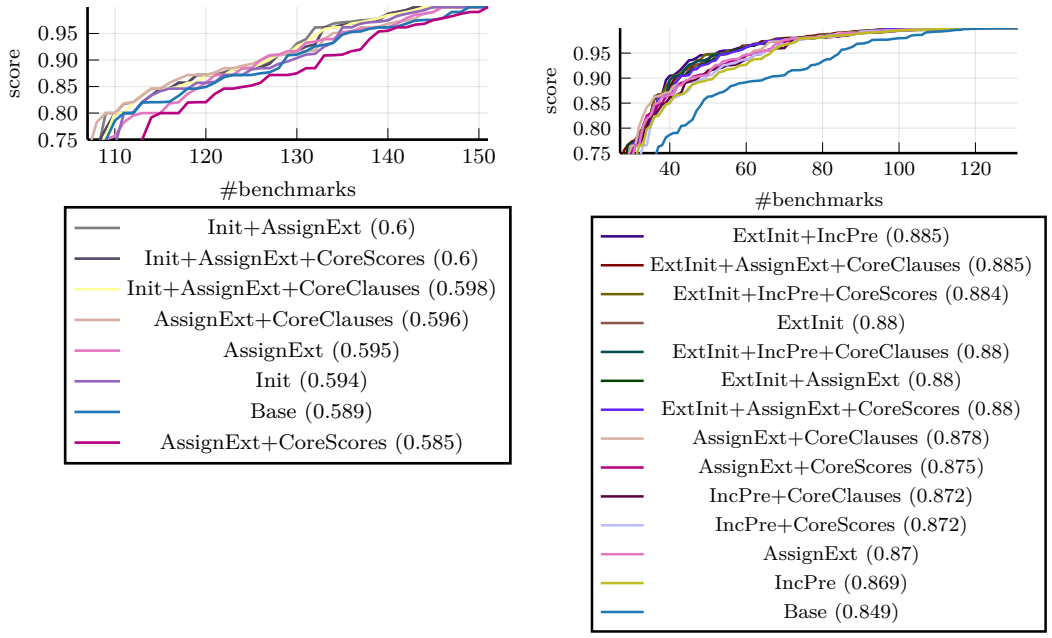
Figure 3 Anytime scores comparing variants of using SLS-enhanced assignment extension and increasing precision.

that, when preprocessing (which we denote by `Preproc`) is applied in the conversion from clause-based to objective-based, there can be a significant difference between the clause-based and the objective-based instance.

Figure 2 shows the results of the experiments. On unweighted instances, `Preproc` decreases the performance of **Base**, but `Init` improves the average score by 0.005. `ExtInit`, however, is not as effective as `Init` only. On the weighted instances, `ExtInit` achieves a score 0.009 higher than `Init`, and 0.031 higher than **Base**. In contrast to the unweighted instances, `Preproc` improves the performance of **Base** and `Init`, yet still degrades that of `ExtInit` by 0.005. We conclude that, on weighted instances, invoking an SLS search prior to the search of a SAT-based algorithm can perform noticeably better on the clause-based instance than on the objective-based one to the extent of being more effective than running preprocessing.

4.2 Impact of SLS-Enhanced Assignment Extension and Increasing Precision

For unweighted instances, the linear search phase only has one precision level. Therefore, using SLS to extend a solution to the relaxed instance after the core-guided phase (`AssignExt` as described in Section 3.2) is practically equivalent to running SLS at the start of each precision level (`IncPre` as described in Section 3.3). Hence, we do not evaluate the latter on unweighted instances. Figure 3 shows the results of applying the two techniques in isolation, in combination with each other, and in combination with the best initial SLS routine for each of the two benchmark sets as determined in Section 4.1. `AssignExt` improves the score by 0.006 for unweighted instances and profits from being paired with `Init`; the combination of the techniques yields an improvement of 0.011. `AssignExt`, in isolation, also improves the score on the weighted instances (by 0.021), and performs similarly to `IncPre`. However, when paired with `ExtInit`, `IncPre` outperforms every other possible combination, and yields an improvement of 0.036 compared to **Base**.



(a) unweighted. **(b)** weighted.

Figure 4 Anytime scores comparing the two variants of using cores.

In conclusion, both SLS integration techniques evaluated in this subsection improve the performance of **Base**, and yield the highest scores when paired with running SLS during the initial phase. In such a pairing, on weighted instances, however, `IncPre` outperforms `AssignExt`.

4.3 Impact of Using Cores in SLS

Figure 4 shows the effect of adding cores as clauses (`CoreClauses`) and using them to modify the SLS weighting scheme (`CoreScores`), as described in Section 3.2. For `CoreScores`, the `core-factor` parameter is set to 1. Both `AssignExt` and `IncPre` benefit from `CoreClauses`, improving the score of `AssignExt` by 0.001 on unweighted instances and by 0.008 on weighted instances. The score of `IncPre` is improved by 0.003. The `CoreScores` variant is not effective on unweighted instances, but on weighted instances its performance is comparable to `CoreClauses` when combined with `IncPre`, and slightly worse in conjunction with `AssignExt`.

The benefits of these (simplistic) approaches of using cores are overshadowed by invoking SLS at the beginning of the search; `Init+AssignExt` remains the best variant for unweighted instances and does not benefit from cores. On weighted instances, `ExtInit+ IncPre` wins, and adding a core strategy reduces the score. Yet, `CoreClauses` can help `ExtInit+ AssignExt` to achieve performance equivalent to `ExtInit+ IncPre`.

In conclusion, while SLS can benefit from the additional information generated in the core-guided phase, employing SLS before the search is more effective. Adding the cores as clauses yields higher scores than the weighting scheme modification introduced in this paper. Yet, both perform almost equivalently on weighted instances, making us carefully optimistic that more sophisticated weighting schemes based on cores could be developed.

4.4 Comparison to State-of-the-Art Solvers

Finally, we note that we did also compare our **Base** solver to the best-performing anytime solvers of the 2024 MSE (SPB_MaxSAT_Band and SPB_MaxSAT_FPS [28]) as well as Loandra as it participated in the 2024 MSE. Detailed results are presented in Appendix A, but the takeaway message is that the best-performing variant of our **Base** solver (`Init+AssignExt` for the unweighted instances and `ExtInit+IncPre` for the weighted) were still outperformed by the 2024 MSE version of Loandra by an average score of 0.018 on unweighted and 0.010 on weighted instances. To look into this, we ran the same experiments with a version of **Base** that uses Glucose as a backend. Without preprocessing, the conclusions with glucose as the backend were very similar to those presented in previous sections. When preprocessing with MaxPRE [29] was turned on, the performance of the solver improved significantly to the extent that any benefits from the SLS integration techniques disappeared. In short, we conclude that variants of Loandra that use Glucose as the backend benefit much more from preprocessing before search than from tighter SLS integration. In contrast, preprocessing has a significantly smaller impact when CaDiCaL is the backend, allowing us to observe benefits from SLS integration regardless. Currently, the best performance of Loandra is achieved with Glucose as the backend and preprocessing.

All in all, our results show that more sophisticated methods of integrating SLS with core-boosted linear search can result in increased performance, but the devil is in the details, and developing universally effective methods is a significant research challenge. We also note that the techniques we describe can be expected to work with any SLS approach for MaxSAT and that the approach implemented in BouMS is not state-of-the-art.

5 Conclusions

We studied the potential synergy between stochastic local search and CDCL-based reasoning for computing low-cost solutions to MaxSAT instances. Focusing on the so-called core-boosted linear search algorithm that makes use of many different CDCL-based optimization algorithms and cost-preserving reasoning, we identified several ways an SLS algorithm can enrich the reasoning performed by the SAT-based search. In addition, we studied ways a SAT-based algorithm can provide non-trivial information about the instance being solved in the form of unsatisfiable cores. Our experimental evaluation of the ideas demonstrated some support for the potential benefits of sophisticated methods of integrating SLS with CDCL-based reasoning in the context of MaxSAT. Further interesting work includes studying more sophisticated ways in which cores can be utilized in SLS.

References

1 Carlos Ansótegui, Maria Luisa Bonet, Joel Gabas, and Jordi Levy. Improving WPM2 for (Weighted) Partial MaxSAT. In *Principles and Practice of Constraint Programming*, volume 8124 of *LNCS*, pages 117–132. Springer, 2013. doi:10.1007/978-3-642-40627-0_12.

2 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. SAT-based MaxSAT algorithms. *Artificial Intelligence*, 196:77–105, 2013. doi:10.1016/j.artint.2013.01.002.

3 Carlos Ansótegui and Joel Gabàs. WPM3: An (in)complete algorithm for weighted partial MaxSAT. *Artificial Intelligence*, 250:37–57, 2017. doi:10.1016/j.artint.2017.05.003.

4 Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 399–404, 2009. URL: https://www.ijcai.org/Proceedings/09/Papers/074.pdf.

5 Florent Avellaneda. EvalMaxSAT 2024. In *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*, page 8. University of Helsinki, Department of Computer Science, 2024.

6 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum Satisfiability. In *Handbook of Satisfiability*, volume 336 of *FAIA*, pages 929–991. IOS Press, 2021. doi:10.3233/FAIA201008.

7 Tomáš Balyo, Armin Biere, Markus Iser, and Carsten Sinz. SAT Race 2015. *Artificial Intelligence*, 241:45–65, 2016. doi:10.1016/j.artint.2016.08.007.

8 Anton Belov, António Morgado, and Joao Marques-Silva. SAT-Based Preprocessing for MaxSAT. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 8312 of *LNCS*, pages 96–111. Springer, 2013. doi:10.1007/978-3-642-45221-5_7.

9 Jeremias Berg. The Loandra MaxSAT solver. Software, swhId: swh:1:dir:6babae3bfbb34e637ca99fab6b5f3a85f2263476 (visited on 2025-07-23). URL: https://github.com/jezberg/loandra/tree/boums-cadical-integration, doi:10.4230/artifacts.24096.

10 Jeremias Berg, Emir Demirović, and Peter J. Stuckey. Core-Boosted Linear Search for Incomplete MaxSAT. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, volume 11494 of *LNCS*, pages 39–56. Springer, 2019. doi:10.1007/978-3-030-19212-9_3.

11 Jeremias Berg, Christoph Jabs, Hannes Ihalainen, and Matti Järvisalo. Loandra in the 2024 MaxSAT Evaluation. In *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*, pages 9–10. University of Helsinki, Department of Computer Science, 2024.

12 Jeremias Berg, Matti Järvisalo, Ruben Martins, Andreas Niskanen, and Tobias Paxian. *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*. University of Helsinki, Department of Computer Science, 2024. URL: http://hdl.handle.net/10138/584878.

13 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In *Computer Aided Verification 36th International Conference*, volume 14681 of *LNCS*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

14 Shaowei Cai. Balance between Complexity and Quality: Local Search for Minimum Vertex Cover in Massive Graphs. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence*, pages 747–753, 2015. URL: https://www.ijcai.org/Proceedings/15/Papers/111.pdf.

15 Shaowei Cai, Xindi Zhang, Mathias Fleury, and Armin Biere. Better Decision Heuristics in CDCL through Local Search and Target Phases. *Journal of Artificial Intelligence Research*, 74:1515–1563, 2022. doi:10.1613/jair.1.13666.

16 Xiamin Chen, Zhendong Lei, and Pinyan Lu. Deep Cooperation of Local Search and Unit Propagation Techniques. In *30th International Conference on Principles and Practice of Constraint Programming*, volume 307 of *LIPIcs*, pages 6:1–6:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.CP.2024.6.

17 Yi Chu, Shaowei Cai, and Chuan Luo. NuWLS-c-2023: Solver Description. In *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*, pages 23–24. University of Helsinki, Department of Computer Science, 2023.

18 Yi Chu, Shaowei Cai, and Chuan Luo. NuWLS: Improving Local Search for (Weighted) Partial MaxSAT by New Weighting Techniques. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 3915–3923, 2023. doi:10.1609/aaai.v37i4.25505.

19 Arnaud De Coster, Nysret Musliu, Andrea Schaerf, Johannes Schoisswohl, and Kate Smith-Miles. Algorithm Selection and Instance Space Analysis for Curriculum-Based Course Timetabling. *Journal of Scheduling*, 25(1):35–58, 2022. doi:10.1007/s10951-021-00701-x.

20 Emir Demirović and Peter J. Stuckey. Techniques Inspired by Local Search for Incomplete MaxSAT and the Linear Algorithm: Varying Resolution and Solution-Guided Search. In *Principles and Practice of Constraint Programming*, volume 11802 of *LNCS*, pages 177–194. Springer, 2019. doi:10.1007/978-3-030-30048-7_11.

21 Jo Devriendt, Stephan Gocht, Emir Demirović, Jakob Nordström, and Peter J. Stuckey. Cutting to the Core of Pseudo-Boolean Optimization: Combining Core-Guided Search with

Cutting Planes Reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3750–3758, 2021. doi:10.1609/aaai.v35i5.16492.

22 Niklas Eén and Niklas Sörensson. Temporal Induction by Incremental SAT Solving. In *BMC'2003: First International Workshop on Bounded Model Checking*, volume 89 of *ENTCS*, pages 543–560. Elsevier, 2005. doi:10.1016/S1571-0661(05)82542-3.

23 Graeme Gange, Jeremias Berg, Emir Demirović, and Peter J. Stuckey. Core-Guided and Core-Boosted Search for CP. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, volume 12296 of *LNCS*, pages 205–221. Springer, 2020. doi:10.1007/978-3-030-58942-4_14.

24 Hao Hu, Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. Learning Optimal Decision Trees with MaxSAT and its Integration in AdaBoost. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence*, pages 1170–1176, 2020. doi:10.24963/ijcai.2020/163.

25 Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Clause Redundancy and Preprocessing in Maximum Satisfiability. In *Automated Reasoning*, volume 13385 of *LNCS*, pages 75–94. Springer, 2022. doi:10.1007/978-3-031-10769-6_6.

26 Christoph Jabs, Jeremias Berg, and Matti Järvisalo. Core Boosting in SAT-Based Multi-Objective Optimization. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, volume 14743 of *LNCS*. Springer, 2024. doi:10.1007/978-3-031-60599-4_1.

27 Menghua Jiang and Yin Chen. NuWLS-c-IBR: Solver Description. In *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*, page 22. University of Helsinki, Department of Computer Science, 2024.

28 Mingming Jin, Kun He, Jiongzhi Zheng, Jinghui Xue, and Zhuo Chen. Combining BandMaxSAT and FPS with SPB-MaxSAT-c. In *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*, pages 23–24. University of Helsinki, Department of Computer Science, 2024.

29 Tuukka Korhonen, Jeremias Berg, Paul Saikko, and Matti Järvisalo. MaxPre: An Extended MaxSAT Preprocessor. In *Theory and Practice of Satisfiability Testing – SAT 2017*, volume 10491 of *LNCS*, pages 449–456. Springer, 2017. doi:10.1007/978-3-319-66263-3_28.

30 Zhendong Lei and Shaowei Cai. SATLike-c: Solver Description. In *MaxSAT Evaluation 2018: Solver and Benchmark Descriptions*, page 24. University of Helsinki, Department of Computer Science, 2018.

31 Zhendong Lei and Shaowei Cai. SATLike-c(w): Solver Description. In *MaxSAT Evaluation 2020: Solver and Benchmark Descriptions*, page 15. University of Helsinki, Department of Computer Science, 2020.

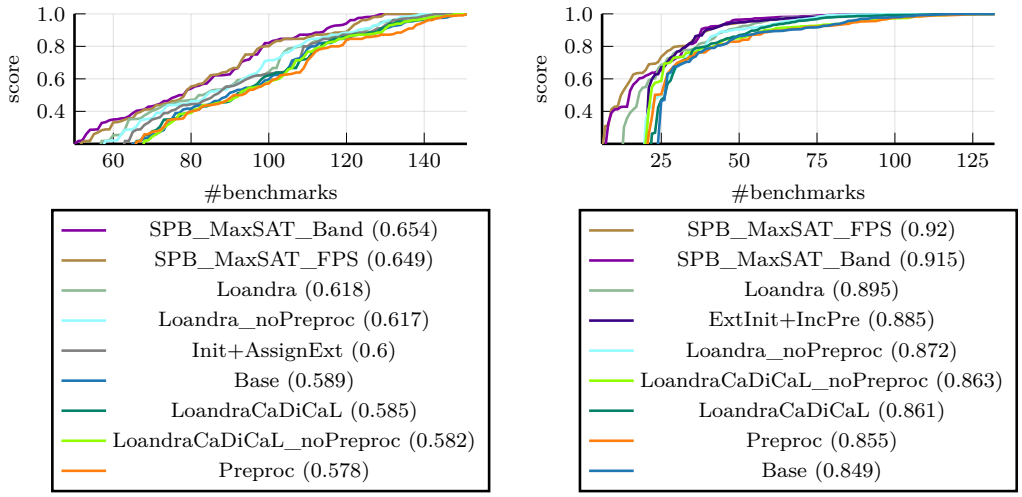
32 Zhendong Lei, Shaowei Cai, Fei Geng, Dongxu Wang, Yongrong Peng, Dongdong Wan, Yiping Deng, and Pinyan Lu. SATLike-c: Solver Description. In *MaxSAT Evaluation 2021: Solver and Benchmark Descriptions*, pages 19–20. University of Helsinki, Department of Computer Science, 2021.

33 Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Combining Clause Learning and Branch and Bound for MaxSAT. In *27th International Conference onf Principles and Practice of Constraint Programming*, volume 210 of *LIPIcs*, pages 38:1–38:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.38.

34 Ole Lübke. BouMS. Zenodo, March 2025. doi:10.5281/zenodo.14999498.

35 João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, volume 336 of *FAIA*, pages 133–182. IOS Press, 2021. doi:10.3233/FAIA200987.

36 Antonio Morgado, Carmine Dodaro, and Joao Marques-Silva. Core-Guided MaxSAT with Soft Cardinality Constraints. In *Principles and Practice of Constraint Programming*, volume 8656 of *LNCS*, pages 564–573. Springer, 2014. doi:10.1007/978-3-319-10428-7_41.



(a) unweighted. **(b)** weighted.

Figure 5 Anytime scores comparing the best performing configuration to other state-of-the-art solvers.

37 António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013. doi:10.1007/s10601-013-9146-2.

38 Alexander Nadel. Polarity and Variable Selection Heuristics for SAT-Based Anytime MaxSAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 12(1):17–22, 2020. doi:10.3233/SAT-200126.

39 Alexander Nadel. TT-Open-WBO-Inc-24: An Anytime MaxSAT Solver Entering MSE'24. In *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*, page 21. University of Helsinki, Department of Computer Science, 2024.

40 Alexander Nadel. TT-Open-WBO-Inc: An Efficient Anytime MaxSAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 15(1):1–7, 2024. doi:10.3233/SAT-231504.

41 Nina Narodytska and Fahiem Bacchus. Maximum Satisfiability Using Core-Guided MaxSAT Resolution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014. doi:10.1609/aaai.v28i1.9124.

42 Tobias Paxian, Sven Reimer, and Bernd Becker. Dynamic Polynomial Watchdog Encoding for Solving Weighted MaxSAT. In *Theory and Applications of Satisfiability Testing*, volume 10929 of *LNCS*, pages 37–53. Springer, 2018. doi:10.1007/978-3-319-94144-8_3.

43 André Schidler and Stefan Szeider. Structure-guided local improvement for maximum satisfiability. In *CP*, volume 307 of *LIPIcs*, pages 26:1–26:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CP.2024.26.

44 Pouya Shati, Eldan Cohen, and Sheila A. McIlraith. SAT-based Optimal Classification Trees for Non-Binary Data. *Constraints*, 28(2):166–202, 2023. doi:10.1007/s10601-023-09348-1.

45 Alice E. Smith and David W. Coit. Penalty Functions. In *Handbook of Evolutionary Computation*, pages C5.2:1–C5.2:6. IOP Publishing and Oxford University Press, 1997.

A Detailed Comparison to State-of-the-Art Solvers

Figure 3 shows a comparison of the best combinations of techniques described in this paper to the winners of the MSE 2024, SPB_MaxSAT_Band and SPB_MaxSAT_FPS [28], and Loandra as it participated in the same MSE. To assess the performance of our resulting solvers

we include variations of Loandra with sophisticated preprocessing disabled (_noPreproc) and with Glucose exchanged for CaDiCaL (LoandraCaDiCaL). Keep in mind, that Preproc (Base) is LoandraCaDiCaL(_noPreproc) without any SLS.

We can see that Loandra performs worse with CaDiCaL than with Glucose, and preprocessing is also much less effective, or even detrimental for this variant. Given the reduced performance of Base, it is no surprise that our best configurations cannot outperform the best anytime solvers currently available. Still, the substantial improvements over Base show that the techniques described in this paper effectively improve the performance of anytime core-boosted linear search.

From Prediction to Action: A Constraint-Based Approach to Predictive Policing

Younes Mechqrane[1]
Ai movement – International Artificial Intelligence Center of Morocco,
University Mohammed VI Polytechnic, Rabat, Morocco

Ismail Elabbassi
Ai movement – International Artificial Intelligence Center of Morocco,
University Mohammed VI Polytechnic, Rabat, Morocco

Abstract

Crime prevention in urban environments demands both accurate crime forecasting and the efficient deployment of limited law enforcement resources. In this paper, we present an integrated framework that combines a machine learning module (i.e. `PredRNN++` [27]) for spatiotemporal crime prediction with a constraint programming module for patrol route optimization. Our approach operates within the `ICON loop` framework [1], facilitating iterative refinement of predictions and immediate adaptation of patrol strategies. We validate our method using the City of Chicago Crime Dataset. Experimental results show that routes informed by crime predictions significantly outperform strategies relying solely on historical patterns or operational constraints. These findings illustrate how coupling predictive analytics with constraint programming can substantially enhance resource allocation and overall crime deterrence.

2012 ACM Subject Classification Applied computing → Law, social and behavioral sciences

Keywords and phrases Inductive Constraint Programming (`ICON`) `Loop`, Next Frame Prediction, `PredRNN++`

Digital Object Identifier 10.4230/LIPIcs.CP.2025.29

1 Introduction

Crime prevention is a key concern in urban management because it directly affects public safety and quality of life. One of the main challenges for law enforcement agencies is deciding how to use their limited resources most effectively. Over the years, researchers have shown that accurate crime prediction models can help police forces act proactively. By anticipating when and where crimes are likely to occur, patrol units can be deployed more efficiently, enhancing their overall impact [25].

Detecting areas with high crime activity, known as crime "hotspots," has been a major focus of criminological studies. Early work, such as the concept of "prospective hot-spotting" by Bowers et al. [3], demonstrated that concentrating on locations with recent crime surges can guide future crime prevention. Subsequent research has used more advanced approaches, like kernel density estimation (KDE) [14, 11] and spatiotemporal modeling [5], to identify evolving crime hotspots.

In parallel, artificial intelligence and machine learning methods have improved both the accuracy and depth of crime prediction. Deep learning architectures, such as convolutional neural networks (CNNs) and long short-term memory (LSTM) networks, are now frequently used to forecast spatiotemporal crime patterns [21, 12, 10, 18]. Meanwhile, multi-density clustering methods [4] are also integrated for improved forecasting.

[1] Corresponding author

© Younes Mechqrane and Ismail Elabbassi;
licensed under Creative Commons License CC-BY 4.0

Other researchers combine decomposition methods and network-based approaches to address short-term crime fluctuations. For example, Zhu et al. [30] propose a hierarchical crime prediction framework that merges a modified gated Graph Convolutional Network with Variational Mode Decomposition, modeling frequency-specific temporal dependencies.

Beyond these frameworks, some studies integrate environmental features and demand-based predictions. Lin et al. [19] develop a deep neural network that fuses criminal environmental data with spatiotemporal crime features for grid-based forecasting, while Ke et al. [15] introduce an end-to-end CNN model that predicts on-demand ride services, offering potential insights for crime-related event prediction.

Once crime-prone areas have been identified, another crucial task is to optimize how patrols are assigned so that resources are used effectively. Various mathematical models, such as integer and mixed-integer programming [7, 17, 23], schedule patrol units with minimal cost and travel time. In addition, metaheuristics (e.g., genetic algorithms, ant colony optimization) are also used to design efficient routes [6, 16, 20], helping agencies balance coverage in high-risk areas with practical constraints [26, 8].

Despite these advances, there remains a notable gap between crime forecasting and patrol optimization. Most predictive models generate heatmaps or highlight hotspots but do not integrate these insights into fully actionable patrol schedules. Conversely, patrol-optimization frameworks often rely on static or simplistic crime data, overlooking the nuanced forecasts produced by modern machine learning.

In this paper, we bridge this gap by presenting a framework that combines machine learning and constraint programming using the Inductive Constraint Programming Loop (`ICON loop`) [1]. Specifically, we develop a crime prediction model based on `PredRNN++` [27], capturing spatiotemporal crime patterns and updating dynamically as new data arrive. We then embed this forecasting module within a constraint programming framework that optimizes patrol routes based on projected crime risks and operational constraints, ensuring available resources are allocated with maximum effectiveness.

By adopting the `ICON loop` paradigm, our approach establishes a continuous feedback loop: crime forecasts inform patrol planning, and the impact of patrols feed back into subsequent predictive refinements. This synergy between advanced crime forecasting and combinatorial route optimization enables more adaptive and impactful patrol strategies than previous methods, which have largely tackled these challenges in isolation.

The remainder of this paper is organized as follows. Section 2 provides a recap of the `ICON loop`. Section 3 describes the crime data processing pipeline. Section 4 details our predictive modeling approach. Section 5 presents the constraint programming module and patrol route optimization. Section 6 reports the experimental setup and results, and Section 7 concludes.

2 Technical Background

The Inductive Constraint Programming Loop (`ICON loop`) [1] is a framework designed to seamlessly integrate machine learning (`ML`) with constraint programming (`CP`). Its main objective is to enable constraint models to adapt to changing environments by drawing on observations collected from the World (`W`). As shown in Figure 1, the `ICON loop` consists of three principal components: the `CP` module, the `ML` module, and the `W` module.

The high-level procedure is outlined in Algorithm 1. At the start of each cycle, new observations are gathered from the `W` component and recorded in the `Observations` repository. These observations are then passed in parallel to the `CP` and `ML` modules via `World-to-CP` and `World-to-ML` (lines 7, 3). To form the learning problem L, the `World-to-ML` channel

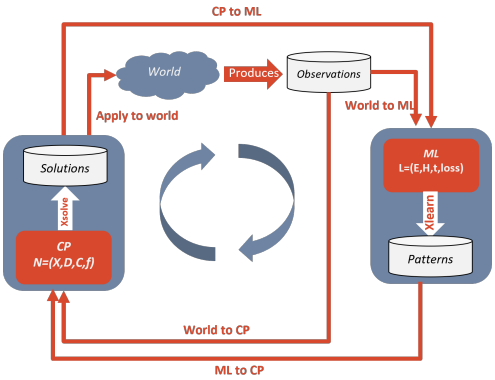


Figure 1 An overview of the inductive constraint programming loop.

(line 3) extracts features or insights directly from the world data, and the `CP-to-ML` channel (line 4) can incorporate information derived from previous solutions, if available. The outputs of these channels collectively define L, which is then processed by `applyXLearn` (line 6) to generate new patterns or hypotheses.

Next, the constraint network is updated. Observations feed into the `CP` module via `World-to-CP` (line 7), and the newly derived patterns are injected through `ML-to-CP` (line 8). The `constructN` function (line 9) merges these two inputs to build or refine the constraint network N. Finally, `applyXSolve` (line 10) generates a set of candidate solutions for the updated constraints network.

These solutions are then applied to the world using `Apply-to-World`. If the solutions turn out to be invalid or inapplicable, the loop immediately begins a new iteration to refine the candidate solutions. If a solution is successfully applied to the world, the cycle concludes, and new observations can be gathered in preparation for the next cycle.

Algorithm 1 `ICON loop`.

```
1  Function CYCLE(Observations, optional PreviousSolutions):
2      repeat
3          L_o ← World-to-ML(Observations);
4          L_p ← CP-to-ML(PreviousSolutions);
5          L ← constructL(L_o, L_p);
6          Patterns ← applyXLearn(L);
7          N_o ← World-to-CP(Observations);
8          N_p ← ML-to-CP(Patterns);
9          N ← constructN(N_o, N_p);
10         Solutions ← applyXSolve(N);
11     until Apply-to-World(Solutions);
```

Not all channels in the `ICON loop` need to be active in every scenario [1]. In our setting, we assume that the machine learning step relies solely on newly collected observations rather than any previous solutions. Therefore, the `CP-to-ML` channel is omitted, and L_p is not included in constructing the learning problem L. This streamlined approach simplifies the loop without compromising its ability to adapt to new environmental data.

Algorithm 2 `World-to-ML`.

Function *`World-to-ML`(CrimeDataset, q, $s_{\max}$)*:
  Initialize empty linear sequence $\{Q^t\}$;
  foreach $w \in \{1, \ldots, n_weeks\}$ **do**
    foreach $d \in \{1, \ldots, 7\}$ **do**
      foreach $s \in \{1, \ldots, s_{\max}\}$ **do**
        $t \leftarrow (w, d, s)$;
        $Crimes \leftarrow$ `CrimeDataset`(t);
        $\hat{f}_t(\phi, \lambda) \leftarrow$ `estimatePDF_KDE`$(Crimes)$;
        Initialize $P^t \leftarrow []$;
        foreach $(i, j) \in I \times J$ **do**
          $P_{i,j,t} \leftarrow$ `integrate`$(\hat{f}_t(\phi, \lambda), i, j)$;
          `appendToMatrix`$(P^t, P_{i,j,t})$;
        $Q^t \leftarrow$ `applyQuantilesAndNormalize`(P^t, q);
        Append Q^t to linear sequence $\{Q^t\}$;
  $\{\mathcal{S}_1, \ldots, \mathcal{S}_{s_{\max}}\} \leftarrow$ `TransformToSlotSequences`$(\{Q^t\}, s_{\max})$;
  return $\{\mathcal{S}_1, \ldots, \mathcal{S}_{s_{\max}}\}$;
Function *`TransformToSlotSequences`($\{Q^t\}$, $s_{\max}$)*:
  Initialize empty sequences $\mathcal{S}_1, \ldots, \mathcal{S}_{s_{\max}}$;
  foreach *matrix Q^t with time index $t = (w, d, s)$* **do**
    Append Q^t to sequence $\mathcal{S}_s$;
  return $\{\mathcal{S}_1, \ldots, \mathcal{S}_{s_{\max}}\}$;

3 World Component

In this section, we describe how the data for the World (`W`) component is prepared for the machine learning (`ML`) module. We use the Chicago Crime Dataset from the City of Chicago Data Portal [24] as our source of observations. From this dataset, we focus on three key attributes essential for spatiotemporal prediction: latitude (ϕ), longitude (λ), and the time of occurrence. To systematically handle the city's geographic extent, we partition the study area into an $I \times J$ grid of rectangular cells, as shown in Figure 2 [2]. Each cell is labeled c_{ij}. This grid structure allows us to aggregate and analyze crimes at a manageable level of granularity.

Generating ML-Ready Features. Algorithm 2 (`World-to-ML`) describes how raw crime data is transformed into quantile-based matrices capturing the spatiotemporal distribution of crime incidents.

The algorithm begins by segmenting crime data into discrete intervals indexed by $t = (w, d, s)$, where w is the week number, d is the day of the week, and s is the time slot within the day (ranging from 1 to $s_{\max}$). For each interval t, we select the corresponding subset of crime events (Figure 3.A) and estimate a continuous spatial probability density function $\hat{f}_t(\phi, \lambda)$ through KDE (Figure 3.B).

[2] In our experiments, we set $I = 100$ and $J = 100$.

From this estimated density, we compute a probability matrix P^t by integrating $\hat{f}_t(\phi, \lambda)$ over each grid cell c_{ij}:

$$P_{i,j,t} = \int_{\phi_{i-1}}^{\phi_i} \int_{\lambda_{j-1}}^{\lambda_j} \hat{f}_t(\phi, \lambda)\, d\lambda\, d\phi.$$

Algorithm 3 `applyXLearn`.

Function *PrepareTrainingData*($\{\mathcal{S}_1, \ldots, \mathcal{S}_{s_{\max}}\}$, l_{input})**:**
  $X \leftarrow []$;
  $Y \leftarrow []$;
  foreach $\mathcal{S}_s$ *in* $\{\mathcal{S}_1, \ldots, \mathcal{S}_{s_{\max}}\}$ **do**
    for $i = 0$ **to** $|\mathcal{S}_s| - l_{input} - 1$ **do**
      $X_i \leftarrow \mathcal{S}_s\big[i : i + l_{\text{input}}\big)$;
      $Y_i \leftarrow \mathcal{S}_s\big[i + l_{\text{input}}\big]$;
      append(X, X_i);
      append(Y, Y_i);
  return (X, Y);
Function *TrainModel*(X, Y)**:**
  `XLearn` $\leftarrow$ `PredRNN++` ();
  train(`XLearn`, X, Y);
  return `XLearn`;
Function *Inference*(*XLearn*, $\mathcal{S}_s$, l_{input})**:**
  $X_{\text{recent}} \leftarrow \mathcal{S}_s\big[\,|\mathcal{S}_s| - l_{\text{input}} : |\mathcal{S}_s|\,\big)$;
  $\hat{Q}_s \leftarrow$ predict(`XLearn`, X_{recent});
  return $\hat{Q}_s$;
Function *applyXLearn* ($\{\mathcal{S}_1, \ldots, \mathcal{S}_{s_{\max}}\}$, l_{input}, *trainFlag*)**:**
  if *trainFlag* = *true* **then**
    $X, Y \leftarrow$ `PrepareTrainingData`$(\{\mathcal{S}_1, \ldots, \mathcal{S}_{s_{\max}}\}, l_{\text{input}})$;
    `XLearn` $\leftarrow$ `TrainModel`(X, Y);
  else
    `XLearn` $\leftarrow$ retrieveExistingModel();
  foreach $\mathcal{S}_s$ *in* $\{\mathcal{S}_1, \ldots, \mathcal{S}_{s_{\max}}\}$ **do**
    $\hat{Q}_s \leftarrow$ `Inference`$(\text{XLearn}, \mathcal{S}_s, l_{\text{input}})$;
    Store $\hat{Q}_s$ in `Patterns`;

The resulting probability matrix P^t encodes the likelihood of crime occurrences in each grid cell c_{ij} at time interval t. To enhance the robustness of the predictions, we apply a q-quantile transformation to each matrix P^t, discretizing the computed probabilities into q quantile-based classes ranging from 0 to $q-1$, and normalizing them into the interval $[0, 1)$.[3] This quantile-based normalization aims to facilitate the learning process of the forecasting model by reducing the influence of outliers and ensuring a more balanced representation of crime levels across the spatial grid. We denote the resulting quantile-transformed matrix as Q^t.

[3] In our experiments, we use $q = 20$.

Algorithm 4 `ML-to-CP`: Converting Predicted Heatmaps into Deployment Locations.

Function *ML-to-CP (cityGrid, densityThreshold p, circleRadius r, timeSlots S)*:
  `// We assume that for each` $s \in S$`, we have a predicted matrix` $\hat{Q}_s$
    `that provides risk values per cell, i.e.,` $\text{risk}_s(c) \equiv \hat{Q}_s(c)$.
  $allPolygons \leftarrow \emptyset$;
  foreach $s \in S$ **do**
    $hotspots_s \leftarrow \{ c \in cityGrid \mid \text{risk}_s(c) \geq \text{percentile}_s(cityGrid, p)\}$;
    $polygons_s \leftarrow \text{ClusterCells}(hotspots_s)$;
    $allPolygons \leftarrow allPolygons \cup polygons_s$;
  $allCircles \leftarrow \text{GenerateCircleGrid}(cityGrid, r)$;
  $validCircles \leftarrow []$;
  foreach $circle \in allCircles$ **do**
    foreach $polygon \in allPolygons$ **do**
      if $IntersectsOrInside(circle, polygon)$ **then**
        append($validCircles$, $circle$);
        break;
  $locations \leftarrow []$;
  foreach $circle \in validCircles$ **do**
    $tsRisk \leftarrow []$;
    foreach $s \in S$ **do**
      $circleRisk_s \leftarrow \sum\{ \text{risk}_s(c) \mid c \in cityGrid, \text{CellCompletelyInside}(c, circle)\}$;
      append($tsRisk$, $circleRisk_s$);
    append($locations$, (CircleCenter($circle$), $tsRisk$));
  return $locations$;

Each Q^t is appended to a linear sequence $\{Q^t\}$. Subsequently, to align the data format explicitly with the forecasting model's requirements, we use the function `TransformToSlotSequences` (described within Algorithm 2) to reorganize this linear sequence into separate slot-specific sequences. Each resulting sequence $\mathcal{S}_s$ contains matrices from the same daily slot s, arranged chronologically by days.

The final output is a set of slot-specific sequences $\{\mathcal{S}_1, \ldots, \mathcal{S}_{s_{\max}}\}$, which serves as input for the machine learning forecasting component.

4 Machine Learning Component

The machine learning component estimates future crime risk using spatiotemporal data provided by the `W` component, specifically the slot-specific sequences $\{\mathcal{S}_1, \ldots, \mathcal{S}_{s_{\max}}\}$ of quantile-based matrices generated by Algorithm 2 (Section 3). Our goal is to predict how crime patterns in each time slot will evolve (*one step ahead*) based on historical observations of that same slot from previous days.

To achieve this, we employ a `PredRNN++` architecture [27]. `PredRNN++` is a recurrent neural network specifically designed to predict future video frames based on a sequence of past frames. It processes each input frame through multiple recurrent layers, each composed of a specialized convolutional unit known as a *Causal LSTM*. Every Causal LSTM cell

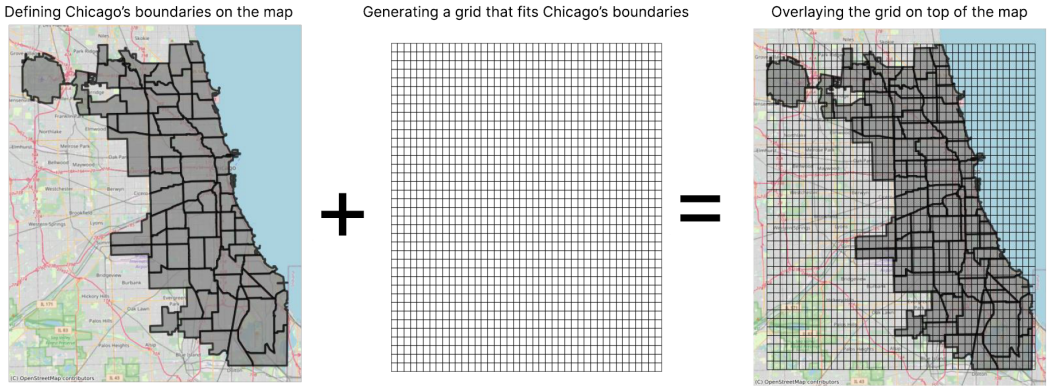


Figure 2 Partitioning the city area into an $I \times J$ grid of rectangular cells.

maintains two distinct memory states: a short-term spatial memory that captures local visual patterns (e.g., textures and shapes), and a long-term temporal memory that tracks motion dynamics across frames. These two memories are updated sequentially, with information flowing in a strictly causal direction ensuring that predictions adhere to the chronological structure of video sequences. To address the vanishing gradient problem, where learning signals degrade across long temporal horizons, `PredRNN++` introduces a *Gradient Highway Unit (GHU)*. This unit provides a shortcut path for gradients to propagate more directly across time, substantially improving training efficiency on long sequences.

Crime patterns typically exhibit strong daily cycles (e.g., distinct day vs. night patterns). This is why, rather than combining all frames into one continuous sequence, we maintain separate, dedicated sequences for each distinct time slot. Specifically, each sequence $\mathcal{S}_s$ comprises chronologically ordered quantile-transformed matrices $\{Q_s^d\}$, where d indexes consecutive days for the given time slot s. Although these slot-specific sequences are maintained separately, the `PredRNN++` model is trained jointly on data from all slots, allowing the model to learn both within-slot and across-slot temporal patterns.

Algorithm 3 presents the complete `applyXLearn` pipeline for this multi-slot forecasting task, where we predict a single future frame for each slot. This pipeline can operate in two modes, controlled by a boolean parameter `trainFlag`. If `trainFlag` is set to `true`, the algorithm prepares a new training dataset from the sequences and trains (or retrains) the model before performing inference. Otherwise, if `trainFlag` is set to `false`, it skips these steps, relying on a pre-trained model for direct inference.

When training is enabled, the function `PrepareTrainingData` (Algorithm 3, lines 1–9) iterates over each slot-specific sequence $\mathcal{S}_s$, generating input–output pairs using a sliding-window approach. For each index i, we create a training example:

$$X_i \;=\; \mathcal{S}_s\big[i : i + l_{\text{input}}\big), \quad Y_i \;=\; \mathcal{S}_s\big[i + l_{\text{input}}\big].$$

Here, X_i contains l_{input} frames, and Y_i is simply the next frame in the sequence. These generated input–output pairs from all slots are merged into a global training set (X, Y).

Next, `TrainModel` (Algorithm 3, lines 11–13) fits the `PredRNN++` model to this dataset. After training (or if a pre-trained model is used when `trainFlag` is `false`), the `Inference` function (Algorithm 3, lines 15–17) employs the most recent l_{input} frames from each slot's sequence to forecast the next frame. Each slot's prediction $\hat{Q}_s$ encodes the anticipated spatial crime-risk distribution in that slot for the next day.

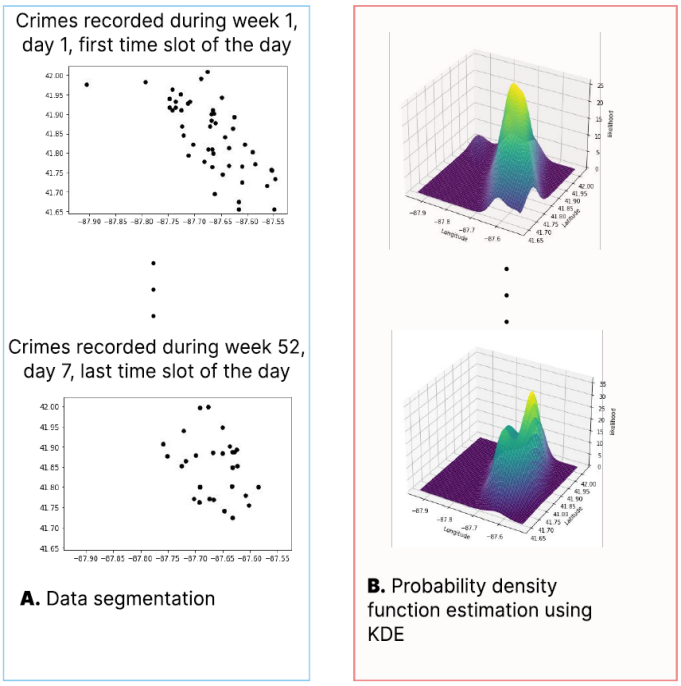


Figure 3 Data segmentation and probability density estimation.

Finally, the main `applyXLearn` pipeline (Algorithm 3, lines 19–27) orchestrates these steps depending on whether training is requested (`trainFlag` = `true`) or not.

The slot-specific predictions $\{\hat{Q}_s\}$ are stored in the `Patterns` repository. Typically, the resulting predictions highlight extensive continuous regions of elevated crime risk that are impractical for direct patrol assignment (Figure 4.A). Therefore, before transferring these predicted heatmaps to the constraint programming (`CP`) module, we apply the `ML-to-CP` procedure (Algorithm 4) to convert these broad, forecasted hotspots into precise, actionable deployment locations suitable for operational patrol planning.

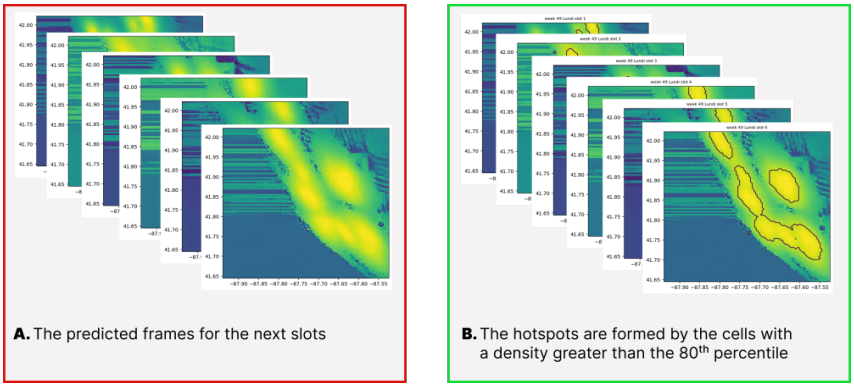


Figure 4 Hotspots identification in contiguous time slots.

In practice, patrol operations typically focus on a contiguous subset of time slots within a specific day, denoted S (for example, a work shift). That is, algorithm 4 converts predicted heatmaps into deployment locations by detecting high-risk cells within each time slot $s \in S$

using a p-th percentile threshold. Following [3], we set p to the 85%-percentile so that only the top 15% of predicted intensities are considered hotspots, thereby reducing the impact of outliers. In Step 1 of the algorithm (lines 1–6), each time slot $s \in S$ (line 3) is processed by first identifying high-risk cells (line 4), then clustering these cells into contiguous polygons using standard 4-connectivity where neighboring high-risk cells are merged into connected components (line 5) (Figures 4.B, 5.A), and finally merging the resulting polygons into the set *allPolygons* (line 6). Next, in Step 2 (line 7), a uniform grid of circles is generated over the city grid to form *allCircles* (Figure 5.B). In Step 3 (lines 8–13), every circle in *allCircles* (line 9) is examined against each polygon in *allPolygons* (line 10), and if the circle satisfies the IntersectsOrInside condition (line 11), it is appended to the list *validCircles* (line 12) and the inner loop is exited (line 13) (Figure 5.C). Finally, in Step 4, for each valid circle (line 15) the algorithm computes the time-slot-specific risk by summing the risk values of all cells completely inside the circle for each slot $s \in S$ (line 18). The circle's center, paired with its list of per-slot risks, is then added to *locations* (line 20), producing as output a set of pairs $\big(\text{CircleCenter}, \mathit{tsRisk}\big)$, where *tsRisk* contains a risk value for every time slot in S.

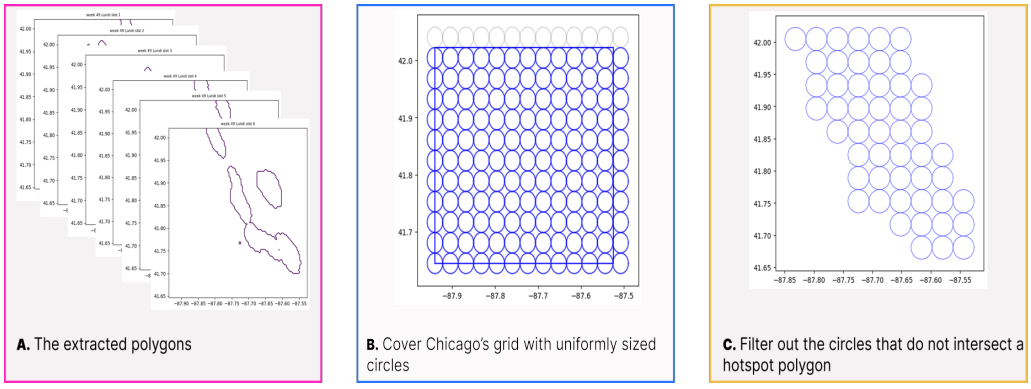


Figure 5 Potential deployment locations identification.

5 Constraint Programming Formulation

Building on the refined set of high-risk *deployment locations* identified via the `ML-to-CP` step, we formulate a vehicle routing problem (VRP) to optimize patrol unit deployment. The problem is defined on a directed graph $G = (\mathcal{L}, \mathcal{A})$, where $\mathcal{L} = \{0, 1, \ldots, n_{\text{loc}}\}$ represents locations (with location 0 denoting the depot) and $\mathcal{A} \subseteq \mathcal{L} \times \mathcal{L}$ defines admissible travel arcs. The subset $\mathcal{C} = \mathcal{L} \setminus \{0\}$ corresponds to the *deployment locations* selected by `ML-to-CP`. The planning horizon is discretized into n_s slots, $S = \{1, \ldots, n_s\}$, each of duration `slot_length`. Patrol routes must be completed within a maximum allowable time `maxT`.

We consider a set of patrol units (vehicles), $\mathcal{V} = \{1, \ldots, n_v\}$. Each location $i \in \mathcal{L}$ has a service time `service_t`$[i] \in \mathbb{N}$. Each deployment location $i \in \mathcal{C}$ has a time-varying risk value `risk`$[i, s] \in \mathbb{R}^+$ for each slot $s \in S$. Travel times between locations i and j are given by `travel_t`$[i, j] \in \mathbb{N}$.

For each vehicle v, we introduce a variable `route_length`$[v]$ indicating how many locations that vehicle visits (including its first position). We also allow each patrol to start at *any* location at time 0.

The decision variables can be summarized as follows:

- $\texttt{route}_{v,k} \in \mathcal{L}$: the k-th location visited by vehicle v.
- $\texttt{arrive_t}_{v,k}, \texttt{depart_t}_{v,k}, \texttt{wait_t}_{v,k} \in \mathbb{N}$: arrival, departure, and waiting times at the k-th location of vehicle v.
- $\texttt{slot}_{v,k} \in \{0\} \cup S$: time slot during which vehicle v services its k-th location (0 indicates either the depot or an unused slot).
- $\texttt{route_length}[v] \in \{1, \ldots, n_{\text{loc}}\}$: total number of visited locations by vehicle v.

The key constraints are as follows:

- Vehicles may begin at any location at time 0:

$$\texttt{arrive_t}_{v,1} = 0, \quad \texttt{route}_{v,1} \in \mathcal{L}, \forall v \in \mathcal{V}.$$

- Beyond the route length of a vehicle v (i.e. $\texttt{route_length}[v]$), we force the following decision variables to be 0:

$$\texttt{route}_{v,k} = 0, \quad \texttt{slot}_{v,k} = 0, \quad \texttt{wait_t}_{v,k} = 0, \forall v \in \mathcal{V},\ \forall k > \texttt{route_length}[v].$$

- No vehicle returns to the depot prematurely:

$$\texttt{route}_{v,k} \neq 0, \forall v \in \mathcal{V},\ \forall k \leq \texttt{route_length}[v].$$

- A vehicle must complete its service time and any waiting time before departing:

$$\texttt{depart_t}_{v,k} = \texttt{arrive_t}_{v,k} \ + \ \texttt{service_t}[\texttt{route}_{v,k}] \ + \ \texttt{wait_t}_{v,k}, \forall v \in \mathcal{V},\ \forall k.$$

- The arrival time at the next location depends on the departure time from the current one plus travel time:

$$\texttt{arrive_t}_{v,k+1} = \texttt{depart_t}_{v,k} \ + \ \texttt{travel_t}\big(\texttt{route}_{v,k}, \texttt{route}_{v,k+1}\big),$$
$$\forall v \in \mathcal{V},\ \forall k < \texttt{route_length}[v].$$

- If the location is a deployment location, determine the slot by (Mapping Arrival Times to Slots):

$$\texttt{slot}_{v,k} = \begin{cases} 1 + \left\lfloor \dfrac{\texttt{arrive_t}_{v,k}}{\texttt{slot_length}} \right\rfloor, & \text{if } \texttt{route}_{v,k} \in \mathcal{C}, \\ 0, & \text{otherwise.} \end{cases} \qquad \forall v \in \mathcal{V},\ \forall k.$$

- Prevent two vehicles from serving the same deployment location simultaneously:

$$\big(\texttt{route}_{v_1,i} = \texttt{route}_{v_2,j} \in \mathcal{C}\big) \implies$$
$$\Big(\texttt{depart_t}_{v_1,i} \ \leq \ \texttt{arrive_t}_{v_2,j} \ \vee \ \texttt{depart_t}_{v_2,j} \ \leq \ \texttt{arrive_t}_{v_1,i}\Big), \forall v_1 < v_2,\ \forall i, j.$$

Objective Function. The goal is to maximize the overall "risk coverage" achieved by assigning patrol vehicles to specific high-risk locations at times when risks are high. Specifically, each visit to a deployment location i in time slot s contributes $\texttt{risk}[i, s]$ to the objective. Because vehicles follow routes and visit multiple locations sequentially, the objective function

$$\max \sum_{v \in \mathcal{V}} \ \sum_{k=1}^{\texttt{route_length}[v]} \texttt{risk}[\texttt{route}_{v,k}, \texttt{slot}_{v,k}]$$

sums the risk values for every $(\texttt{route}_{v,k}, \texttt{slot}_{v,k})$ pair across all vehicles v and all visited points k. In other words, whenever vehicle v visits location $\texttt{route}_{v,k}$ during slot $\texttt{slot}_{v,k}$, that location's time-dependent risk $\texttt{risk}[\texttt{route}_{v,k}, \texttt{slot}_{v,k}]$ is accrued toward the total. Hence, the model is driven to schedule patrol units so that the highest-risk locations are visited at the most impactful time slots, thereby maximizing the sum of all risk covered during the planning horizon.

6 Results

In this section, we present and discuss the training of our `XLearn` model and the subsequent results obtained from both validation and testing. We further demonstrate the effectiveness of the proposed strategy through `ICON loop` simulations.

6.1 Training `XLearn`

The core of our `ML` component is the `XLearn` model, which builds on the `PredRNN++` architecture [27]. We use the publicly available `PredRNN++` implementation provided in the OpenSTL repository[4]. Our experiments adopt the default architecture and training configuration, except for the number of recurrent layers, which we increase from four to six by adjusting the hidden layer structure. All other hyperparameters – including convolutional filter size, patch size, learning rate, batch size, and the scheduled sampling schedule – remain unchanged. The model is trained with cell-wise Mean-Squared Error.

Data Preparation and Setup

As mentioned earlier, we evaluated `XLearn` on the City of Chicago Crime Dataset [24], focusing on reported incidents from 2023. After cleaning, the processed data amounted to 261 798 records, from which we retained only latitude, longitude, and time as relevant features. Following Section 3, `World-to-ML` transformed these raw crime observations into a sequence of quantile-based matrices, $\{Q^t\}$. With one year split into 52 weeks, 7 days per week, and 48 time slots of half an hour per day, we obtained 17 472 matrices, each encoding the spatial distribution of crime risk at a specific slot.

Model Training

We implemented `applyXLearn` (Algorithm 3) to form training pairs (X_i, Y_i). Afterwards, an 80%–15%–5% split was applied for training, validation, and testing, respectively. Various input-sequence lengths were tested, and a good configuration was found at $l_{\text{input}} = 10$. The output-sequence length $l_{\text{output}} = 1$ corresponds to a single prediction step (i.e., forecasting the upcoming time slot of the next day).

We trained for 300 epochs on an HPC equipped with an NVIDIA A100-GPU, running CUDA 12.2. The GPU had a maximum memory capacity of 81 920 MiB. Figure 7A plots the training and validation loss curves over 300 epochs. Both curves show steady progress and minimal overfitting.

We assessed model accuracy and fidelity using four widely adopted metrics:

- Mean Squared Error (`MSE`) measures the average squared differences between predictions and ground-truth data;

[4] `https://github.com/chengtan9907/OpenSTL`

- Peak Signal-to-Noise Ratio (`PSNR`) [13], expressed in decibels (dB), quantifies reconstruction fidelity (higher values, typically 30–50 dB, denote better quality);
- Structural Similarity Index (`SSIM`) [28], which evaluates structural alignment between images on a scale of 0 to 1.0 (1.0 corresponds to perfect match);
- Learned Perceptual Image Patch Similarity (`LPIPS`) [29], a deep-learning metric scored between 0 and 1, where lower values indicate greater perceptual resemblance in feature space (0 for identical images).

Figures 7B–7D depict the `PSNR`, `SSIM`, and `LPIPS` metrics over training epochs. Our best validation metrics reached:

- `PSNR` ≈ 31.47, indicating a high-fidelity reconstruction,
- `SSIM` ≈ 0.96, reflecting strong structural alignment with the ground truth,
- `LPIPS` ≈ 0.02, suggesting high perceptual similarity.

On the held-out test set, the model consistently achieved: `SSIM` $= 0.82$, `PSNR` $= 27.11$, `LPIPS` $= 0.04$, indicating that `XLearn` generalizes well: it preserves essential structure (`SSIM`), yields visually coherent predictions (`PSNR`), and remains perceptually faithful (`LPIPS`) (Figure 6).

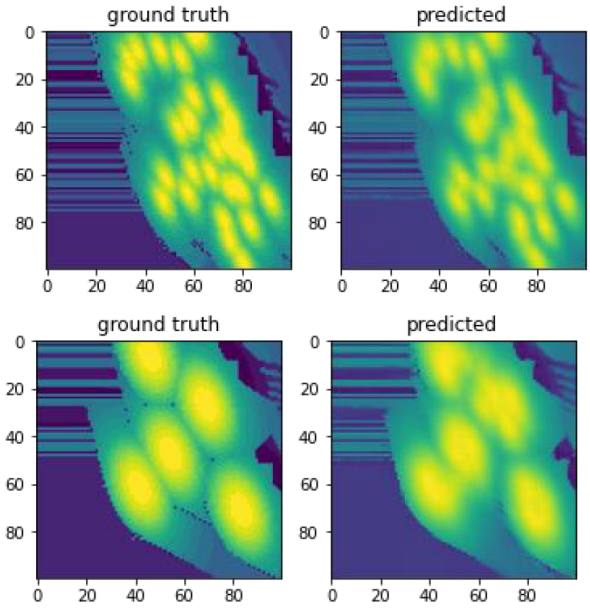
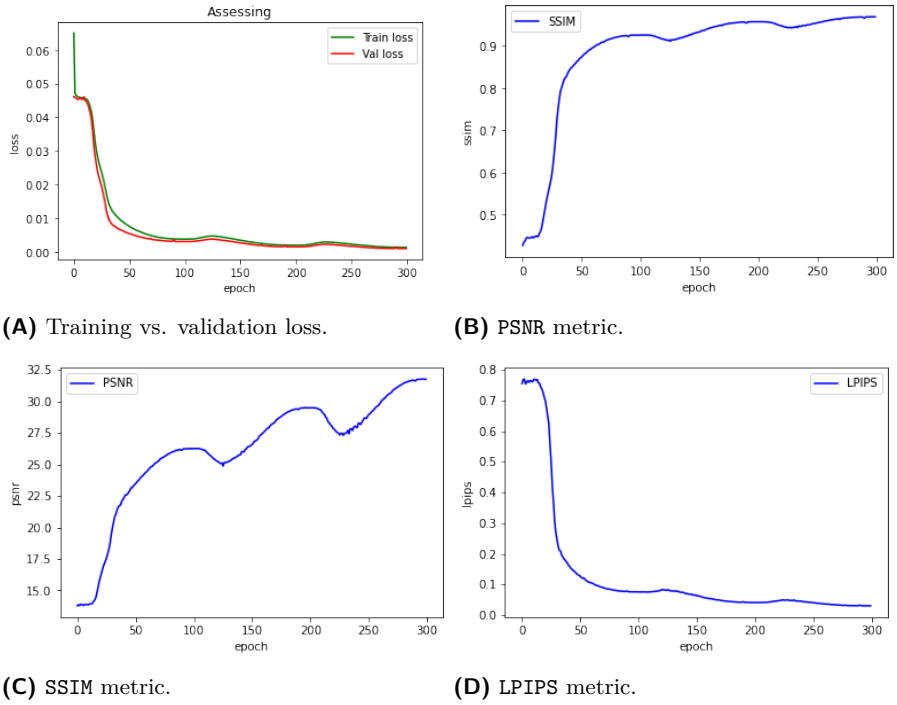


Figure 6 Examples of predicted heatmaps.

Overall, these results confirm that the `PredRNN++` `XLearn` effectively captures spatial-temporal crime patterns, producing robust and high-quality forecasts that are well-suited for guiding subsequent patrol route optimization.

6.2 Simulating the `ICON loop`

We conducted our simulations on a computer equipped with a 12th Gen Intel® Core™ i7-12700H processor (2.30 GHz), 32 GB RAM, and a 64-bit Windows operating system. We set a fixed service time of 20 minutes per location. Travel distances were computed using the OSMnx Python library [2], assuming a constant travel speed of 60 km/h.



(A) Training vs. validation loss.

(B) `PSNR` metric.

(C) `SSIM` metric.

(D) `LPIPS` metric.

Figure 7 Key performance metrics.

To maintain operational feasibility, we set the radius of each deployment location circle to 2 km. This radius ensures that traversing the circle's perimeter ($\approx$ 12.57 km) takes approximately 12.6 minutes, fitting comfortably within the 20-minute service duration. Hence, a 2 km radius allows sufficient time to patrol the perimeter fully, make strategic movements, or handle additional points of interest within the deployment area. By contrast, increasing the radius to 3 or 4 km would expand the perimeter to roughly 18.85 or 25.13 km, complicating thorough coverage of the deployment area.

We performed simulations using week 49 of 2023 (unseen during training) from the Chicago Crime Dataset [24]. Each cycle of the `ICON loop` spans three hours, subdivided into six consecutive 30-minute time slots (so $s_{\max} = 48$ for the full day, but only 6 slots per cycle). Thus, each day includes 8 cycles, and the full simulation week consists of 56 cycles. After each cycle, simulation time is advanced by exactly 3 hours. The vehicle routing problem is solved for each individual 3-hour cycle. We examined three distinct approaches for crime deterrence:

Predicted Hotspots. This strategy combines forecasting and patrol planning in three main steps:

1. **Forecasting crime risk.** Each day is divided into 48 time slots of 30 minutes. For any 3-hour deployment cycle, we focus on a window of 6 consecutive slots $s_1, s_2, \ldots, s_6$. For each slot s_i, the `World-to-ML` function retrieves the historical sequence of crime-density matrices $\{Q^d_{s_i}\}$ over the previous $l_{\text{input}} = 10$ days. These sequences are passed to the `applyXLearn` function, which, using a pre-trained model (`trainFlag` = false), generates forecasts $\{\hat{Q}_{s_1}, \hat{Q}_{s_2}, \ldots, \hat{Q}_{s_6}\}$ representing predicted crime risk in each time slot.

2. **Identifying deployment locations.** The predicted risk matrices are then processed by the `ML-to-CP` module. This step transforms each heatmap into a set of actionable patrol locations by applying a quantile-based threshold (e.g., top 15% of risk values) and clustering nearby high-risk cells. The result is a list of circular zones likely to contain high-risk activity, suitable for patrol deployment.
3. **Optimizing patrol routes.** Using operational parameters provided by `World-to-CP` – such as the number of available patrol units and maximum route duration – the `applyXsolve` function builds a constraint network incorporating the selected deployment zones. The `CP` solver then computes patrol routes that maximize total crime risk coverage within the cycle's 3-hour window.

Historical Hotspots. This alternative approach replaces forecasting with a simple aggregation of past observations:

1. **Averaging past crime patterns.** For each of the six 30-minute slots in the 3-hour deployment window $s_1, s_2, \ldots, s_6$, we collect the corresponding historical matrices $\{Q^d_{s_i}\}$ from the previous $l_{\text{input}} = 10$ days. By computing the element-wise average across days, we obtain $\bar{Q}_{s_1}, \bar{Q}_{s_2}, \ldots, \bar{Q}_{s_6}$, which serve as estimated crime-risk distributions for each slot.
2. **Extracting hotspot locations.** These averaged heatmaps are then passed to the `ML-to-CP` module, which applies the same quantile-based thresholding and clustering procedure as in the predicted hotspot approach. The result is a set of patrol deployment zones derived from historical trends.
3. **Route optimization.** Given these deployment locations and the real-world constraints provided by `World-to-CP`, the `CP` solver computes optimized patrol routes to maximize the total historical risk covered during the cycle.

Constraint Satisfaction. This approach serves as a baseline that prioritizes feasibility over risk-based optimization:

1. Random selection of locations: A set L of approximately 50 circles is selected randomly. This quantity mirrors the scale of candidate locations derived by the other approaches. Crucially, these circles are chosen without reference to any crime data, resulting in an arbitrary spatial distribution.
2. No risk prioritization: Patrol routes are computed among these randomly placed circles, adhering only to basic feasibility constraints. Consequently, this approach provides a baseline for comparison, illustrating the effect of disregarding crime-intensity indicators entirely.

Once patrol routes are fixed for a 3-hour cycle, the deterrence rate is determined as follows:

For each vehicle $v \in \mathcal{V}$, let $\mathcal{P}_v$ be the set of points visited during that cycle. Define the number of deterred crimes for vehicle v, denoted n_v, as

$$n_v \;=\; \sum_{p \in \mathcal{P}_v} \Big| \big\{ \text{cr} \;\big|\; \text{distance}(\text{cr}, p) \le 2 \text{ km},\; \texttt{arrive_t}[v,p] \le t(\text{cr}) \le \texttt{depart_t}[v,p] \big\} \Big|,$$

where $\text{distance}(\text{cr}, p)$ is the distance from crime cr to point p, and $t(\text{cr})$ is the time at which crime cr occurred. Thus, for each visited point p, we count all crimes within the patrol's deployment circle of radius 2 km that fall between arrival and departure times.

The overall deterrence rate for the cycle is then computed as

$$\text{Hit Rate} = \frac{\sum_{v \in \mathcal{V}} n_v}{N},$$

where N is the total number of crimes in the study area during the same 3-hour window. This process is repeated for each cycle in our evaluation, and the final hit rate is the mean across all cycles.

Table 1 Average Hit Rates (%) for varying patrol units.

Scenarios	Predicted Hotspots	Historical Hotspots	Constraint Satisfaction
2 patrols	8	3	0.1
4 patrols	14	6	0.2
6 patrols	19	8	0.3
8 patrols	24	10	0.4
10 patrols	26	13	0.7

Table 1 shows the average hit rates (i.e., the fraction of crimes deterred) for each approach when varying the number of patrol units. The *Predicted Hotspots* method consistently achieves the highest hit rates, highlighting the benefits of using forecasts to guide where patrols should be deployed. Even with only two patrols, this approach yields a deterrence rate of about 8%, which further increases to 26% when ten patrols are used. This improvement illustrates how forecasting-based route assignments can be scaled up to maintain significant coverage in larger forces.

The *Historical Hotspots* method performs moderately well, with hit rates surpassing those of the random baseline but trailing behind the predicted hotspot approach. As the number of patrols increases, however, the historical method still benefits from having more units to cover broader areas.

The *Constraint Satisfaction* approach serves as a baseline. Due to its purely random location selection and route generation, the resulting hit rates are understandably quite low (ranging from 0.1% to 0.7%), underscoring the importance of using forecasts in patrol routing. Despite having a comparable number of potential deployment circles to the other approaches, the absence of risk prioritization substantially diminishes this method's effectiveness.

6.3 Discussion: Closing the `ICON loop`

The results confirm that forecasts generated by `PredRNN++` can guide the CP-based patrol-routing model to cover significantly more `risk` than baseline strategies, validating the usefulness of the `ML-to-CP` channel within each `ICON loop` cycle. However, in the present work, the `CP-to-ML` direction (Figure 1) was not exploited, meaning that the information flow remains one-way: forecasts influence patrol plans, but the effectiveness of those plans is not used to improve future predictions. This feedback is critical in operational settings, as patrol deployment can displace or suppress crime and thus modify the spatio-temporal patterns that `XLearn` must learn in subsequent cycles.

A promising way to activate the `CP-to-ML` channel is to adopt the *Smart Predict then Optimize* framework with its surrogate loss SPO+ [9, 22]. The idea is to incorporate feedback from each `ICON loop` cycle to fine-tune `PredRNN++` based on the quality of the patrol plans it enables – measured here by the total *risk coverage* defined in the objective function.

The procedure may unfold as follows:

1. **Observed pass.** At the end of a given `ICON loop` cycle, the most recent crime data is used to construct an *observed* space–time risk map `risk`. Solving the CP-based deployment model with this map yields a reference patrol plan. The total `risk` covered – computed as the sum of $\texttt{risk}[\texttt{route}_{v,k}, \texttt{slot}_{v,k}]$ across all vehicles v and visited locations k – is recorded.
2. **Feedback.** Recall that, earlier in the same cycle, a patrol plan was already computed using the predicted map `risk` produced by `PredRNN++`. The difference in total risk coverage between the reference plan (based on observed data) and the forecast-based plan can be interpreted as a loss. SPO+ can then be used to convert this gap into a gradient, which is backpropagated to fine-tune `PredRNN++`, encouraging it to generate forecasts that lead to more effective patrol deployments in future cycles.

This decision-aware fine-tuning requires only one additional solve per cycle – on the observed risk map – and can be performed offline without disrupting operations. It effectively closes the `ICON loop` by turning each patrol deployment into both a consequence of current forecasts and a source of supervision for future predictions. By doing so, the learning objective of the ML module is aligned with the operational goal of maximizing total risk coverage throughout the planning horizon.

7 Conclusion

This paper introduced a data-driven framework that integrates predictive modeling and constraint-based optimization to improve the efficiency of police patrol deployments. By using a `PredRNN++` approach to generate spatiotemporal crime forecasts, our system anticipates evolving crime risks. These forecasts are then mapped into actionable hotspots via quantile transformations and geometric filtering, before being passed to the constraint programming component for patrol route optimization.

Results from simulations on the Chicago Crime Dataset confirm that prediction-informed routing policies achieve higher hit rates compared to methods relying on static historical data or random deployments. Future work will focus on extending the model to account for multi-criteria decision-making, incorporating additional data sources (e.g., demographic or environmental factors), and exploring more sophisticated forecasting architectures. Overall, our findings demonstrate the promise of tightly coupling machine learning and constraint programming for proactive, effective crime prevention strategies.

References

1 Christian Bessiere, Luc De Raedt, Tias Guns, Lars Kotthoff, Mirco Nanni, Siegfried Nijssen, Barry O'Sullivan, Anastasia Paparrizou, Dino Pedreschi, and Helmut Simonis. The inductive constraint programming loop. *IEEE Intelligent Systems*, 32(5):44–52, 2017. doi:10.1109/MIS.2017.3711637.

2 Geoff Boeing. Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems*, 65:126–139, 2017. doi:10.1016/j.compenvurbsys.2017.05.004.

3 Kate J Bowers, Shane D Johnson, and Ken Pease. Prospective hot-spotting: the future of crime mapping? *British journal of criminology*, 44(5):641–658, 2004.

4 Eugenio Cesario, Paolo Lindia, and Andrea Vinci. Multi-density crime predictor: an approach to forecast criminal activities in multi-density crime hotspots. *Journal of Big Data*, 11(1):75, 2024. doi:10.1186/S40537-024-00935-4.

5 Spencer Chainey, Lisa Tompson, and Sebastian Uhlig. The utility of hotspot mapping for predicting spatial patterns of crime. *Security journal*, 21:4–28, 2008.

6 Huanfa Chen, T. Cheng, and Sarah Wise. Designing daily patrol routes for policing based on ant colony algorithm. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-4/W2:103–109, July 2015. doi:10.5194/isprsannals-II-4-W2-103-2015.

7 Huanfa Chen, T. Cheng, and Xinyue ye. Designing efficient and balanced police patrol districts on an urban street network. *International Journal of Geographical Information Science*, 33:1–22, October 2018. doi:10.1080/13658816.2018.1525493.

8 Maite Dewinter, Christophe Vandeviver, Tom Vander Beken, and Frank Witlox. Analysing the police patrol routing problem: A review. *ISPRS Int. J. Geo-Information*, 9(3):157, 2020. doi:10.3390/ijgi9030157.

9 Adam N. Elmachtoub and Paul Grigas. Smart "Predict, then optimize". *Manag. Sci.*, 68(1):9–26, 2022. doi:10.1287/MNSC.2020.3922.

10 Xinge Han, Xiaofeng Hu, Huanggang Wu, Bing Shen, and Jiansong Wu. Risk prediction of theft crimes in urban communities: An integrated model of lstm and st-gcn. *IEE ACCESS*, 8:217222–217230, 2020. doi:10.1109/ACCESS.2020.3041924.

11 Timothy Hart and Paul Zandbergen. Kernel density estimation and hotspot mapping: Examining the influence of interpolation method, grid cell size, and bandwidth on crime forecasting. *Policing: An International Journal of Police Strategies & Management*, 37(2):305–323, 2014.

12 Corey D Holmes, Christian Orji, and Chris Papesh. Geospatial temporal crime prediction using convolution and lstm neural networks: Enhancing the las vegas cardiff model. *SMU Data Science Review*, 8(2):3, 2024.

13 Alain Hore and Djemel Ziou. Image quality metrics: PSNR vs. SSIM. In *2010 20th international conference on pattern recognition*, pages 2366–2369. IEEE, 2010.

14 Maja Kalinic and Jukka Matthias Krisp. Kernel density estimation (kde) vs. hot-spot analysis–detecting criminal hot spots in the city of san francisco. In *The 21st AGILE International Conference on Geographic Information Science*, Lund, Sweden, 2018.

15 Jintao Ke, Hongyu Zheng, Hai Yang, and Xiqun Michael Chen. Short-term forecasting of passenger demand under on-demand ride services: A spatio-temporal deep learning approach. *Transportation research part C: Emerging technologies*, 85:591–608, 2017.

16 Dongyeon Kim, Yejin Kan, YooJin Aum, Wanhee Lee, and Gangman Yi. Hotspots-based patrol route optimization algorithm for smart policing. *Heliyon*, 9(10):e20931, 2023. doi:10.1016/j.heliyon.2023.e20931.

17 Yunfeng Kong, Yanfang Zhu, and Yujing Wang. A center-based modeling approach to solve the districting problem. *International Journal of Geographical Information Science*, 33(2):368–384, 2019. doi:10.1080/13658816.2018.1474472.

18 Zhonghang Li, Chao Huang, Lianghao Xia, Yong Xu, and Jian Pei. Spatial-temporal hypergraph self-supervised learning for crime prediction. In *2022 IEEE 38th international conference on data engineering (ICDE)*, pages 2984–2996. IEEE, 2022. doi:10.1109/ICDE53745.2022.00269.

19 Ying-Lung Lin, Meng-Feng Yen, and Liang-Chih Yu. Grid-based crime prediction using geographical features. *ISPRS International Journal of Geo-Information*, 7(8):298, 2018. doi:10.3390/IJGI7080298.

20 He Luo, Peng Zhang, Jiajie Wang, Guoqiang Wang, and Fanhe Meng. Traffic patrolling routing problem with drones in an urban road system. *Sensors*, 19(23):5164, 2019. doi:10.3390/s19235164.

21 X. Lv, Changfeng Jing, Yi Wang, and S. Jin. A deep neural network for spatiotemporal prediction of theft crimes. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLVIII-3/W2-2022:35–41, October 2022. doi:10.5194/isprs-archives-XLVIII-3-W2-2022-35-2022.

22 Jayanta Mandi, Emir Demirovic, Peter J. Stuckey, and Tias Guns. Smart predict-and-optimize for hard combinatorial optimization problems. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*, pages 1603–1610. AAAI Press, 2020. doi:10.1609/AAAI.V34I02.5521.

23 Fatemeh Mousapour, Rajan Batta, and Jose L. Walteros. On the development and analysis of a comprehensive police patrolling model. *EURO Journal on Transportation and Logistics*, 14:100153, 2025. doi:10.1016/j.ejtl.2025.100153.

24 Data Portal. City of chicago. URL: https://data.cityofchicago.org/Public-Safety/Crimes-2023/xguy-4ndq/about_data.

25 Jerry Ratcliffe. *Intelligence-led policing*. Willan Publishing, 2008.

26 Sukanya Samanta, Goutam Sen, and Soumya Kanti Ghosh. A literature review on police patrolling problems. *Annals of Operations Research*, 316(2):1063–1106, 2022. doi:10.1007/S10479-021-04167-0.

27 Yunbo Wang, Zhifeng Gao, Mingsheng Long, Jianmin Wang, and S Yu Philip. Predrnn++: Towards a resolution of the deep-in-time dilemma in spatiotemporal predictive learning. In *International conference on machine learning*, pages 5123–5132. PMLR, 2018.

28 Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004. doi:10.1109/TIP.2003.819861.

29 Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 586–595, 2018. doi:10.1109/CVPR.2018.00068.

30 Di Zhu, Ninghua Wang, Lun Wu, and Yu Liu. Street as a big geo-data assembly and analysis unit in urban studies: A case study using beijing taxi data. *Applied Geography*, 86:152–164, 2017.

Exact Methods for the Travelling Salesperson Problem with Self-Deleting Graphs

Daniel Pekar
Department of Mechanical and Industrial Engineering, University of Toronto, Canada

J. Christopher Beck
Department of Mechanical and Industrial Engineering, University of Toronto, Canada

Abstract

Finding the minimal-cost closed loop on a weighted graph where every vertex is visited exactly once is known as the Travelling Salesperson Problem (TSP). In a recently proposed variant, TSP with Self-Deleting graphs (TSP-SD), visiting a vertex i deletes a set of edges in the graph, preventing their subsequent traversal. Due to the dependency between vertex visits and edge deletion, in TSP-SD the feasibility of a cycle depends on the start node. The best performing solution approaches in the literature rely on a simple problem reformulation to find a backward tour where vertex visits add edges rather than delete them. This paper investigates exact model-based approaches, specifically Constraint Programming (CP), Domain-Independent Dynamic Programming (DIDP), and Mixed Integer Linear Programming (MIP) to solve TSP-SD. We show that simple preprocessing can substantially reduce the options for start/end vertex pairs but typically has a limited positive impact on search performance. Our numerical results demonstrate that the difference between the deletion and addition variants is small for CP and MIP but that the reformulation is critical for DIDP performance. Overall, the DIDP addition model is the best of the exact methods on all test instances and outperforms existing heuristic solvers for small and medium-sized instances while trailing in terms of solution quality on larger instances.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Mathematics of computing → Combinatorial optimization

Keywords and phrases Decision Diagrams & Dynamic Programming, Operations Research & Mathematical Optimization, Modelling & Modelling Languages

Digital Object Identifier 10.4230/LIPIcs.CP.2025.30

Supplementary Material *Software (Source Code)*: `https://github.com/uoft-tidel/tsp-sd` [18] archived at `swh:1:dir:c287b1067c276693ca630e6e6f7a18daf1a90d38`

Funding This work was supported the Natural Sciences and Engineering Research Council of Canada.

Acknowledgements Computations were performed on the Niagara supercomputer at the SciNet HPC Consortium. SciNet is funded by ISED Canada; the Digital Research Alliance of Canada; Ontario Research Fund:RE; and the University of Toronto.

1 Introduction

The Travelling Salesperson Problem (TSP) is an NP-hard problem that has been extensively studied since it was first formulated by Hamilton in the 19th century [9]. Variations of the TSP with path or time dependencies, which alter the properties of the graph during its traversal, have also been studied with time-dependent TSP or TSP with time windows being the most common [6, 7, 17]. Carmesin et al. [2] introduced TSP with self-deleting graphs (TSP-SD) where subsets of edges are rendered unavailable after corresponding vertices are visited. TSP-SD has applications in mining, where visits correspond to excavation operations that make traversal of some areas unsafe as well as in driving pile foundations with the related problem of TSP with circle placement [11, 15, 22]. We are interested in TSP-SD as

© Daniel Pekar and J. Christopher Beck;
licensed under Creative Commons License CC-BY 4.0

a simple example of a state-dependent problem [8, 14], where problem components, such as costs, can change depending on the decisions that have already been made. With the goal of exploring solution approaches to such problems, this paper compares state-based and constraint-based approaches in terms of ease of modelling and problem solving performance.

Carmesin et al. showed that TSP-SD could be solved by a depth-first search (DFS) that constructs the sequence in reverse order by iteratively adding vertices to the beginning of the tour; vertex visits, therefore, add edges to the graph rather than deleting them. For the DFS, a metaheuristic seeded by a DFS solution, and a later GRASP approach, this backward/addition approach was shown to perform significantly better than searching for forward sequences where visits delete edges [2, 22].

We approach TSP-SD using exact solvers in a model-and-solve paradigm. To that end, eight distinct models are created: addition and deletion variants for two constraint programming (CP) models, one domain-independent dynamic programming (DIDP) model, and one mixed integer programming (MIP) model, respectively. We further observe that the final edge in a tour (i.e., the one that closes the loop from the last vertex visited to the first) must be an edge that is not deleted by any visit. As a consequence, preprocessing can identify a restricted set of first/last pairs that can be easily specified and experimented with in model-based approaches.

Using problem instances from the literature [2], we compare our eight models, with and without first/last vertex restrictions, with existing approaches. We demonstrate that by a significant margin, the DIDP addition model performs best among all the exact approaches and outperforms previous heuristic approaches on small and medium-sized instances while trailing the previous work on larger problems. The addition variants typically exhibit small reductions in measures of search effort compared to the deletion variants with the exception of DIDP where the benefit of the addition model is substantial. The first/last vertex restrictions generally improve performance for finding an initial solution and reducing primal and optimality gaps. However, the addition variants in one CP model and the MIP model performed better without the imposition of first/last vertex restrictions.

This paper is structured as follows. In Section 2, we formally define the problem and summarize related work. Section 3 observes that the first and last vertex pairs in the tour can be restricted through simple preprocessing. In Section 4, we present four exact models each with addition and deletion variants: a CP model based on the ranks of visits, a CP model based on a scheduling perspective, a DIDP model, and a MIP model. Section 5 details the experimentation and our analysis of the results. Finally, Section 6 concludes.

2 Background

We first introduce and define the TSP-SD to give a clearer picture of how it relates to other problems explored in the literature.

2.1 Problem Definition

Given a complete, undirected graph, $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, the TSP with self-deleting graphs (TSP-SD) problem is to find a Hamiltonian tour of minimum length such that no edge is traversed after a visit to a vertex that deletes it [2]. Formally, let c_{ij} be the length of the edge between vertices i and j and let $\mathbf{D}_i$ be the set of edges that are deleted when vertex i is visited. If z_{ij} is a binary variable equal to 1 if edge $\{i, j\}$ is traversed in the tour, then the objective to minimize $\sum_{i,j \in V} c_{ij} z_{ij}$ subject to constraints that ensure a Hamiltonian tour and that edges are not deleted before their use. Carmesin et al. [2] show that TSP-SD is NP-hard. Note

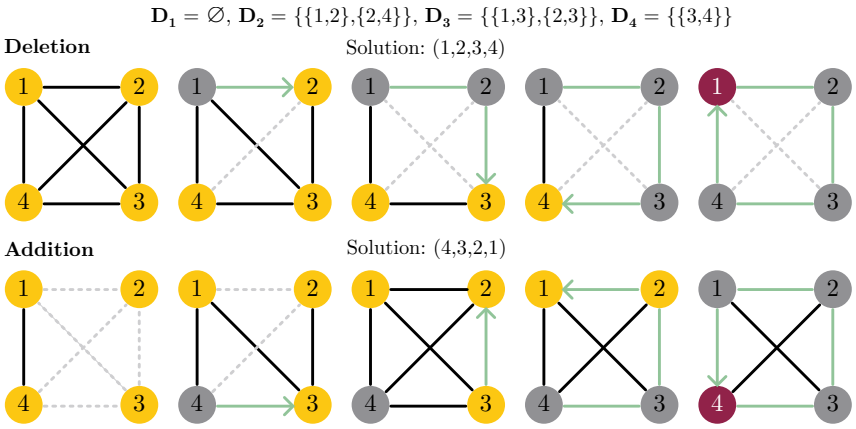


Figure 1 An example of deletion and addition variants of TSP-SD.

that an edge may or may not be incident to a vertex that deletes it, a given edge may be deleted by more than one vertex visit, and that, unlike standard TSP, the vertex at which the tour starts can affect the tour's feasibility. We call this definition of the problem the *deletion* variant.

The top row of Figure 1 shows a four-vertex example. Starting from the complete graph, vertex 1 is chosen as the start node and no edges are deleted as $\mathbf{D}_1 = \emptyset$. Then, edge $\{1, 2\}$ is traversed to vertex 2 and the edges in $\mathbf{D}_2$ are deleted. The deletion of $\{1, 2\}$ has no effect on the tour as that edge has already been traversed. However, now edge $\{2, 4\}$ can no longer be traversed. The sequence continues to vertices 3 and 4 with corresponding deletions of edges in $\mathbf{D}_3$ and $\mathbf{D}_4$ before edge $\{4, 1\}$ is used to return to the start vertex. Of course, finding such a solution without backtracking is not guaranteed.

2.2 Literature Review

Carmesin et al. [2] proposed a depth-first search (DFS) algorithm to solve the TSP-SD. The algorithm begins by choosing a start vertex and then builds the vertex sequence by choosing the next one to visit, while ensuring that the intervening edge has not yet been deleted. When the final vertex is selected, the algorithm ensures that the edge connecting it with the start vertex has not been deleted to ensure that the tour can be completed.

Carmesin et al. also defined a DFS algorithm that constructs the sequence backward from the final vertex to the start vertex. Starting with a graph only containing edges that are not deleted by any vertex visit (i.e., edges $\{j, k\}$ s.t. $\nexists i, \{j, k\} \in \mathbf{D}_i$), a vertex is inserted at the beginning of the partial sequence, while ensuring that the intervening edge exists. Under this regime, a visit to vertex i *adds* the edges $\mathbf{D}_i$ to the graph. If an edge appears in multiple deletion sets, then all corresponding vertex visits must be present in the partial sequence before the edge can be used to add a vertex to the beginning of the partial sequence. When the starting vertex is selected (i.e., in the last step of the DFS), it is not sufficient for the edge connecting it to the final node to be present in the current graph because in the forward sequence the edge connecting the first and last vertices is the last edge to be traversed. Thus, to close the tour, the last edge must exist after all vertices have been visited (i.e., it must be an edge that no vertex deletes). The DFS algorithm performs this extra check when adding the starting vertex and backtracks if it is not satisfied. Carmesin et al. prove the correctness of their backward DFS. We refer to this approach to solving as the *addition* variant.

The second row of Figure 1 illustrates the backward DFS algorithm. Vertex 4 is selected as the last vertex and the edges in $\mathbf{D}_4$ are added to the graph. Vertices 3, 2, and 1 are then iteratively added to the beginning of the sequence traversing edges that exist at the time of the traversal. As noted, the DFS checks that the edge $\{1, 4\}$ existed in the starting graph to ensure that the forward tour can be completed.

Carmesin et al. show, experimentally, that the backward DFS outperforms the forward DFS in finding feasible solutions and proving infeasibility, requiring significantly fewer search nodes and less time. Further work [2, 22], proposed a metaheuristic, warm-started with a solution found by a time-limited backward DFS, and a Greedy Randomized Adaptive Search Procedure (GRASP) [5]. The metaheuristic improved on the backward DFS and GRASP further improved performance, finding best and mean solutions up to 16% and 6.7% better, respectively, than the warm-started metaheuristic on large instances.

While TSP-SD shares similarities with other time- or path-dependent routing problems such as the time-dependent TSP (TDTSP) [19], the closest work is by Lipovetzky et al. [15]. That work investigates AI planning for a mining application where parts of the mine are only physically accessible after previous blocks have been excavated. These physical considerations impose partial orderings of operations, similar to TSP-SD, but whereas the mining operations add subsequent travel paths, vertex visits in TSP-SD remove them.

3 First/Last Vertex Constraints

Before presenting our exact models of TSP-SD, we make the following observation.

▶ **Observation 1.** *Since all vertices are visited upon completion of a Hamiltonian path, in any solution, all edges that could be deleted are deleted. As such, the set of undeleted edges at the end of the tour,* $\boldsymbol{E}_{remain}$*, can be defined as:* $\boldsymbol{E}_{remain} = \boldsymbol{E} \setminus \bigcup_{i=1}^{n} \boldsymbol{D}_i$.

Since the tour must return to the start vertex, the returning edge must be an edge in $\boldsymbol{E}_{remain}$*, and therefore the start and end vertices of the tour must be incident to the remaining edges. From this we can define* $\boldsymbol{V}_{remain}$ *as:* $\boldsymbol{V}_{remain} = \bigcup\{i, j\}, \forall\{i, j\} \in E_{remain}$.

While the existing DFS approaches correctly ensure that the edge between the first and last vertices in the tour is not deleted, they do not exploit this observation. No restrictions are made on the choice of the start vertex nor is any reasoning done about partial tours with remaining return edges to the first vertex. We implement these first/last restrictions in our models and evaluate their impact on problem solving.

4 TSP-SD Models

Given the performance differences in the literature between the deletion and addition variants and their embedding in a DFS search, we are interested in understanding if such differences are manifest in model-based approaches. Thus, we formulated four distinct models (two CP models, one DIDP model, and one MIP model) each with a deletion and addition variant. Further, we investigate Observation 1 by testing each of the eight models with and without first/last restrictions. The notation we adopt is summarized in Table 1.

4.1 Constraint Programming

We present two CP models: a rank-based model, where the main decision variable is the position of each vertex in the tour, and a scheduling-based model, where we use optional interval variables to represent the sequence of edge traversals.

Table 1 Notation used to define TSP-SD.

Symbol	Explanation
c_{ij}	Distance from vertex i to vertex j
$\mathbf{V}$	Set of all vertices $\{1, \ldots, n\}$
$\mathbf{E}$	Set of all edges
$\mathbf{D}_i$	Set of edges deleted after visiting vertex i
$\mathbf{D}_{ij}$	Set of vertices which delete the edge $\{i, j\}$
$\mathbf{E}_{remain}$	Set of remaining edges after all vertices are visited
$\mathbf{V}_{remain}$	Set of vertices along remaining edges

4.1.1 CP Rank Model

The CP Rank model is premised on the idea that each vertex must be assigned a unique rank (i.e., position in the sequence) and each rank must be associated with a unique vertex. Let $x_i \in X$ be the vertex visited at rank i in the tour and let $y_j \in Y$ be the rank of vertex j in the tour. The domains of both variables are $\{1, ..., n\}$ for n vertices.

The *CP Rank Del* model is defined in Figure 2. The objective function seeks to minimize the sum of costs between vertices in consecutive ranks, with the modulo function accounting for the return edge. Constraint (1b) ensures that $x_{y_i} = y_{x_i}$. Each vertex is visited exactly once so the values of both X and Y must form permutations as enforced with constraints (1c) and (1d). Constraint (1e) expresses the deletion behavior by constraining the ordering of vertex visits and edge traversals. For every deleted edge, $\{j, k\} \in \mathbf{D}_i$, either the two vertices j, k are not adjacent in the sequence or visits to both vertices j and k must be before the visit to i. Note that the ranks of vertex i and either vertex j or k may be equal as it is possible to traverse an edge to a vertex that deletes it. That is, i may be equal to j or k. Constraint (1f) ensures that the edge between the first and last vertices is not deleted by any vertex visit.

We can modify CP Rank Del to represent the addition variant of the problem simply by reversing the ordering in constraint (1e): an edge $\{j, k\}$ can be traversed only after visiting every vertex i in the set $\mathbf{D}_{jk} = \{i : \{j, k\} \in \mathbf{D}_i\}$. Thus, constraint (1e) is replaced with constraint (3).

$$(|y_j - y_k| \neq 1) \vee (y_j \geq y_i \wedge y_k \geq y_i) \quad \forall \{j, k\} \in \mathbf{D}_i \quad \forall i \in \mathbf{V} \tag{3}$$

$$\min \sum_{i \in V} c_{i x_{(y_i \bmod n)+1}} \tag{1a}$$

$$\text{s.t. } \textsc{Inverse}(X, Y) \tag{1b}$$

$$\textsc{AllDifferent}(X) \tag{1c}$$

$$\textsc{AllDifferent}(Y) \tag{1d}$$

$$(|y_j - y_k| \neq 1) \vee (y_j \leq y_i \wedge y_k \leq y_i) \quad \forall \{j, k\} \in \mathbf{D}_i \quad \forall i \in \mathbf{V} \tag{1e}$$

$$|y_j - y_k| \neq n - 1 \quad \forall \{j, k\} \in \mathbf{D}_i \quad \forall i \in \mathbf{V} \tag{1f}$$

$$\text{Integer variable} \quad x_i = \{1, \ldots, n\} \quad \forall i \in \mathbf{V} \tag{1g}$$

$$\text{Integer variable} \quad y_j = \{1, \ldots, n\} \quad \forall i \in \mathbf{V} \tag{1h}$$

Figure 2 The CP Rank Del model.

$$\min \sum_{i \in V} c_{i x_{(y_i \bmod n)+1}} \tag{2a}$$

s.t. $(1b), (1c), (1d), (1e)$

$$\textsc{AllowedAssignments}(\{x_1, x_n\}, \{\{i,j\} \in \mathbf{E}_{remain}\}) \tag{2b}$$

$$\text{Integer variable} \quad x_i = \begin{cases} \{1,\ldots,n\} & \forall \quad i \in \{2,\ldots,n-1\} \\ \mathbf{V}_{remain} & \forall \quad i \in \{1,n\} \end{cases} \tag{2c}$$

$$\text{Integer variable} \quad y_j = \begin{cases} \{2,\ldots,n-1\} & \forall \quad j \notin \mathbf{V}_{remain} \\ \{1,\ldots,n\} & \forall \quad j \in \mathbf{V}_{remain} \end{cases} \tag{2d}$$

Figure 3 The CP Rank Del model with first/last vertex restrictions.

We call this second formulation *CP Rank Add*. The solutions to CP Rank Add is a solution to CP Rank Del with each x_i, y_j in the addition variant being equal to x_{n-i+1}, y_{n-i+1} in the deletion variant.

We can include Observation 1 in the CP Rank models by directly constraining the domains of the corresponding variables in X and Y as in the model presented in Figure 3. We replace constraint (1f) with a table constraint specifying the possible return edges and restrict the domains of the variables x_i and y_j in constraints (2c) and (2d), respectively.

4.1.2 CP Interval Model

Motivated by the success of CP scheduling constructs both in scheduling and non-scheduling problems [10, 16], we present the intuition of the CP Interval model in Figure 4. In the model, each edge $\{i,j\}$ is represented as an optional interval variable, $traverse_{ij}$, with length c_{ij}. As each vertex i must have exactly one edge entering and one edge exiting it in a tour, we create two interval variables, in_i and out_i, and employ an ALTERNATIVE constraint to ensure that in_i is set to the chosen $traverse_{ji}$ variable that enters vertex i and out_i is set to the chosen $traverse_{ij}$ variable that exits vertex i. Each $traverse_{ij}$ variable appears in two ALTERNATIVE constraints, one related to out_i and one for in_j, and thus when the solver sets $traverse_{ij}$ as present, out_i will correctly equal in_j. We further constrain out_i to start at the end of in_i as visits are instantaneous. Finally, though not illustrated in Figure 4, the in_i variables are sequenced to form a permutation.

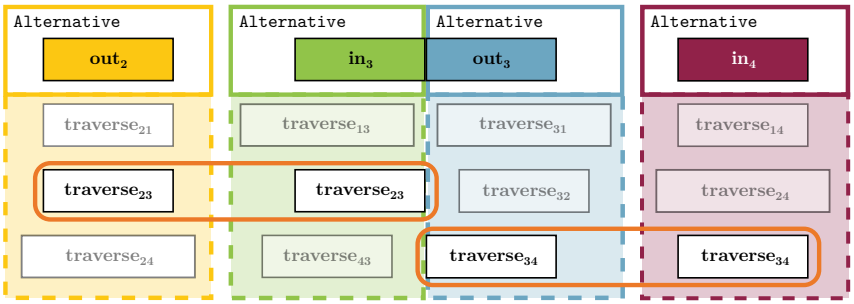


Figure 4 An illustration of the interval variables in the CP Interval model. The traversal of an edge $\{i,j\}$ is represented by an optional interval variable $traverse_{ij}$ that occurs in the ALTERNATIVE constraints with the out_i and in_j interval variables.

$$\min \textsc{EndOf}(in_{n+1}) \tag{4a}$$
$$\text{s.t. } \textsc{NoOverlap}(\text{INS}) \tag{4b}$$
$$\textsc{Last}(\text{INS}, in_{n+1}) \tag{4c}$$
$$\textsc{StartAtEnd}(out_i, in_i) \quad \forall i \in \mathbf{V} \tag{4d}$$
$$\textsc{Alternative}(in_i, traverse_{ji} \forall j \in \mathbf{V}) \quad \forall i \in \mathbf{V}) \tag{4e}$$
$$\textsc{Alternative}(out_i, traverse_{ij} \cup traverse_last_{ij} \forall j \in \mathbf{V}) \quad \forall i \in \mathbf{V} \setminus \{0\}) \tag{4f}$$
$$\textsc{Alternative}(in_{n+1}, traverse_last_{ij}) \quad \forall \{i,j\} \in \mathbf{E} \tag{4g}$$
$$\textsc{IsPresent}(traverse_last_{ij}) \leftrightarrow (\textsc{IsPresent}(traverse_{0i}) \wedge \textsc{IsPresent}(traverse_{j,n+1})) \quad \forall \{i,j\} \in \mathbf{E} \tag{4h}$$
$$\textsc{EndBeforeEnd}(traverse_{jk}, in_i \quad \forall \{j,k\} \in \mathbf{D}_i) \quad \forall i \in \mathbf{V} \tag{4i}$$
$$\neg\textsc{IsPresent}(traverse_last_{jk}) \quad \forall \{j,k\} \in \mathbf{D}_i, i \in \mathbf{V} \tag{4j}$$
$$\text{Optional interval variable} \quad traverse_{ij} \quad size = c_{ij} \quad \forall \{i,j\} \in \mathbf{E} \tag{4k}$$
$$\text{Optional interval variable} \quad traverse_{0i} \quad size = 0 \quad \forall i \in \mathbf{V} \tag{4l}$$
$$\text{Optional interval variable} \quad traverse_last_{ij} \quad size = c_{ij} \quad \forall \{i,j\} \in \mathbf{E} \tag{4m}$$
$$\text{Interval variable} \quad in_i \quad \forall i \in \mathbf{V} \cup \{n+1\} \tag{4n}$$
$$\text{Interval variable} \quad out_i \quad \forall i \in \mathbf{V} \cup \{0\} \tag{4o}$$
$$\text{Sequence variable INS} \quad \{in_i \quad \forall i \in \mathbf{V} \cup \{n+1\}\} \tag{4p}$$

Figure 5 The CP Interval Del model.

Figure 5 presents the formal definition of *CP Interval Del*. In addition to the interval variables described above, we introduce two dummy vertices, with indices $i = 0$ and $i = n+1$, as the start and end points in the permutation with corresponding interval variables out_0, in_{n+1}, $traverse_{0j}$, and $traverse_{j,n+1}$. The return edges are represented by another interval variable, $traverse_last_{ij}$, that are alternative realizations of in_{n+1}.

We seek to minimize the end time in_{n+1} in objective (4a). Constraints (4b) and (4c) constrain all in_i variables to form a permutation with in_{n+1} coming last. Constraint (4d) ensures that the in_i and out_i variables are sequenced contiguously. The ALTERNATIVE constraints (4e)-(4g) represent the logic described in Figure 4 extended to include the return edge. Constraint (4h) defines the return edge to be between the vertices visited immediately after the dummy start vertex and immediately before the dummy end vertex. The self-deletion requirement is modeled by sequencing of intervals out_i and intervals $traverse_{jk}$ for $\{j,k\} \in \mathbf{D}_i$. Finally, constraint (4j) prevents a deleted edge from being the return edge.

To define the *CP Interval Add* model, the deletion constraint (4i) is replaced with (5), such that if $traverse_{jk}$ is present, it must occur after the visit to the vertex that adds it.

$$\textsc{StartBeforeStart}(out_i, traverse_{jk}) \quad \forall \{j,k\} \in \mathbf{D}_i \quad \forall i \in \mathbf{V} \tag{5}$$

To incorporate first/last vertex restrictions, we adjust the domains of the $traverse_{0i}$ and $traverse_last_{ij}$ variables in Figure 6. The domain of $traverse_{0i}$ is restricted $\mathbf{V}_{remain}$ (constraint (6f)), while the domain of $traverse_last_{ij}$ is restricted to $\mathbf{E}_{remain}$ (constraint (6g)). The same domain restrictions are similarly replicated in constraints (6b), (6c), and (6d). We also replace constraint (4h) in the CP Interval Del model with constraint (6e) to link the first traverse variable with the corresponding $traverse_last_{ji}$ variables.

$$\min \textsc{EndOf}(in_{n+1}) \tag{6a}$$

s.t.

$$(4b), (4c), (4d), (4e), (4i)$$

$$\textsc{Alternative}(out_i, traverse_{ij} \cup traverse_last_{ik} \forall j \in \mathbf{V} \quad k \in \mathbf{V}_{remain} : \{i,k\} \in E_{remain}) \qquad \forall i \in \mathbf{V} \setminus \{0\}) \tag{6b}$$

$$\textsc{Alternative}(in_{n+1}, traverse_last_{ij}) \qquad \forall \{i,j\} \in \mathbf{E}_{remain} \tag{6c}$$

$$\textsc{Alternative}(out_0, traverse_{0i}) \qquad \forall i \in \mathbf{V}_{remain} \tag{6d}$$

$$\textsc{IsPresent}(traverse_{0i}) = \sum_{(j,i) \in E_{remain}} \textsc{IsPresent}(traverse_last_{ji}) \qquad \forall i \in \mathbf{V}_{remain} \tag{6e}$$

$$\text{Optional interval variable} \quad traverse_{0i} \quad size = 0 \qquad \forall i \in \mathbf{V}_{remain} \tag{6f}$$

$$\text{Optional interval variable} \quad traverse_last_{ij} \quad size = c_{ij} \qquad \forall \{i,j\} \in \mathbf{E}_{remain} \tag{6g}$$

Figure 6 The CP Interval Del model with first/last vertex restrictions.

4.2 Domain-Independent Dynamic Programming

DIDP is a declarative model-based paradigm for combinatorial optimization based on dynamic programming [12]. Our TSP-SD model builds on existing formulations for TSPTW [4, 13] and assembly line balancing problem with sequence-dependent setup times [23]. The deletion variant, *DIDP Del*, is shown in Figure 7.

The state variables are U, the set of unvisited vertices, i, the current vertex, and f, the first vertex visited in the tour. As in the CP Interval model we let vertex 0 represent a dummy start node.

We aim to compute the value function of target state $\langle N \setminus \{0\}, 0, 0 \rangle$ (term (7a)). Equation (7b) recursively computes the cost of the state as we traverse each edge. When $i = 0$, the transition to the start vertex is selected. Subsequently, while the set of unvisited vertices is not a singleton, the next vertex is chosen ensuring that the intervening edge has not been deleted. This condition is expressed as $D_{ij} \subseteq U$, as all vertices that delete edge $\{i,j\}$ must still be unvisited. The selection of the final vertex is a special case, captured in the next condition that requires that the return edge has not been deleted: $D_{jf} \subseteq U$. The base case of $U = \emptyset$ has a cost of 0. If none of these conditions are satisfied, the state is a deadend and so is assigned the cost of ∞. Following the TSPTW model [12], we specify two dual bounds, inequalities (7c) and (7d), based on the minimum cost of all incoming edges from or all outgoing edges to the unvisited vertices.

To create the addition variation, *DIDP Add*, the Bellman equation is replaced by (8). In the second and third cases, rather than ensuring the traversed edge has not yet been deleted, we require that all adding vertices must be already visited: $D_{ij} \cap U = \emptyset$.

Finally, to incorporate first/last vertex restrictions into the DIDP Del model, the possible choices for the first vertex are restricted to only those in set $\mathbf{V}_{remain}$. Similarly, the possible choices for the last vertex are restricted to only those such that the edge $\{j,f\}$ is in the set of remaining edges $\mathbf{E}_{remain}$. The Bellman equation is shown in Figure 9.

$$\text{compute } \mathcal{V}(N \setminus \{0\}, 0, 0) \tag{7a}$$

$$V(U,i,f) = \begin{cases} \min_{j \in V} V(U \setminus \{j\}, j, j) & \text{if } i = 0 \\ \min_{j \in U | D_{ij} \subseteq U} c_{ij} + V(U \setminus \{j\}, j, f) & \text{else if } |U| > 1 \wedge (\exists j \in U \mid D_{ij} \subseteq U) \\ \min_{j \in U} c_{ij} + c_{jf} + V(U \setminus \{j\}, j, f) & \text{else if } |U| = 1 \wedge \\ & \quad (\exists j \in U \mid D_{fj} = \emptyset \wedge D_{ij} \subseteq U) \\ 0 & \text{else if } U = \emptyset \\ \infty & \text{else} \end{cases} \tag{7b}$$

$$V(U,i,f) \geq \sum_{j \in U \setminus \{i\}} \min_{k \in N \setminus \{j\}} c_{kj} \tag{7c}$$

$$V(U,i,f) \geq \sum_{j \in U \setminus \{f\}} \min_{k \in N \setminus \{j\}} c_{jk} \tag{7d}$$

Figure 7 The DIDP Del model.

$$V(U,i,f) = \begin{cases} \min_{j \in V} V(U \setminus \{j\}, j, j) & \text{if } i = 0 \\ \min_{j \in U | D_{ij} \cap U = \emptyset} c_{ij} + V(U \setminus \{j\}, j, f) & \text{else if } |U| > 1 \wedge \\ & \quad (\exists j \in U \mid D_{ij} \cap U = \emptyset) \\ \min_{j \in U} c_{ij} + c_{jf} + V(U \setminus \{j\}, j, f) & \text{else if } |U| = 1 \wedge \\ & \quad (\exists j \in U \mid D_{fj} = \emptyset \wedge D_{ij} \cap U = \emptyset) \\ 0 & \text{else if } U = \emptyset \\ \infty & \text{else} \end{cases} \tag{8}$$

Figure 8 The Bellman equation in the DIDP Add model.

4.3 Mixed Integer Programming

We develop a MIP model similar to the one for Time Dependent TSP [17]. The primary difference, and an advantage in TSP-SD, is that the variables are indexed by rank as opposed to time slot and thus do not scale with the time horizon, only with the number of vertices.

Let binary decision variable x_{ijr} represent the traversal of edge $\{i, j\}$ at rank r. As such, the number of binary decision variables for a given graph is $O(n^3)$. To account for the cost of returning to the starting vertex, we introduce the binary variable y_{ij} that is 1 if $\{i, j\}$ is the return edge.

Figure 10 shows the *MIP Del* model. The objective is to minimize the total distance traveled on all intermediate edges plus the return edge. Constraints (10b) and (10c) ensure that each vertex is entered once and each rank is chosen once. The proper rank of edges is maintained via constraint (10d). The next three constraints specify that the return edge cannot be deleted (10e), that it must connect vertices in the first and last rank (10f), and that there can be only one return edge (10g). Finally, constraint (10h) ensures that the rank of any deleted edge must be less than or equal to any vertex that deletes it.

$$V(U,i,f) = \begin{cases} \min\limits_{j \in V_{remain}} V(U \setminus \{j\}, j, j) & \text{if } i = 0 \\ \min\limits_{j \in U \mid D_{ij} \subseteq U} c_{ij} + V(U \setminus \{j\}, j, f) & \text{else if } |U| > 1 \wedge (\exists j \in U \mid D_{ij} \subseteq U) \\ \min\limits_{j \in U} c_{ij} + c_{jf} + V(U \setminus \{j\}, j, f) & \text{else if } |U| = 1 \wedge \\ & \quad (\exists j \in U \mid (j,f) \in \mathbf{E}_{remain} \wedge D_{ij} \subseteq U) \\ 0 & \text{else if } U = \emptyset \\ \infty & \text{else} \end{cases} \tag{9}$$

Figure 9 The Bellman equation for the DIDP Del model with first/last vertex restrictions.

For the addition variant, *MIP Add*, constraint (10h) is replaced with constraint (11) to ensure that either an added edge is ranked after all adding vertices or the edge is not used.

$$\sum_{r \in \mathbf{V}} r x_{klr} + M(1 - \sum_{r \in \mathbf{V}} x_{klr}) \geq 1 + \sum_{r \in \mathbf{V}} \sum_{j \in \mathbf{V}} r x_{jir} \quad \forall \{k,l\} \in \mathbf{D}_i \quad \forall i \in V \tag{11}$$

We can adjust the MIP Del model to include first/last vertex restrictions by modifying the constraints that affect the choice of first and last vertex and the scope of variable y_{ij} as shown in Figure 11. Constraints (12b) and (12c) are added to restrict the first and last vertices to only those in the set $\mathbf{V}_{remain}$. We ensure that the first and last vertices are connected by an edge in $\mathbf{E}_{remain}$ using constraint (12d). Finally, constraints (12e) and (12f) are analogous to (10f) and (10g), only restricted to the set of possible edges in $\mathbf{E}_{remain}$.

5 Numerical Experiments

The aims of our numerical experiments are to evaluate the performance of the exact models, to investigate whether the deletion or addition variants perform differently, to assess the impact of the first/last restrictions, and finally to compare the best-performing model-based techniques to the existing custom approaches. We address the first three aims in Experiment 1 and the final one in Experiment 2.[1]

Recall that Carmesin et al. [2] found a substantial computational advantage in adopting the addition variant within a depth-first search that built the sequence backward. For our CP and MIP models, the difference between the addition and deletion variants is simply the direction of a set of sequencing constraints. Further, as the corresponding solvers do not necessarily build the sequence in order (either backward or forward), we expect limited performance differences between the variants. In contrast, the DIDP models are solved by adding vertices to the start or end of a partial sequence. Thus, we expect to see a similar effect as observed in the previous work with the addition variant performing better than deletion. As the first/last restrictions limit the search space, we expect to see their inclusion will improve performance of all of our models.

[1] GitHub: `https://github.com/uoft-tidel/tsp-sd`
Instance Data and Results: `https://tidel.mie.utoronto.ca/external/Pekar_CP2025/extra.php`

$$\min \sum_{r \in \mathbf{V}} \sum_{\{i,j\} \in \mathbf{E}} c_{ij} x_{ijr} + \sum_{\{i,j\} \in \mathbf{E}} c_{ij} y_{ij} \tag{10a}$$

$$\text{s.t.} \sum_{i \in \mathbf{V}} \sum_{r \in \mathbf{V}} x_{ijr} = 1 \qquad \forall j \in \mathbf{V} \tag{10b}$$

$$\sum_{\{i,j\} \in \mathbf{E}} x_{ijr} = 1 \qquad \forall r \in \mathbf{V} \tag{10c}$$

$$\sum_{j \in \mathbf{V}} x_{ijr} = \sum_{j \in \mathbf{V}} x_{j,i,r-1} \qquad \forall i \in \mathbf{V}, r \in \mathbf{V} \tag{10d}$$

$$y_{jk} = 0 \qquad \forall \{j,k\} \in \mathbf{D}_i, i \in \mathbf{V} \tag{10e}$$

$$x_{0i0} + \sum_{k \in \mathbf{V}} x_{kjn} \leq y_{ij} + 1 \qquad \forall \{i,j\} \in \mathbf{E} \tag{10f}$$

$$\sum_{(i,j) \in \mathbf{E}} y_{ij} = 1 \tag{10g}$$

$$\sum_{r \in \mathbf{V}} r x_{klr} \leq \sum_{r \in \mathbf{V}} \sum_{j \in \mathbf{V}} r x_{jir} \qquad \forall \{k,l\} \in \mathbf{D}_i, i \in \mathbf{V} \tag{10h}$$

$$x_{ijr} = \begin{cases} 1 & \text{if traversing edge } \{i,j\} \text{ at rank } r \\ 0 & \text{else} \end{cases} \qquad \forall i,j \in \mathbf{V}, r \in \mathbf{V} \tag{10i}$$

$$y_{ij} = \begin{cases} 1 & \text{if traversing edge } \{i,j\} \text{ last} \\ 0 & \text{else} \end{cases} \qquad \forall \{i,j\} \in \mathbf{E} \tag{10j}$$

Figure 10 The MIP Del model.

5.1 Experiment 1: Comparison of Exact Models

Problem Set. The exact models were compared using a dataset of 60 instances chosen from an existing set of 30,000 randomly generated problems with size $n = [10, 20, \ldots, 100, 150, 200]$ and 50 instances per size [21]. For each size, a deletion function was randomly generated with differing probabilities to produce instances with varying expected vertex density over the course of the tour (see Carmesin et al. [2] for the exact definition of density used). For each n, we chose the first instance at each quintile of average vertex degree (0%, 25%, 50%, 75%, 100%), producing 60 instances in total with diverse sizes and densities.

We did not filter the problem instances for feasibility. A posteriori, at least one of our exact methods found a feasible solution or proved infeasibility for each instance showing that the set consists of 39 feasible and 21 infeasible instances.

Experimental Set-up. Each model is run with a single-thread with a 30-minute time-out and an 8 GB memory limit for each instance. All model-based approaches are run on a dedicated Linux server, with Intel(R) Xeon(R) Gold 6148 CPUs running at 2.4 GHz. The CP models were run using CP Optimizer 22.1.1.0 via IBM DOcplex, DIDP was run using the CABS solver via DIDPPy v0.8.0, and the MIP models were solved by Gurobi 12.0.0.

Model Comparison. An overview of the results can be seen in Figures 12 and 13 with detailed results in Appendix A.

$$\min \sum_{r \in \mathbf{V}} \sum_{\{i,j\} \in \mathbf{E}} c_{ij} x_{ijr} + \sum_{\{i,j\} \in \mathbf{E}_{remain}} c_{ij} y_{ij} \tag{12a}$$

s.t. $(10b), (10c), (10d), (10h)$

$$\sum_{i \in \mathbf{V}_{remain}} x_{0i0} = 1 \tag{12b}$$

$$\sum_{i \in \mathbf{V}} \sum_{j \in \mathbf{V}_{remain}} x_{ijn} = 1 \tag{12c}$$

$$x_{0i0} \leq \sum_{j \in \mathbf{V}_{remain} | (j,i) \in \mathbf{E}_{remain}} \sum_{k \in \mathbf{V}} x_{kjn} \qquad \forall i \in \mathbf{V}_{remain} \tag{12d}$$

$$x_{0i0} + \sum_{k \in \mathbf{V}} x_{kjn} \leq y_{ij} + 1 \qquad \forall \{i,j\} \in \mathbf{E}_{remain} \tag{12e}$$

$$\sum_{\{i,j\} \in \mathbf{E}_{remain}} y_{ij} = 1 \tag{12f}$$

$$x_{ijr} = \begin{cases} 1 & \text{if traversing edge } \{i,j\} \text{ at rank } r \\ 0 & \text{else} \end{cases} \qquad \forall i,j \in \mathbf{V}, r \in \mathbf{V} \tag{12g}$$

$$y_{ij} = \begin{cases} 1 & \text{if traversing edge } \{i,j\} \text{ last} \\ 0 & \text{else} \end{cases} \qquad \forall \{i,j\} \in \mathbf{E}_{remain} \tag{12h}$$

Figure 11 The MIP Del model with first/last vertex restrictions.

The DIDP Add model and both CP Rank models outperform the others, with DIDP Add having the overall advantage. DIDP Add proved optimality or infeasibility on the largest number of instances, produced substantially better solutions, and achieved the best optimality gap (see Table 5). The CP Rank models typically achieve second and third place on these measures. The MIP models are next in terms of proofs of optimality/infeasibility but trail the CP Interval models in terms of primal gap. This poor MIP performance is due to the solver running out of memory for some instances with $n \geq 100$, while none of the other solvers exhibited memory issues. Finally, DIDP Del is the worst performing model in terms of proofs but outperforms the MIP models based on the quality of primal solutions.

Deletion vs. Addition Variants. As expected, the DIDP Add model is substantially better than DIDP Del model: using the deletion variant takes DIDP from the best performing exact approach to the worst. Consistent with Carmesin et al.'s DFS, Figure 13 shows that the ability to quickly find high quality solutions in DIDP Add is substantially impaired in the DIDP Del model. Similarly following expectations, the MIP and CP Rank models exhibit minor differences in performance between their deletion and addition variants in Figures 12 and 13. Interestingly, CP Interval Add performs substantially better than CP Interval Del, with the quick improvement in primal solution quality again likely a key factor. We speculate that in solving a scheduling model, CP Optimizer may be employing primal heuristics that build the solution chronologically.

Table 2 provides more detailed data. For a given approach (e.g., CP Rank), we identified the problem instances for which both the deletion and addition variants proved optimality or infeasibility. We provide the shifted geometric mean (with a shift of 1) and the arithmetic mean of solver-specific measures of the search effort in Table 2. With the exception of MIP

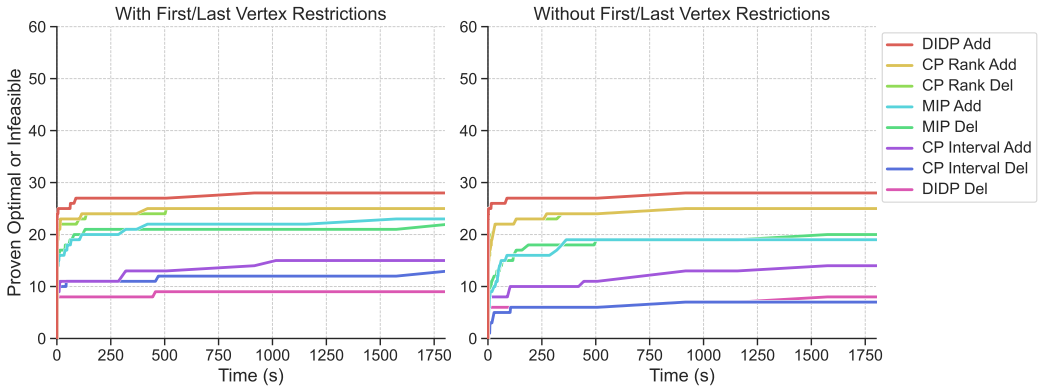


Figure 12 Instances proven optimal or infeasible over time for all exact models. Recall that 21 of the instances are infeasible and the rest admit feasible solutions.

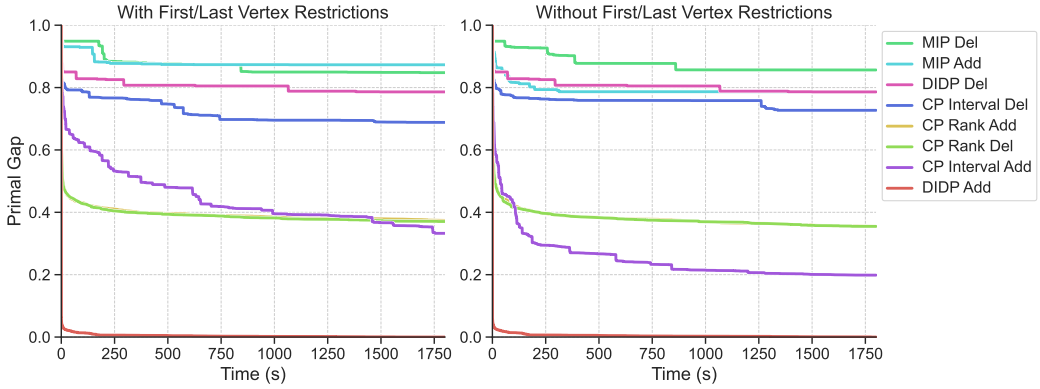


Figure 13 Mean primal gap to best known solution over time of all exact models, averaged over 39 feasible instances. Note the the two CP Rank plots coincide.

Add having almost four times the number of simplex iterations in geometric mean as MIP Del, CP Rank and MIP models show minor differences. The CP Interval models show a larger reduction in most measures of search effort when using the addition variant, consistent with the overall performance results. DIDP shows a substantial difference in the search effort (e.g., 4 orders of magnitude in the arithmetic means of the number of generated nodes), again consistent with the overall performance.

The Impact of First/Last Restrictions. Against our expectations, the inclusion of the first/last vertex restrictions appears to have either no impact or a small positive one on all models in terms of proving optimality or infeasibility. In terms of primal gap, however, both MIP Add and, to a greater extent, CP Interval Add improve when the redundant restriction constraints are *not* present. We speculate that the lack of restrictions allows some primal heuristics to perform better in these solvers. Further analysis is needed to understand the behavior of the models both with and without the first/last restrictions.

Table 2 The arithmetic and shifted geometric mean with shift of 1 of the solver specific search space measures over problems where each pair proved optimality or infeasibility for models without first/last vertex processing. The number in parentheses is the number of such instances.

		Geometric Mean		Arithmetic Mean	
Model	**Measure**	Deletion	Addition	Deletion	Addition
CP Rank (25)	No. Branches	111	139	333 006	239 412
	No. Fails	113	139	157 413	113 665
CP Interval (16)	No. Branches	1104	1069	1 288 442	795 770
	No. Fails	1038	1044	870 779	554 633
DIDP (9)	Nodes Generated	570	23.3	5 332 782	502
	Nodes Explored	518	21.9	4 630 268	421
MIP (21)	Nodes Explored	2.96	2.68	799	787
	Simplex Iterations	5.13	19.7	107 668	109 639

5.2 Experiment 2: Comparison with Previous Approaches

Problem Set. The second experiment uses the 11 instances with sizes from 14 to 1084 vertices adapted from TSPLIB [20] that Carmesin et al. [2] used for a detailed analysis.

Experimental Set-up. We use the same hardware and software as used in Experiment 1 for the single-threaded experiments. Unlike Experiment 1, following Carmesin et al., the time limit for each instance is $10n$ seconds, where n is the number of vertices. We also run each model using 8 threads with a memory limit of 32 GB to better compare with the settings of previous work. For Exact Init., following the literature, we run the single and multi-threaded versions 50 times for each instance and report the mean results.

The best performing approaches in the literature are the *Exact Init.* approach of Carmesin et al. [2] which employs a metaheuristic that is warm-started with the addition DFS approach and the *GRASP* of Woller et al. [22]. As the source code for Exact Init. is provided in the authors' repository,[2] we run it on our hardware based on the published hyper-parameter settings. We rely on the published results for the GRASP approach for which the source code is not available. The mean results for GRASP were back calculated given the gap between the mean results for the Exact Init. We compare the existing state of the art to the three best models in Experiment 1: CP Interval Add without first/last vertex processing, CP Rank Add with first/last vertex processing, and DIDP Add with first/last vertex processing.

Results. Table 3 presents the single-threaded results run on our hardware. DIDP Add dominates all other approaches by a considerable amount.

Moving to eight threads (Table 4), all approaches show some improvement compared to their single-thread results but the relative performance is the same, with DIDP Add again being dominant. In comparison to the literature, DIDP Add out-performs the reported results for Exact Init. on all problem instances and the reported results for GRASP up to and including size $n = 160$. For the three larger instances, GRASP finds the best solutions.

[2] `https://imr.ciirc.cvut.cz/Research/TSPSD`

Table 3 The primal bounds obtained by the exact approaches and mean primal bound for the heuristic algorithms, all run on a single thread for $10n$ seconds with a memory limit of 8 GB. "w." indicates the model was run with first/last vertex restrictions, while "w.o." indicates the model was run without them. The "-" symbol indicates the instances was proved infeasible, while "t.o." is a time-out, and "m.o." indicates a memory-out.

Instance	CP Rank Add w.	CP Interval Add w.o.	DIDP Add w.	Exact Init.
burma14-3.1	**52**	**52**	**52**	**52**
ulysses22-5.5	**141**	**141**	**141**	**141**
berlin52-10.4	-	t.o.	-	t.o.
berlin52-13.2	22 810	17 045	**15 331**	19 741
eil101-27.5	2 270	1 382	**1 183**	1 806
gr202-67.3	1 485	880	**777**	1 402
lin318-99.3	238 643	t.o.	**94 776**	312 641
fl417-160.6	213 448	t.o.	**22 789**	264 009
d657-322.7	465 012	t.o.	**85 547**	414 544
rat783-481.4	89 614	m.o.	**13 458**	62 679
vm1084-848.9	5 517 296	m.o.	**366 429**	2 667 016

Table 4 The primal bounds obtained by the exact approaches and mean primal bound for the heuristic algorithms, all run with 8 threads for $10n$ seconds with a memory limit of 32 GB. The "*" symbol indicates an invalid solution. See Table 3 for the meaning of the remaining symbols.

Instance	CP Rank Add w.	CP Interval Add w.o.	DIDP Add w.	Exact Init.	Exact Init. [2]	GRASP [22]
burma14-3.1	**52**	**52**	**52**	**52**	**52**	**52**
ulysses22-5.5	**141**	**141**	**141**	**141**	166	**141**
berlin52-10.4	-	-	-	t.o.	25 741*	26 231*
berlin52-13.2	20 025	17 438	**15 265**	18 908	17 835	18 852
eil101-27.5	2 153	1 416	**1 187**	1 655	1 513	1 484
gr202-67.3	1 446	835	**777**	1 152	849	870
lin318-99.3	223 836	t.o.	**93 660**	225 803	110 888	105 787
fl417-160.6	119 525	t.o.	**22 272**	27 259	27 259	25 787
d657-322.7	322 985	t.o.	84 257	264 829	85 347	**82 531**
rat783-481.4	69 212	t.o.	13 763	36 148	13 833	**12 906**
vm1084-848.9	5 756 755	m.o.	358 163	913 184	326 067	**305 525**

Note that the berlin52-10.4 instance was proved infeasible by the exact solvers. However, the published results for both Exact Init. and GRASP show that a feasible solution was found. We checked the solution published by Carmesin et al. and confirmed that it uses a deleted edge and thus is invalid. In our execution, the Exact Init. algorithm, correctly, did not find any feasible solutions for berlin52-10.4.

6 Conclusion

In this paper, we investigated three model-and-solve paradigms for the Traveling Salesperson with Self-Deleting graphs (TSP-SD), a problem introduced by Carmesin et al. [2]: two constraint programming (CP) models based, respectively, on ranking and scheduling of vertex visits, a domain-independent dynamic programming (DIDP) model, and a mixed integer

programming (MIP) model. We experimented with four variants of each model, constraining them to find forward sequences with edge deletion or backward sequences with edge addition, and with or without redundant constraints that restricted the start and end vertices.

Our numerical results showed that DIDP solving the addition variant of the problem significantly outperformed all the other exact models, performed better than the state-of-the-art heuristic methods on smaller and medium instances, but trailed the best heuristic approach in terms of solution quality on the largest of the tested instances.

Reformulating the problem to add edges rather than delete them showed little impact on the rank-based CP model and the MIP model but had modest and large positive impact, respectively, for the scheduling-based CP model and the DIDP model.

Our primary direction for future work is to generalize the problem to allow edges to be added and deleted by vertex visits. We expect this problem to be more challenging and, given the poor performance of DIDP on the deletion variant, that CP models may prove superior to the other approaches. We also plan to investigate model-based approaches to more complex state-dependent problems such as scheduling with time- or sequence-dependent costs [1, 3].

References

1 Matan Atsmony, Baruch Mor, and Gur Mosheiov. Single machine scheduling with step-learning. *Journal of Scheduling*, 27(3):227–237, June 2024. doi:10.1007/s10951-022-00763-5.

2 S. Carmesin, D. Woller, D. Parker, M. Kulich, and M. Mansouri. The Hamiltonian Cycle and Travelling Salesperson problems with traversal-dependent edge deletion. *Journal of Computational Science*, 74:102156, 2023. doi:10.1016/j.jocs.2023.102156.

3 Erik Diessel and Heiner Ackermann. Domino sequencing: Scheduling with state-based sequence-dependent setup times. *Operations Research Letters*, 47(4):274–280, July 2019. doi:10.1016/j.orl.2019.04.004.

4 Y. Dumas, J. Desrosiers, E. Gelinas, and M.M. Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations Research*, 43(2):367–371, 1995. doi:10.1287/OPRE.43.2.367.

5 T.A. Feo and M.G.C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67–71, 1989. doi:10.1016/0167-6377(89)90002-3.

6 F. Focacci, A. Lodi, and M. Milano. A hybrid exact algorithm for the tsptw. *INFORMS Journal on Computing*, 14(4):403, fall 2002.

7 A.R. Güner, A. Murat, and R.B. Chinnam. Dynamic routing for milk-run tours with time windows in stochastic time-dependent networks. *Transportation Research Part E: Logistics and Transportation Review*, 97:251–267, 2017. doi:10.1016/j.tre.2016.10.014.

8 Christoph Hansknecht, Imke Joormann, and Sebastian Stiller. Dynamic Shortest Paths Methods for the Time-Dependent TSP. *Algorithms*, 14(1):21, January 2021. doi:10.3390/a14010021.

9 Jahangirnagar University, E. Islam, M. Sultana, and F. Ahmed. A Tale of Revolution: Discovery and Development of TSP. *International Journal of Mathematics Trends and Technology*, 57(2):136–139, May 2018. doi:10.14445/22315373/IJMTT-V57P520.

10 W.-K. Ku and J. C. Beck. Revisiting off-the-shelf mixed integer programming and constraint programming models for job shop scheduling. *Computers & Operations Research*, 73:165–173, 2016.

11 M. Kulich, D. Woller, S. Carmesin, M. Mansouri, and L. Přeučil. Where to place a pile? In *2023 European Conference on Mobile Robots (ECMR)*, pages 1–7, 2023. doi:10.1109/ECMR59166.2023.10256330.

12 R. Kuroiwa. *Domain-Independent Dynamic Programming*. PhD thesis, University of Toronto, 2024.

13 R. Kuroiwa and J.C. Beck. Domain-independent dynamic programming: Generic state space search for combinatorial optimization. In *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling (ICAPS2023)*, pages 245–253, 2023.

14 Raphael Kühn, Christian Weiß, Heiner Ackermann, and Sandy Heydrich. Scheduling a single machine with multiple due dates per job. *Journal of Scheduling*, 27(6):565–585, December 2024. doi:10.1007/s10951-024-00825-w.

15 N. Lipovetzky, C. Burt, A. Pearce, and P. Stuckey. Planning for mining operations with time and resource constraints. *Proceedings of the International Conference on Automated Planning and Scheduling*, 24(1):404–412, May 2014. doi:10.1609/icaps.v24i1.13666.

16 Yiqing L. Luo and J. C. Beck. Packing by scheduling: Using constraint programming to solve a complex 2d cutting stock problem. In Pierre Schaus, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 249–265, Cham, 2022. Springer International Publishing.

17 J.J. Miranda-Bront, I. Méndez-Díaz, and P. Zabala. An integer programming approach for the time-dependent tsp. *Electronic Notes in Discrete Mathematics*, 36:351–358, 2010. ISCO 2010 - International Symposium on Combinatorial Optimization. doi:10.1016/j.endm.2010.05.045.

18 Daniel Pekar and J. Christopher Beck. Travelling Salesperson Problem with Self Deleting Graphs. Software, swhId: swh:1:dir:c287b1067c276693ca630e6e6f7a18daf1a90d38 (visited on 2025-07-28). URL: https://github.com/uoft-tidel/tsp-sd, doi:10.4230/artifacts.23555.

19 J.-C. Picard and M. Queyranne. The time-dependent traveling salesman problem and its application to the tardiness problem in one-machine scheduling. *Operations Research*, 26(1):86–110, 1978. doi:10.1287/OPRE.26.1.86.

20 G. Reinelt. Tsplib—a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991. doi:10.1287/ijoc.3.4.376.

21 D. Woller. URL: https://imr.ciirc.cvut.cz/Research/TSPSD.

22 D. Woller, M. Mansouri, and M. Kulich. Making a complete mess and getting away with it: Traveling salesperson problems with circle placement variants. *IEEE Robotics and Automation Letters*, 9(10):8555–8562, 2024. doi:10.1109/LRA.2024.3445817.

23 J. Zhang and J.C. Beck. Domain-independent dynamic programming and constraint programming approaches for assembly line balancing problems with setups. *INFORMS Journal on Computing*, 2024. in press, published onlinne October 2024. doi:10.1287/ijoc.2024.0603.

A Detailed Results

Table 5 Exact models run for 30 minutes with an 8 GB memory limit over 39 randomly generated feasible instances. The time to first solution (TTFS) in seconds, initial, and final primal gaps are averaged over all 39 instances. If the model experienced a memory-out prior to obtaining a feasible solution, the time was taken as 1800s, and the primal and optimal gap were taken as 100%.

	Model	Feasible	Best	Opt	TTFS	Primal Gap Initial	Primal Gap Final	Optimality Gap Final
Add	CP Rank	39	6	4	1.4	59.1%	35.0%	81.7%
	CP Interval	31	6	6	548	55.0%	31.2%	74.9%
	DIDP	39	39	9	<0.01	22.2%	0.0%	32.9%
	MIP	5	4	3	1 577	91.2%	87.4%	88.0%
Del	CP Rank	39	6	4	1.1	58.9%	34.7%	78.7%
	CP Interval	14	7	6	1 200	79.2%	69.0%	80.9%
	DIDP	9	5	2	1 421	80.8%	78.7%	84.4%
	MIP	6	4	2	1 559	88.8%	84.9%	85.8%
Without First/Last Vertex Restriction								
Add	CP Rank	39	4	4	5.9	59.4%	35.6%	83.3%
	CP Interval	36	5	6	238	47.4%	20.1%	78.8%
	DIDP	39	38	9	<0.01	22.9%	0.1%	33.0%
	MIP	11	4	4	1 395	86.0%	78.5%	83.9%
Del	CP Rank	39	6	4	5.3	58.9%	35.6%	82.6%
	CP Interval	10	3	4	1 372	82.7%	74.8%	88.3%
	DIDP	9	5	2	1 422	80.9%	78.7%	84.5%
	MIP	6	3	2	1 609	88.1%	85.7%	88.0%

Table 6 Performance of exact models, averaged over the 21 infeasible instances. Time taken as 1800s if the instance could not be proven infeasible.

		With First/Last Vertex Restrictions		Without First/Last Vertex Restrictions	
	Model	Infeasible	Time (s)	Infeasible	Time (s)
Add	CP Rank	21	1.2	21	9.4
	CP Interval	11	929	8	1 127
	DIDP	21	<0.01	21	<0.01
	MIP	20	108	15	543
Del	CP Rank	21	2.9	21	10.7
	CP Interval	10	1 021	3	1 543
	DIDP	7	1 222	6	1 350
	MIP	19	179	18	321

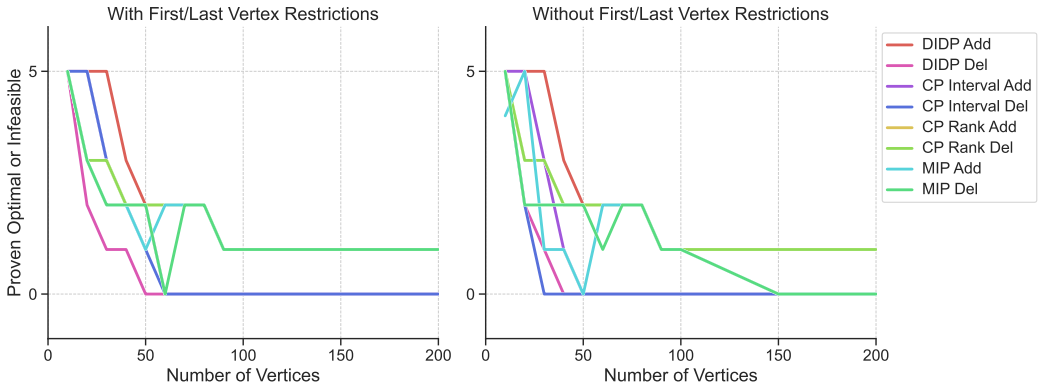


Figure 14 Instances proven optimal or infeasible for all exact models by the number of vertices in the instance for Experiment 1. Recall that 21 of the instances are infeasible and the rest admit feasible solutions, and that there are 5 instances per n.

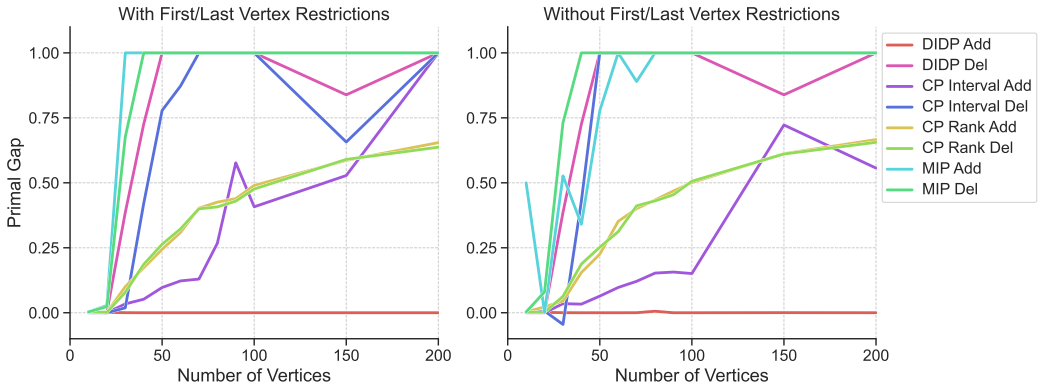


Figure 15 Mean gap to best known solution (i.e., primal gap) by the number of vertices in the instance of all exact models for Experiment 1, averaged over 39 feasible instances.

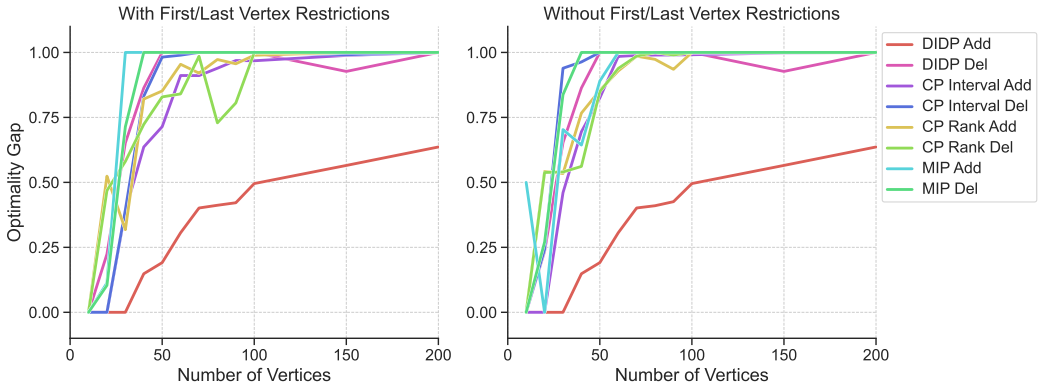


Figure 16 Mean optimality gap by the number of vertices in the instance of all exact models for Experiment 1, averaged over 39 feasible instances.

Transformer-Based Feature Learning for Algorithm Selection in Combinatorial Optimisation

Alessio Pellegrino
Department of Mathematics and Computer Science, University of Southern Denmark, Denmark

Özgür Akgün
School of Computer Science, University of St Andrews, Scotland

Nguyen Dang
School of Computer Science, University of St Andrews, Scotland

Zeynep Kiziltan
Dept. of Computer Science and Engineering, University of Bologna, Italy

Ian Miguel
School of Computer Science, University of St Andrews, Scotland

Abstract

Given a combinatorial optimisation problem, there are typically multiple ways of *modelling* it for presentation to an automated solver. Choosing the right combination of model and target solver can have a significant impact on the effectiveness of the solving process. The best combination of model and solver can also be instance-dependent: there may not exist a single combination that works best for all instances of the same problem. We consider the task of building machine learning models to automatically select the best combination for a problem instance. Critical to the learning process is to define *instance features*, which serve as input to the selection model. Our contribution is the automatic learning of instance features directly from the high-level representation of a problem instance using a transformer encoder. We evaluate the performance of our approach using the Essence modelling language via a case study of three problem classes.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence

Keywords and phrases Constraint modelling, algorithm selection, feature extraction, machine learning, transformer architecture

Digital Object Identifier 10.4230/LIPIcs.CP.2025.31

Supplementary Material
Software (Source Code): `https://github.com/SeppiaBrilla/EFE_project` [45]
archived at `swh:1:dir:0d5708bbc3b0395ddcd80b52bbb6ed8da6ffe252`

Funding This work was supported by the European Union's Justice programme, under GA No 101087342, POLINE (Principles Of Law In National and European VAT), by a scholarship from the Department of Computer Science and Engineering of the University of Bologna, and by UK EPSRC grant EP/V027182/1.

1 Introduction

It has long been observed that no single algorithm performs best on all combinatorial optimisation problems, or even on all instances of one problem [48, 34, 31]. This observation gave rise to automated Algorithm Selection (AS), where the aim is to automatically select the best algorithm(s) from a *portfolio* of algorithms with complementary strengths to solve a given problem instance. AS has been successful in a variety of domains, including Boolean Satisfiability (SAT) [60], Constraint Programming (CP) [44, 18, 5, 6], AI planning [58], and general combinatorial optimisation [35].

© Alessio Pellegrino, Özgür Akgün, Nguyen Dang, Zeynep Kiziltan, and Ian Miguel;
licensed under Creative Commons License CC-BY 4.0

An algorithm can be seen as an automated solver (or a parameter configuration of a solver). However, the concept of an algorithm can be extended to include the *model* (i.e. description) of the problem presented to the solver. There are typically many possible models of a problem, and so high-level constraint modelling languages such as MiniZinc [41] and Essence [23] have been proposed that support the specification of models without commitment to low-level modelling details. Accompanying these languages, modelling toolchains support the automated translation of a higher level representation of a problem to the low-level input supported by automated solvers in different paradigms, such as SAT and CP. These include MiniZinc [41], Conjure [2], and Savile Row [42]. The translation process involves several modelling choices, the combination of which has a significant impact on the performance of the target solver [2].

Adapting AS techniques for the extended context of combining modelling and solving choices is a challenge. AS often relies on training Machine Learning (ML) models to predict the best algorithm(s) for a problem instance based on its features. A good set of input features is critical: they must be informative relative to both the problem instance and the performance landscape of the combination of modelling and solver choices on that instance. One well-known instance feature collection for constraint models is fzn2feat [4], a set of 95 features that can be extracted from a representation of a constraint model written in the FlatZinc modelling language [41]. However, FlatZinc models are low-level representations relative to MiniZinc or Essence models, obtained after modelling choices have been made. Hence, the features extracted are tied to a particular low-level model.

Instead of translating a problem instance into a low-level representation (i.e. FlatZinc) before extracting (fzn2feat) instance features, herein we propose to use a transformer encoder [59] to automatically learn features directly from the high-level representation of the instance. Our approach offers three advantages over the fzn2feat method. First, in contrast to the hand-crafted fzn2feat features, our approach learns instance features automatically from the textual description of a problem instance. Second, fzn2feat relies on a low-level problem representation in FlatZinc, while our approach works directly on a high-level representation, which can offer more information for the task of choosing the best combination of model and solver. Third, as shown empirically, the proposed features, once learned, are computationally cheaper to extract. We demonstrate our approach using the Essence constraint modelling pipeline via three case studies that present disparate modelling and solving challenges: Car Sequencing [24], Covering Array [30] and Social Golfers [54].

In the rest of the paper, after giving the background and discussing the related work in Sections 2 and 3, we introduce in Section 4 our methodology and in Section 5 our case studies. Then, we present in Section 6 the experimental evaluation the methodology and finally conclude in Section 7.

2 Background

2.1 High-level constraint modelling

MiniZinc [41] and Essence [23] are examples of domain-specific modelling languages. Essence supports higher-level concepts in abstract domains, such as functions, relations and sets, and nesting of these (e.g. set of functions). This enables concise problem specification without committing to low-level modelling details. Since abstract domains are not directly represented in typical solvers, they must be *refined* into commonly supported concrete domains, such as integers, Booleans and collections of these into arrays or sets.

Conjure [2] refines Essence into Essence Prime, a lower level solver-independent constraint modelling language [42] similar to MiniZinc. It is able to produce a portfolio of models from a single Essence specification, predominantly by exploring alternative refinement pathways for decision variables with abstract domains. This creates a complex algorithm selection landscape: selecting representations for decision variables with abstract domains is akin to viewpoint selection and it is known to significantly affect solution performance, depending on the problem instances being solved [51]. In Section 5, we present the high-level representations of the three case study problems in Essence and describe the alternative Essence Prime models generated by Conjure.

2.2 Transformer architecture

Neural Networks (NNs) are a powerful ML paradigm, able to learn complex patterns from large datasets without user-defined features [20, 36]. NNs have been successful in a wide range of tasks, including text classification [59]. NNs have multiple layers, each processing the input obtained from the output of the previous layer. The first layer accepts the raw input, and the last produces the final output. Layers are typically followed by activation functions, which introduce non-linearity to facilitate learning [26]. The final activation function projects the output of the last layer onto the desired range (e.g. probabilities for classification models). The most fundamental layer type is the linear layer of n neurons, each computing an affine transformation of the input from the previous layer to produce an output. The n output features form an n-dimensional vector [26].

A specific combination of layers defines its architecture. Over the years, many successful architectures have been presented, each with specific advantages and disadvantages. One of the most famous architectures is the *transformer* [59], which is specifically designed to address sequential data such as text, more efficiently than traditional recurrent models. The key innovation of the transformer is its *attention mechanism*, which allows the model to capture relationships between all elements in a sequence simultaneously, rather than processing them sequentially. By considering each input element in the context of all others, rather than in isolation, Transformers have achieved state-of-the-art performance in numerous tasks, including text classification [20, 33].

3 Related Work

ML-based techniques for algorithm selection have been proven effective in many problem domains [60, 58, 38, 34, 31]. The selection of the best algorithm(s) is via the characteristics of the given instance, represented as a feature vector. The goal is to construct predictive machine learning models that maps these instance features to the best algorithm(s), thereby optimising a predefined performance metric (e.g. runtime or solution quality). There is often an additional cost associated with feature extraction, which is taken into account when measuring the performance of an AS approach.

Many ML-based AS tools have been proposed for combinatorial optimisation problems. SATZilla [43, 60, 61] is a state-of-the-art SAT solving system that combines existing SAT solvers in a portfolio and applies AS on top of the constructed portfolio. In addition to adopting various ML techniques (i.e., empirical hardness models for predicting algorithm performance [43, 60], and cost-sensitive pairwise classification approaches for predicting the better algorithms in an algorithm pair [61]), a key success of SATZilla is the development of a comprehensive set of features to characterise SAT instances. SATZilla's instance features were expertly designed to capture the characteristics of SAT instances, which help ML models

to effectively predict SAT solver performance. In fact, in a recent study, a revisited and enhanced version of this SAT feature set has been shown to provide significant improvement to SATZilla's performance [50], indicating the importance of instance features in AS systems in SAT contexts.

Similarly, in CP contexts, several AS-based solving approaches have been proposed. CPHydra [12] and SUNNY [5, 6, 39] focus on computing an instance-specific schedule of solvers via k-nearest neighbours algorithms. CPHydra uses a small set of 36 features developed by the authors of the work, while SUNNY makes uses of `fzn2feat` [4], a comprehensive set of features specially designed for characterising constraint problems in FLATZINC representations. Recently, highly effective SAT encoding selection approaches for Pseudo-Boolean and linear integer constraints have been developed via a combination of various ML-based AS methods and well-designed instance features [55, 56, 57].

These works highlight the critical role of high-quality instance features in AS approaches. However, designing such features is a challenging task and requires extensive knowledge of the corresponding domains. In response to this challenge, automated feature learning approaches have started to emerge. Recurrent NNs were adopted to learn features directly from the input sequence of item sizes in online bin packing problems, which were then used for selecting among different bin-packing heuristics [3]. In the Travelling Salesman Problem (TSP) domain, a transformer-based architecture was developed to learn instance features directly from the raw TSP instance input [49]. The transformer has also been adopted in a black-box optimisation domain for selecting the best solving algorithm [13]. In this case, the input is the trajectory of the optimisation function. Unlike all the previous work cited, our approach learns features directly from the textual description of any combinatorial optimisation problem instance.

4 Methodology

We assume that we are a given (i) a textual description of a combinatorial optimisation problem instance written in a high-level modelling language (such as ESSENCE) and (ii) a *portfolio of algorithms* to solve the instance, each a combination of a lower-level model (such as an ESSENCE PRIME model) and a solver. Our AS task is to predict with ML the *best* algorithm for a given instance from this portfolio, defined here as that which solves the given instance in the shortest runtime.

Integrating ML in AS requires: (i) extracting a set of features that accurately characterise the instance, and (ii) using these features to predict the best algorithm in the portfolio for the instance. In this section, we describe our methodology to these two requirements. Our approach is agnostic regarding satisfaction versus optimisation – it just requires that algorithm performance can be measured as a numerical value.

4.1 Transformer-based Neural Network for Feature Learning

We propose to employ an NN that encapsulates a transformer encoder [20] to deal with textual input. This approach has many advantages. First, an NN model can automatically generate the necessary features starting from the raw input. This eliminates the need for handcrafting an effective feature set. Second, transformer encoders such as BERT have been proven to be effective in capturing semantic meaning of textual data, [20], eliminating the need to run a solver to extract the necessary features. For example, since all parameter names are identical across problem instances, we expect the learnt features to capture differences in parameter values. Intuitively, the parameter values of an instance are likely to be correlated

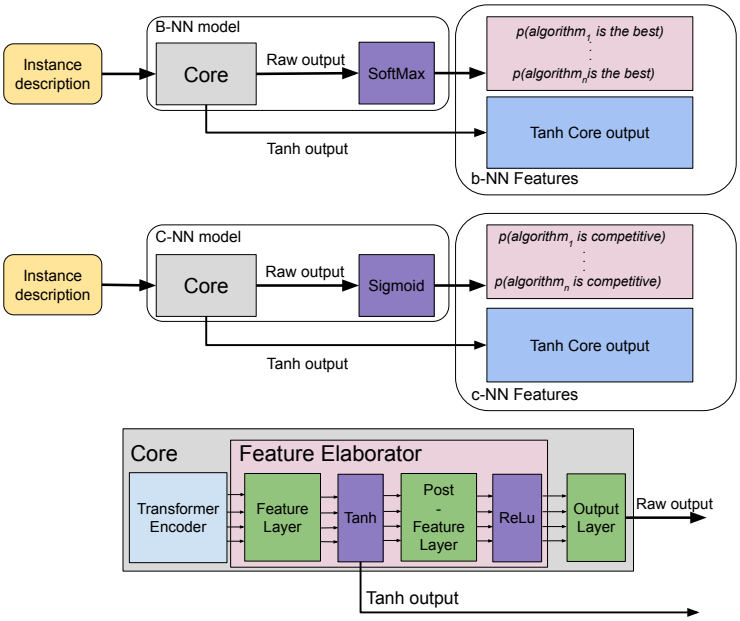


Figure 1 The NN models for feature learning.

with its characteristics, such as problem size, which, in turn, are likely to influence the performance of an algorithm. In principle, our approach could work with any textual input (such as natural language). The advantage of starting from a formal representation (such as Essence) is that it is more structured and precise, while natural language tends to contain ambiguity.

We build two NN models, called B-NN and C-NN and depicted in Figure 1, both of which receive as input the raw text of the instance in tokenized form (where each input word and symbol are transformed into a number).

B-NN Model. This model learns the best algorithm for a given instance. Learning is modelled as a multi-class classification task where the assigned class is the best algorithm. The final activation function is SoftMax, which generates a probability distribution over all the algorithms in the portfolio, where a higher probability indicates a higher likelihood to be the best. SoftMax activation is well suited to associating the highest probability to one best algorithm.

C-NN Model. Learning one best algorithm is restrictive when multiple algorithms perform similarly well. The C-NN model thus learns the *competitiveness* of the algorithms in the portfolio. We consider an algorithm competitive if it solves an instance in under ten seconds or less than double the time taken by the best algorithm for that instance. Hence, multiple algorithms may be deemed competitive. We propose to model learning as a multi-label classification task where each algorithm is associated with a competitiveness fraction. The final activation function is Sigmoid, which yields a probability per algorithm, and is well suited to our purpose: there could be multiple equally competitive algorithms and the competitiveness probability of one algorithm is uncorrelated with that of the others. We refrain from using SoftMax activation with the top k competitive algorithms, as the number

of competitive algorithms is not fixed. Imposing a fixed value of k may constrain the model when the actual number of competitive algorithms exceeds k, and may lead to incorrect representations when it is fewer than k.

The two models share the same architecture, differing in their final activation function. The probability values in their output will be part of the extracted features for AS (as described in Section 4.2). The common part of the models (referred to as *core*) is composed of three components: (i) Transformer Encoder, (ii) Feature Elaborator, and (iii) Output Layer. Increasing the depth of the network by incorporating multiple layers with different objectives enhances the learning process [53].

Transformer Encoder. Transformer Encoder transforms input instance text into a feature vector describing its semantic meaning without any pre-processing. The encoder is based on the BERT-base-uncased [20] architecture and shares hyper-parameters with the original except an increased input size (2048 tokens instead of 512) to support large textual input. Our choice of architecture is motivated by several factors. First, it has been successfully applied to classification tasks [46], making it a strong candidate for our purposes. Second, since we do not employ a pre-trained model, BERT shares the same architecture as RoBERTa [40]. Third, although architectures such as Longformer [9] offer a larger context window and more parameters, they are also more computationally expensive, and our preliminary experiments did not demonstrate any clear advantage.

Feature Elaborator. Feature Elaborator passes the feature vector from the Transformer Encoder through two layers and the corresponding activation functions to project the features into the desired dimension. The Feature Layer is a linear layer that condenses the feature vector, mitigating the risk of dimensionality issues in subsequent non-neural ML applications (curse of dimensionality) [17]. Tanh activation replicates the last activation function of the BERT Transformer Encoder on the reduced feature vector. Its output will be part of the extracted features for AS (as described in Section 4.2). Post-Feature Layer is a linear layer that up-projects the feature vector, allowing the model to further elaborate the features avoiding underfitting [47]. Relu activation introduces non-linearity on the up-projected vector to facilitate learning.

Output Layer. Output Layer operates on the features to predict a floating point score per algorithm in the portfolio (referred to as Raw output). The final activation functions of B-NN and C-NN transform raw output into probabilities to better interpret it.

4.2 Algorithm Selection Using the Learnt Features

To address the second requirement of our ML-based AS we propose two approaches: integrating feature learning and AS within a single NN model (referred to as *fully neural*), or exploiting an external ML-based AS algorithm using the extracted features (referred to as *hybrid*). While the first approach seems natural and more straightforward, the latter allows exploiting state-of-the-art AS tools and experimenting with alternative AS strategies.

Once the NN models are trained, we can use B-NN as a fully neural approach to predict the best algorithm by using the probability values in the output as features and picking the algorithm associated with the highest probability. For the hybrid approach, we extract features from each NN model by combining the probability values in the output with the output of Tanh in a single feature vector. We have observed in a preliminary study that the models trained using only the Tanh output misclassified some instances with a prediction

that was not deemed as competitive by the NN. We thus combined it with the probability output to limit this type of error. The combined feature vector integrates the encoder-derived semantic representation with informative prediction, as we will show in Section 6.5. We refer to such combined feature vectors as b-NN and c-NN features (Fig. 1).

As an external ML-based AS algorithm, we consider both the state-of-the-art Autofolio [38] and a simpler alternative. Autofolio can perform both classification and regression and can be tuned via SMAC [37]. The tuning process can choose between random forest for classification, random forest for regression, and XGboost [14], as well as refine the sets of hyper-parameters via Bayesian optimisation. Each configuration is evaluated using 10-cross-fold evaluation.

A simpler alternative is clustering, motivated by its success in AS [8] and the spatial characteristics of transformer encoder embeddings [27]: semantically similar instances are represented by similar vectors. We cluster the instances using K-means [1] based on their similarities and assign to each cluster the algorithm with the lowest total runtime on the cluster instances. For optimisation, we pre-define a set of hyper-parameter configurations, each evaluated by fitting a model on the training data and scoring its performance on the evaluation data. The score is calculated by assigning validation instances to their respective clusters and evaluating them with the corresponding algorithm. The configuration with the lowest total score is selected as the optimal. At test time, an unseen instance is assigned to a cluster based on the best-fit configuration and the cluster's algorithm is applied as the best algorithm.

In summary, we have five ML-based AS approaches: fully neural `B-NN` and four hybrids `(b-NN,A)`, `(b-NN,K)`, `(c-NN,A)`, `(c-NN,K)`, where `A`, and `K` denote Autofolio and K-means.

5 Case Studies

We evaluate our methodology using the Essence constraint modelling pipeline via three well-known, challenging problems: Car Sequencing, Covering Array and Social Golfers. In the following sections, we first present an Essence model for each problem, followed by a description of its algorithm portfolio, which consists of various combinations of an Essence Prime model (obtained from Conjure) and a solver. To establish suitability for AS, we then show for each problem that the algorithms in the portfolio have complementary performance on the chosen instance set, i.e. there is no dominating algorithm.

We use instances from [52], generated using the AutoIG framework [19], which systematically varies problem parameters to produce diverse and challenging instance set. Our selection of instances varies in domain size and number of variables, covering a wide range of problem sizes (small, medium, and large) and levels of difficulty. All models and instance sets are publicly available in the Essence Catalogue [16] and in the supplementary repository. [1]

5.1 Problem Description in Essence and Instance Set

Car Sequencing. Car Sequencing [22] involves scheduling the production of a set of cars, each potentially requiring different optional features. The production line is divided into stations, each responsible for installing a particular option (e.g., air conditioning, sunroofs). Figure 2a shows the parameter and decision variable declarations in an Essence model. It

[1] `https://github.com/SeppiaBrilla/EFE_project/`

```
given n_cars, n_classes, n_options : int(1..)

letting
    Slots  be domain int(1..n_cars),
    Class  be domain int(1..n_classes),
    Option be domain int(1..n_options)

given
    quantity : function (total) Class  --> int(1..),
    maxcars  : function (total) Option --> int(1..),
    blksize  : function (total) Option --> int(1..),
    usage    : relation of ( Class * Option )

find car : function (total) Slots --> Class

such that
    forAll c : Class . |preImage(car,c)| = quantity(c),
    forAll opt : Option .
        forAll s : int(1..n_cars+1-(maxcars(opt)+blksize_delta(opt))) .
            (sum i : int(s..s+(maxcars(opt)+blksize_delta(opt))-1) .
                toInt(usage(car(i),opt))) <= maxcars(opt)
```

(a) Car Sequencing, CSPLib #1. Uses a single function variable and a preImage operator on the function variable.

```
given t : int(1..) $ strength (size of subset of rows)
given k : int(1..) $ rows
given g : int(2..) $ number of values
given b : int(1..) $ columns

where k>=t, b>=g**t

find CA: matrix indexed by [int(1..k), int(1..b)] of int(1..g)

such that
    forAll rows : sequence (size t) of int(1..k) .
        (forAll i : int(2..t) . rows(i-1) < rows(i)) ->
        forAll values : sequence (size t) of int(1..g) .
            exists column : int(1..b) .
                forAll i : int(1..t) .
                    CA[rows(i), column] = values(i)

$ symmetry breaking
such that forAll i : int(2..k) . CA[i-1,..] <=lex CA[i,..]
such that forAll i : int(2..b) . CA[..,i-1] <=lex CA[..,i]
```

(b) Covering Array, CSPLib #45. Uses a single matrix variable and quantified expressions over sequence domains to state the problem constraint concisely.

```
given w, g, s : int(1..)

letting Golfers be new type of size g * s

find sched :
    set (size w) of
        partition (regular, numParts g, partSize s)
            from Golfers

such that
    forAll g1, g2 : Golfers, g1 != g2 .
        (sum week in sched . toInt(together({g1, g2}, week))) <= 1
```

(c) Social Golfers, CSPLib #10. Uses a single decision variable whose domain is a set of partitions. The problem constraint is concisely stated using a quantified expression over the set variable and the together operator that works on partitions.

Figure 2 ESSENCE models for the three problem classes. Each model contains a single decision variable with an abstract domain and a single top-level problem constraint. The Covering Array problem also has explicit symmetry-breaking constraints.

defines three integer parameters: *n_cars*, *n_classes*, and *n_options*, from which domains *Slots*, *Class*, and *Option* are derived. Further parameters refine the problem: *quantity*, mapping each class to the number of cars required; *maxcars*, mapping each option to the maximum number of cars allowed in any consecutive block; *blksize*, a function mapping each option to the size of each consecutive block; and *usage*, a relation indicating which classes require which options. A decision variable with an abstract domain, *car*, maps from production slots to classes.

The problem contains a single top level constraint. This constraint uses a sliding-window mechanism to control the distribution of options along the car sequence. For each option, it defines a window of consecutive car slots whose length is the sum of the maximum allowed cars with that option and an additional block size delta. Within every such window, the total number of cars requiring the option must not exceed the allowed maximum. This ensures that the usage of each option is evenly distributed, preventing any clustering that could overload production capabilities.

We use 10,214 car sequencing instances.

Covering Array. Covering Array [30] originates from hardware design. A covering array $CA(t, k, g)$ of size b is a $k \times b$ array over $\{0, \ldots, g-1\}$ in which every t-way combination of row indices and values appears in at least one column. The smallest such b is called the covering array number, $CAN(t, k, g)$. We use the decision version of the problem (Figure 2b). The input parameters of the model are: t, the strength, k, the rows, g the array's values domain, b the number of columns of the covering array. The decision variable is CA: an integer-indexed matrix of integer values in the range 1 to g.

The first constraint in this model requires that for every strictly increasing sequence of t rows, every possible combination of t values (from the g available) is represented in at least one column of the matrix CA. This guarantees that the matrix forms a covering array of strength t. The remaining two constraints are for symmetry breaking.

We use 2,236 instances for this problem.

Social Golfers. Social Golfers [54] involves scheduling golfers in g groups of size s for w weeks, such that no pair of golfers meets more than once. Figure 2c shows the parameter and decision variable in an Essence model of the problem. An unnamed type, *Golfers*, represents all the golfers in the schedule, with size $g \times s$. The decision variable, *sched*, is a set of partitions, each corresponding to a week's grouping of golfers.

The only constraint in this model is that of socialisation: it ensures that any two distinct golfers are paired together in the same group in at most one week. For every pair, it sums a Boolean indicator (converted to an integer) across all weeks, with each indicator signalling whether the pair played together in that week. The total for each pair is constrained to be no more than one, ensuring that no pair of golfers is scheduled to play together more than once over the course of the w weeks.

We use 1,039 instances for this problem.

5.2 Combinations of Essence Prime Models and Solvers

We use Conjure in portfolio mode to generate up to four Essence Prime models from a single Essence specification, exploring different ways to represent variables and constraints. Key model variations for each problem are summarised below. We then combine the various Essence Prime models with four solvers to create a diverse set of algorithms that may perform well on different subsets of instances.

Car Sequencing. Car Sequencing has three models based on two representations for the *car* decision variable and the *usage* parameter. The *car* variable can be represented as either a one-dimensional array of integers or a two-dimensional Boolean array, while the *usage* parameter can be either a Boolean array or a set of tuples. Model M_1 uses a one-dimensional array for *car* and a set of tuples with a `table` constraint for *usage*. Model M_2 uses a one-dimensional array for *car* with a Boolean array for *usage* and the `element` constraint. Model M_3 uses a two-dimensional Boolean array for *car* and a set of tuples with a `table` constraint for *usage*.

Covering Array. Covering Array has one model M_1 because the problem is expressed with a single matrix decision variable, and matrix variables map directly to Essence Prime. The constraints likewise do not permit additional variations.

Social Golfers. Social Golfers has four models offering different matrix-based encodings of the weekly partitions. Model M_1 uses a single 3D matrix indexed by $weeks \times groups \times slots$, where each entry is an integer indicating which golfer is in a particular part of the partition for a particular group and week. M_2 uses a 3D Boolean matrix, indexed by *weeks*, *groups*, and *golfers*, to indicate whether a given golfer is in a specific part of the partition. M_3 splits the partitioning into multiple matrices, capturing information such as the number of parts, which golfer goes into each position, and how many slots are used in each group. By breaking down the partition into several matrices, this approach can exploit different constraint formulations within the same overall representation. M_4 combines two matrix encodings: one stores explicit integer assignments for each slot and another tracks membership via Booleans.

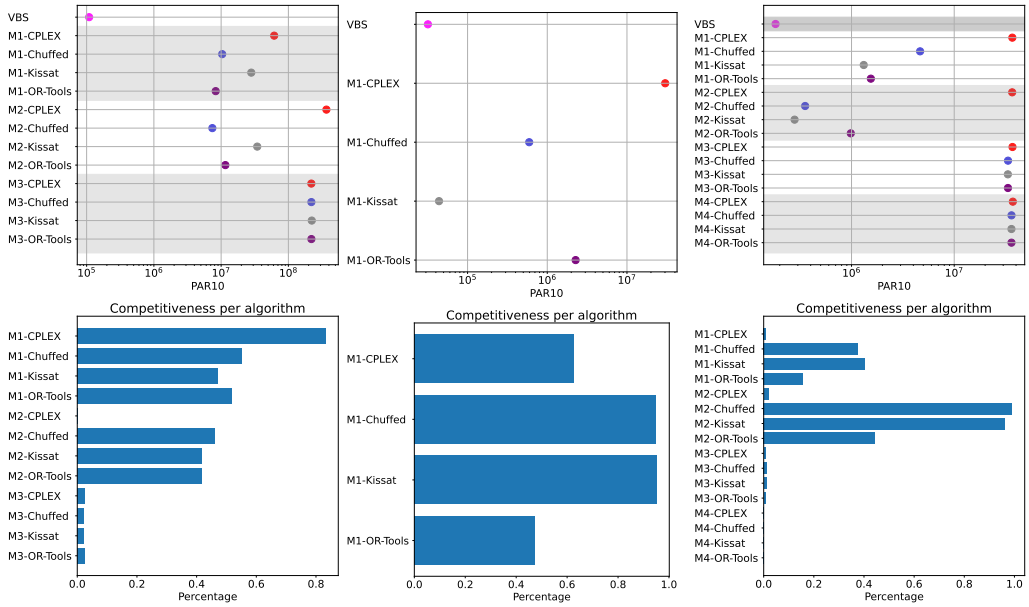
Solvers. We use Kissat [10], a SAT solver; Chuffed [15], a CP solver with lazy clause generation; CPLEX [29], a commercial MIP solver; and OR-Tools CP-SAT [21], a hybrid solver combining clause learning, CP-style propagation, and MIP-based methods. We interface to these solvers via Savile Row [42].

5.3 Algorithm Complementarity

We analyse the portfolio of algorithms employed for each problem and assess the potential benefits of applying AS. In Section 4, we defined the best algorithm in terms of shortest runtime. In practice, we often impose a cut-off time on every algorithm. When an algorithm fails to solve an instance within the cut-off, we penalise the run using the PAR10 (Penalised Average Runtime) metric [38], where the runtime of a failed run is recorded as 10 times the cut-off time.

All experiments were conducted on a computational setup equipped with an AMD EPYC 7763 CPU. Each algorithm was allocated a single CPU core with a one-hour cut-off per instance. As baseline, we will use the Single Best Solver (SBS) algorithm, defined as the algorithm with the lowest PAR10 score in the whole instance set, and the Virtual Best Solver (VBS), a theoretical construct representing the optimal algorithm selector that always identifies the best algorithm for each instance.

Car Sequencing. Car Sequencing uses a portfolio of 12 algorithms derived from the combination of three Essence Prime models and four constraint solvers. Figure 3a (top) shows the PAR10 scores for these algorithms, evaluated across the entire instance set. A key observation from Figure 3a (top) is the lack of a dominant model or solver: there is no algorithm with



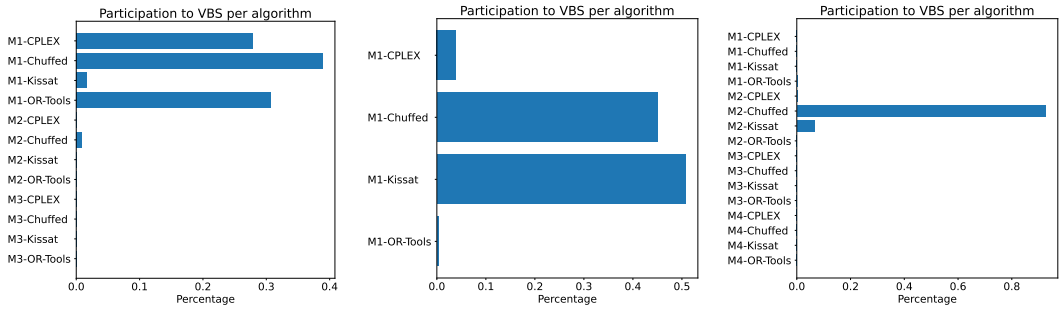
(a) Car Sequencing. **(b)** Covering Array. **(c)** Social Golfers.

Figure 3 PAR10 value of each algorithm and the VBS in log scale (top) and algorithm competitiveness in percentage (bottom).

the same behaviour as VBS, meaning that no algorithm is the "best" on each instance. The algorithms, excluding those involving M_3, which always underperform independently of the solver, exhibit diverse performance characteristics. Notably, the gap between the VBS and the SBS algorithm (M_2-Chuffed) is significant, with the SBS achieving only 0.01% of the VBS's performance. This underscores the potential for AS to leverage complementary strengths among algorithms. Figure 3a (bottom) shows the average competitiveness (as the percentage of the instances where the algorithm is competitive). All algorithms are competitive on some instances, except M_2-CPLEX, which was the worst overall algorithm. It is typically very difficult for an AS method to always select the best algorithm for a given instance. At the same time, this may not always be necessary, as competitive algorithms could also do well on the instance.

We can see that even though M_2-Chuffed appears to be the best overall algorithm in Figure 3a, it is the winner on a fairly small number of instances according to plot of Figure 4a. Instead, M_1-CPLEX, M_1-Chuffed and M_1-OR-Tools have significantly higher numbers of instances where they win. These three algorithms, which are also the top three competetive algorithms according to Figure 3a (bottom), cover a significant part of the instance space.

Covering Array. Covering Array has one Essence Prime model, hence four different algorithms combining the model with the four solvers. Figure 3b (top) shows PAR10 scores of each algorithm in the full instance set. Contrary to Car Sequencing, for this problem class the gap between SBS and VBS is much smaller. With M_1-Kissat being the SBS, VBS as a fraction of SBS is 0.72. In terms of competitiveness (bottom figure), we can see that M_1-Chuffed and M_1-Kissat are almost always competitive, M_1-CPLEX is competitive around 60% of the time and M_1-OR-Tools roughly 50% of the time. This second view shows a different story from the top figure where M_1-Kissat seems to be the clear winner. This suggests a much more complex instance set where AS could take advantage of the different algorithms.



(a) Car Sequencing. **(b)** Covering Array. **(c)** Social Golfers.

Figure 4 Participation as a percentage to VBS.

Even though the gap between SB and VBS is much smaller compared with the Car Sequencing case, Figure 4b shows that each algorithm contributes to VBS, even M_1-CPLEX which has the worst PAR10 score. M_1-OR-Tools is the least contributing algorithm while M_1-Chuffed and M_1-Kissat, the main competitive algorithms according to Figure 3b (bottom), both contribute similarly to VBS, with M_1-Kissat contributing slightly more than 50% of the time and M_1-Chuffed roughly 45% of the time.

Social Golfers. Social Golfers has four Essence Prime models, resulting in sixteen different algorithms in conjunction with the four solvers. The PAR10 scores of Social Golfers (Figure 3c, top) differ from those of the previous problems: 6 algorithms dominate all the others with similarly bad results. The algorithms including the models M_1 and M_2 have the best performance when coupled with Chuffed, Kissat and OR-Tools. The same cannot be said for CPLEX because it performs similarly to the other algorithms. The gap between SBS and VBS is in the middle between Car Sequencing and Covering Array since VBS as a fraction of SBS is 0.65. M_1-Chuffed, M_1-Kissat, M_1-OR-Tools, M_2-Chuffed, M_2-Kissat and M_2-OR-Tools are the only competitive algorithms more than 10% of the time (Figure 3c, bottom). This may seem less than ideal. However, the gap between SBS and VBS shows that whenever M_2-Kissat is not the best algorithm it is worse by a considerable margin, motivating AS.

The participation to VBS, in Figure 4c, shows that, although M_2-Kissat is the SBS, M_2-Chuffed is the algorithm that contributes by far the most to VBS while M_2-Kissat wins only in a handful of instances. This is a clear indication of the potential of AS for this problem class.

We conclude that each problem in our case study, along with its set of instances and algorithm portfolio, serves as a suitable candidate for evaluating our methodology.

6 Experimental Results

We experimentally evaluate the effectiveness of our five ML-based AS approaches proposed in Section 4: the fully neural `B-NN` and the four hybrid approaches `(b-NN,A)`, `(b-NN,K)`, `(c-NN,A)` and `(c-NN,K)`. All source code and data are available in the supplementary repository. [1] The following questions guide the evaluation: (Q1) Is the fully neural approach effective for AS, or is it necessary to split the learning process into two phases (like the hybrids)? (Q2) How do the learnt features compare to the existing fzn2feat features in solving the AS task? (Q3) What is the computational cost of the feature extraction process and its impact on the AS performance? (Q4) Is it beneficial to combine the propabability values in the output of an NN model with the output of Tanh for feature learning?

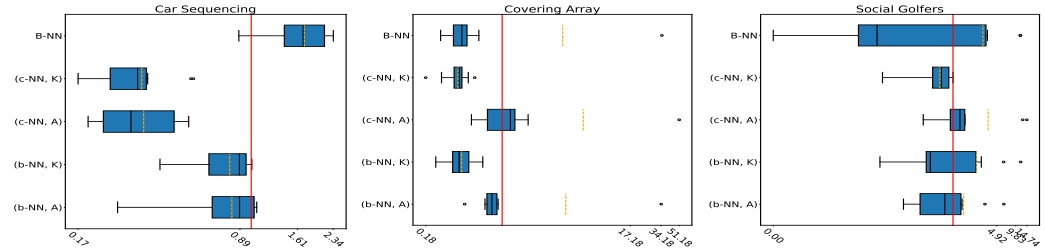


Figure 5 Normalised PAR10 scores of `B-NN` and the four hybrid approaches on the test set across 10 folds. Red line is SBS.

6.1 Experimental Design

Algorithm Selection Setup. Each AS approach is evaluated using PAR10 across 10-fold cross validation. In each fold, 10% of the training data serves as a validation set to support neural network training and hyper-parameter optimisation. We use the same computer configuration as in Section 5.3 for the CPU-based experiments, while GPU is used for neural network training and inference, as detailed below. In the hybrid approaches, we make use of K-means and AutoFolio for the AS task. We tested the number of clusters in K-means in the range [2,21] and chose that which resulted in the lowest validation PAR10 score. AutoFolio is an AS framework that offers a tuning mode with its own hyper-parameter configuration space. The tuning is done using the hyper-parameter optimisation tool SMAC [28], which we employ with a single CPU core and a tuning budget of 5 hours.

Neural Network Training. All NN models are implemented using PyTorch [7] and trained from scratch on a GPU equipped with an NVIDIA A5000 accelerator. The Adam optimiser [32] is employed for all training processes. Model hyper-parameters, including the learning rate, (mini-)batch size, and number of epochs, are selected based on a combination of computational resource constraints and manual experimentation. The final hyper-parameter values are detailed in Section A. Cross Entropy is applied in the B-NN models, while Binary Cross Entropy is used in the C-NN models. In addition to the fold mechanism used as a precaution against overfitting, we monitor the validation loss during training and retain the model with the lowest validation loss to further reduce the risk of overfitting.

Normalised PAR10 Score. Due to the different scales of PAR10 across folds, following the existing AS literature [38], we define the normalised score as $(p(AS) - p(VBS))/(p(SBS) - p(VBS))$, where $p(AS)$, $p(VBS)$ and $p(SBS)$ are the PAR10 scores of an AS approach, the VBS, and the SBS on the same fold, respectively. VBS has a score of 0 while SBS has a score of 1. We want to minimise the normalised score. Feature extraction time is included in all PAR10 calculations unless its exclusion is mentioned explicitly. An AS approach is considered effective if its normalised score is less than 1, i.e., it performs better than SBS – the AS approach without any learning required.

6.2 Fully Neural vs Hybrid Approaches

In this section we address Q1 and compare the fully neural `B-NN` with its hybrid counterparts `(b-NN,*)` and `(c-NN,*)`. Fig. 5 presents the normalised PAR10 scores of the five approaches across 10 folds. Overall, `B-NN` is generally less competitive than the hybrids. Among the hybrids, `(c-NN, K)` consistently obtains the best overall performance across the

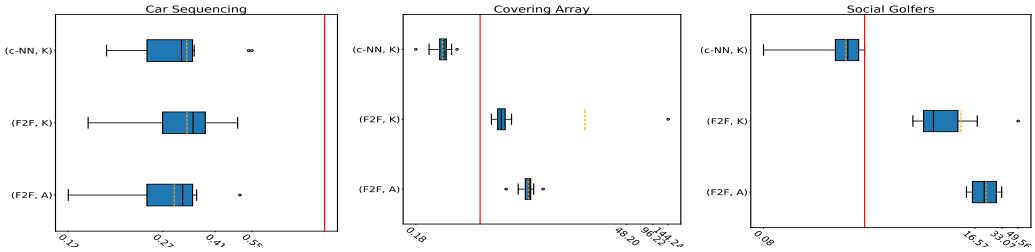


Figure 6 Normalised PAR10 scores of `(c-NN,K)` and the two F2F-based AS approaches on the test set across 10 folds. Red line is SBS.

three problems. It is the only one that consistently achieves better performance than SBS. A likely explanation for the underperformance of `B-NN` is the inherent imbalance of the training data in our multi-class classification setting. Algorithms that excel on only a small subset of instances are under-represented, making them harder to be detected despite their significant impact on the overall PAR10 score. This is confirmed by [11], which demonstrated that treating this task as a pure multi-class classification is generally the least effective approach, since classification accuracy does not account for the actual runtime of each predicted algorithm and thus is not aligned with the final performance metric. The hybrid approaches mitigate this limitation by separating feature learning and AS, and the ML models used in the AS part do take into account performance difference between algorithms, allowing training to be tailored towards the true performance metric PAR10. Finally, `(c-NN,K)` performs much better than `(c-NN,A)`, despite K-means' simplicity w.r.t. Autofolio. Our finding illustrates that simpler ML-based AS approaches can be quite effective for certain AS tasks compared with the more sophisticated ones commonly used in the AS literature.

6.3 Comparison with FZN2FEAT Features

Having established that `(c-NN,K)` is the best approach for automatically learning instance features, we consider Q2 by comparing our learnt features with FZN2FEAT (`F2F`) [4], which were expertly-designed for representing constraint models. Since `F2F` only works in the low-level constraint representation in FlatZinc, we first need to translate each problem instance into the FlatZinc format [41]. This translation involves choosing a specific combination of an ESSENCE PRIME model and a solver. We use CONJURE's default model M_1 and the Chuffed solver. For each instance, the `F2F` features are extracted using 2 CPU cores and 8GB RAM, and `c-NN` feature extraction uses the same GPU-enabled computer as in the training process. The use of different machines is motivated by the slower CPU on the GPU-enabled machine and the observation that fzn2feat runs faster on a CPU-only machine. This setup leverages the hardware characteristics best suited for each computation. While we expect weaker performance on CPU-only systems, emerging consumer hardware with unified memory architectures is likely to narrow this gap. We observe that the memory usage of the `c-NN` feature extraction process remains under a few GBs; however, in some runs, `F2F` extraction crashes due to memory limits. In such cases, we revert to SBS as the best algorithm for the corresponding problem instance.

`F2F` features combined with K-means and Autofolio result in two new AS approaches `(F2F,K)` and `(F2F,A)`. Fig. 6 shows their normalised PAR10 scores and `(c-NN,K)`'s on the test set across 10 folds. On Car Sequencing, `(c-NN,K)` outperforms `(F2F,K)` on seven out of ten folds. Compared with `(F2F,A)`, `(c-NN,K)` obtains the best overall median score. The strength of `(c-NN,K)` is even clearer on the remaining two problems, where it performs

Table 1 Average PAR10 scores (in seconds) of `c-NN`-based and `F2F`-based approaches with (+ E) and without feature extraction cost on the test set across 10 folds. The AS model is K-means.

fold	Car Sequencing				Covering Array				Social Golfers			
	F2F + E	c-NN + E	F2F	c-NN	F2F + E	c-NN + E	F2F	c-NN	F2F + E	c-NN + E	F2F	c-NN
1	**132.51**	300.57	**127.17**	300.52	23.23	**14.85**	19.23	**14.85**	1,547.11	**209.45**	1,224.51	**209.44**
2	301.97	**156.22**	296.69	**156.17**	28.60	**20.99**	25.00	**20.99**	1,163.31	**268.35**	848.11	**268.35**
3	273.85	**212.42**	268.54	**212.37**	26.48	**15.95**	22.65	**15.95**	518.72	**234.85**	**199.68**	234.85
4	229.78	**179.38**	224.35	**179.33**	25.37	**18.60**	21.80	**18.60**	1,175.69	**253.20**	857.06	**253.20**
5	**348.88**	441.89	**343.56**	441.83	20.79	**14.78**	17.27	**14.78**	528.35	**275.22**	216.90	**275.22**
6	246.55	**177.89**	241.23	**177.84**	27.93	**18.21**	24.43	**18.21**	481.67	**181.12**	**176.76**	181.11
7	**283.33**	359.54	**277.90**	359.49	20.56	**14.40**	17.04	**14.40**	841.24	**221.02**	535.98	**221.02**
8	352.25	**183.32**	346.96	**183.27**	15.95	**10.82**	12.44	**10.82**	849.04	**268.51**	548.80	**268.51**
9	284.51	**203.13**	278.98	**203.07**	16.89	**11.72**	13.37	**11.72**	911.93	**280.15**	594.83	**280.15**
10	179.17	**151.15**	173.60	**151.10**	504.62	**21.22**	501.03	**21.22**	1,228.29	**168.23**	916.30	**168.23**

Table 2 Maximum, minimum, mean and median feature extraction cost (in seconds) for `F2F` and `c-NN` features across 10 folds.

	Car Sequencing		Covering Array		Social Golfers	
	F2F	c-NN	F2F	c-NN	F2F	c-NN
max	33.68	0.33	8.96	0.26	1304.95	0.25
min	0.80	0.01	3.33	0.01	0.52	0.01
mean	5.38	0.05	3.61	0.01	312.23	0.01
median	6.71	0.06	3.55	0.01	271.02	0.01

significantly better than the `F2F`-based AS approaches across all folds. In fact, `(c-NN,K)` is the only one that surpasses SBS. These results further confirm the effectiveness of our proposed learning approach in capturing the semantic properties of high-level constraint instances when compared to the existing low-level `F2F` features, where essential information about the high-level structure of the problem instance can be lost during translation.

6.4 Feature Extraction Cost

We address Q3 by analysing the computational cost associated with the feature extraction process and its influence on the final AS performance. Table 1 presents average PAR10 scores on the test set of `(c-NN,K)` and `(F2F,K)` with (`+E`) and without feature extraction cost across 10 folds. While the extraction cost of `c-NN` features is marginal, it is generally higher for `F2F`, resulting in worse PAR10 scores. On Car Sequencing, feature extraction cost does not affect the ranking between the two approaches: `(c-NN,K)` performs better than `(F2F,K)` on a majority of folds (7/10) no matter whether feature extraction cost is included. On Covering Array, the `F2F` feature extraction cost is smaller in scale and does not impact the relative ranking in AS performance between `(c-NN,K)` and `(F2F,K)`, as the former is consistently better across all folds. Additionally, even when feature extraction cost is excluded, `(F2F,K)`'s performance is still worse than SBS across all folds, suggesting that those features are insufficient to capture the semantic properties of this problem. Finally, on Social Golfers, feature extraction cost of `F2F` significantly negatively impacts the corresponding AS's overall performance and alter the ranking on 3 folds, while `(c-NN,K)` achieves better score on the remaining 7 folds no matter what.

Table 2 shows the maximum, minimum, mean, and median extraction cost (in seconds) of the `F2F` and `c-NN` features. We observe a better consistency in the values of the `c-NN` features, which are marginally affected by the instance text size and always negligible. In contrast, the values of the `F2F` features vary significantly across the three problems, and they are not negligible, especially in the case of Social Golfers.

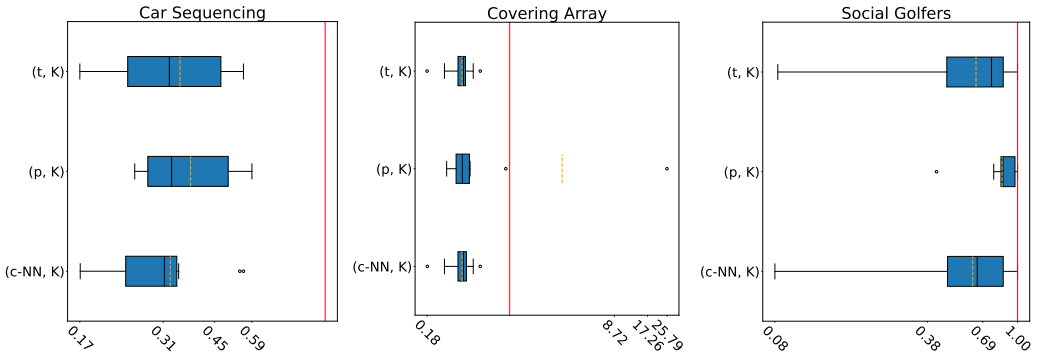


Figure 7 Comparison of the normalised PAR10 scores of K-means paired with `c-NN`, the probabilities-only features (`p`) and Tanh-only features (`t`) for the three problem classes.

6.5 Ablation Study

To address Q4, we show in Figure 7 the benefits of combining the NN output probabilities and the Tanh output for feature learning by focusing on the `(c-NN,K)` approach. Using only the output probabilities `(p,K)` significantly hinders the learning process and leads to poorer performance compared with using the combined features in `(c-NN,K)`. In contrast, when only the Tanh features are used `(t,K)`, the performance is more comparable with `(c-NN,K)`. Interestingly, for the Covering Array problem, both approaches achieve identical results. However, in the case of Car Sequencing and Social Golfers, both the average and median performance metrics indicate worse outcomes with `(t,K)` compared to `(c-NN,K)`.

7 Conclusions

We have explored automatic feature learning for algorithm selection on three problems. Our approach employs a transformer encoder to learn instance features directly from a high-level problem description, which are used to predict the best algorithm for solving an instance. Our experiments demonstrate that the learnt features can be effectively used in a diverse set of AS algorithms. In particular, a K-means-based approach performed competitively on all problem classes beating existing state-of-the-art features previously used for AS.

Our results indicate that neural-based feature extraction offers a viable and efficient alternative to traditional methods, with significantly lower computational costs for feature extraction. Furthermore, our results show the importance of extracting features from an high-level instance, especially when the AS task involves selecting the best constraint model as well as a solver.

Our study highlights the potential of ML and automatic feature learning in enhancing algorithm selection for combinatorial problems, paving the way for more adaptive and efficient solving techniques in various application domains.

In future work, we will conduct a more systematic hyper-parameter optimisation of the whole pipeline and explore other types of learning techniques, for instance by representing the problem instances in the form of a graph and using GNNs for feature extraction. We also plan to instantiate our approach on other high-level modelling languages like MiniZinc [41] and CPMpy [25], and adapt it to receive natural language in the input, with the use of LLMs. While challenging due to the the black-box nature of neural network-based approaches, an interesting direction is to analyse the learnt features and be able to explain their semantic meaning.

References

1 M. Ahmed, R. Seraj, and S. M. S. Islam. The k-means algorithm: A comprehensive survey and performance evaluation. *Electronics*, 9(8):1295, 2020.

2 Ö. Akgün, A. M Frisch, I. P Gent, C. Jefferson, I. Miguel, and P. Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artificial Intelligence*, 310:103751, 2022. doi:10.1016/J.ARTINT.2022.103751.

3 M. Alissa, K. Sim, and E. Hart. Automated algorithm selection: from feature-based to feature-free approaches. *Journal of Heuristics*, 29(1):1–38, 2023. doi:10.1007/S10732-022-09505-4.

4 R. Amadini, M. Gabbrielli, and J. Mauro. An enhanced features extractor for a portfolio of constraint solvers. In *Proceedings of the 29th annual ACM symposium on applied computing*, pages 1357–1359, 2014.

5 R. Amadini, M. Gabbrielli, and J. Mauro. SUNNY: a lazy portfolio approach for constraint solving. *Theory and Practice of Logic Programming*, 14(4-5):509–524, 2014. doi:10.1017/S1471068414000179.

6 R. Amadini, M. Gabbrielli, and J. Mauro. SUNNY-CP: a sequential CP portfolio solver. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1861–1867, 2015.

7 J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 929–947, 2024.

8 C. Ansótegui, M. Sellmann, and K. Tierney. Self-configuring cost-sensitive hierarchical clustering with recourse. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming*, pages 524–534. Springer, 2018.

9 I. Beltagy, M. E P.s, and A. Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

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

11 B. Bischl, P. Kerschke, L. Kotthoff, M. Lindauer, Y. Malitsky, A. Fréchette, H. Hoos, F. Hutter, K. Leyton-Brown, K. Tierney, and J. Vanschoren. ASlib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, 2016. doi:10.1016/j.artint.2016.04.003.

12 D. Bridge, E. O'Mahony, and B. O'Sullivan. Case-based reasoning for autonomous constraint solving. In *Autonomous search*, pages 73–95. Springer, 2012. doi:10.1007/978-3-642-21434-9_4.

13 G. Cenikj, G. Petelin, and T. Eftimov. Transoptas: Transformer-based algorithm selection for single-objective optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 403–406, 2024.

14 T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

15 Chuffed Developers. Chuffed, a lazy clause generation solver. https://github.com/chuffed/chuffed. Accessed: 2024-07-05.

16 Conjure developers. Essence catalog: A collection of problem specifications in essence. https://github.com/conjure-cp/EssenceCatalog, 2024.

17 A. Crespo Márquez. The curse of dimensionality. In *Digital Maintenance Management: Guiding Digital Transformation in Maintenance*, pages 67–86. Springer, 2022.

18 N. Dang. A portfolio-based analysis method for competition results. *arXiv preprint arXiv:2205.15414*, 2022.

19 N. Dang, Özgür Akgün, J. Espasa, I. Miguel, and P. Nightingale. A Framework for Generating Informative Benchmark Instances. In Christine Solnon, editor, *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2022.18.

20 J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

21 F. Didier, L. Perron, S. Mohajeri, S. A. Gay, T. Cuvelier, and V. Furnon. OR-Tools' vehicle routing solver: a generic constraint-programming solver with heuristic search for routing problems, 2023.

22 M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *Proceedings of the European Conference on Artificial Intelligence*, pages 290–295, 1988.

23 A. M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, and I. Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008. doi:10.1007/s10601-008-9047-y.

24 I. P Gent. Two results on car-sequencing problems. *Report University of Strathclyde, APES-02-98*, 7, 1998.

25 Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

26 K. Gurney. *An introduction to neural networks*. CRC press, 2018.

27 A. Handler. An empirical study of semantic similarity in wordnet and word2vec, 2014.

28 F. Hutter, Holger H Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization (LION)*, pages 507–523. Springer, 2011.

29 IBM. Ilog cplex optimization studio: Cplex optimizer. https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer, 2022.

30 L. Kampel and D. E Simos. A survey on the state of the art of complexity problems for covering arrays. *Theoretical Computer Science*, 800:107–124, 2019. doi:10.1016/J.TCS.2019.10.019.

31 P. Kerschke, H. H Hoos, F. Neumann, and H. Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary computation*, 27(1):3–45, 2019. doi:10.1162/EVCO_A_00242.

32 Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

33 R. Kora and A. Mohammed. A comprehensive review on transformers models for text classification. In *2023 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*, pages 1–7, 2023. doi:10.1109/MIUCC58832.2023.10278387.

34 L. Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data mining and constraint programming: Foundations of a cross-disciplinary approach*, pages 149–190. Springer, 2016. doi:10.1007/978-3-319-50137-6_7.

35 L. Kotthoff, P. Kerschke, H. Hoos, and H. Trautmann. Improving the state of the art in inexact TSP solving using per-instance algorithm selection. In *LION 9*, pages 202–217. Springer, 2015.

36 A. Krizhevsky, I. Sutskever, and G. E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25, 2012.

37 M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, D. Deng, C. B., T. Ruhkopf, R. Sass, and F. Hutter. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022. URL: http://jmlr.org/papers/v23/21-0888.html.

38 M. Lindauer, H. H Hoos, F. H., and T. Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015. doi:10.1613/JAIR.4726.

39 T. Liu, R. Amadini, M. Gabbrielli, and J. Mauro. sunny-as2: Enhancing SUNNY for algorithm selection. *Journal of Artificial Intelligence Research*, 72:329–376, 2021. doi:10.1613/JAIR.1.13116.

40 Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019. arXiv:1907.11692.

41 N. Nethercote, P. J Stuckey, R. Becket, S. Brand, G. J Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

42 P. Nightingale, Ö. Akgün, I. P Gent, C. Jefferson, I. Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, 2017. doi:10.1016/J.ARTINT.2017.07.001.

43 E. Nudelman, K. Leyton-Brown, A. Devkar, Y. Shoham, and H. Hoos. SATzilla: An algorithm portfolio for SAT. *Solver description, SAT competition*, 2004, 2004.

44 E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish conference on artificial intelligence and cognitive science*, pages 210–216, 2008.

45 Alessio Pellegrino, Özgür Akgün, Nguyen Dang, Zeynep Kiziltan, and Ian Miguel. EFE repository. Software, version 1.0., swhId: swh:1:dir:0d5708bbc3b0395ddcd80b52bbb6ed8da6ffe252 (visited on 2025-07-22). URL: https://github.com/SeppiaBrilla/EFE_project, doi:10.4230/artifacts.24086.

46 R. Qasim, W. H. Bangyal, M. A Alqarni, and A. Ali Almazroi. A fine-tuned bert-based transfer learning approach for text classification. *Journal of healthcare engineering*, 2022(1):3498123, 2022.

47 A. Rangamani, M. Lindegaard, T. GA.ti, and T. A Poggio. Feature learning in deep classifiers through intermediate neural collapse. In *International Conference on Machine Learning*, pages 28729–28745. PMLR, 2023.

48 J. R Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976. doi:10.1016/S0065-2458(08)60520-3.

49 M. V. Seiler, J. Rook, J. Heins, O. L. Preuß, J. Bossek, and H. Trautmann. Using reinforcement learning for per-instance algorithm configuration on the TSP. In *IEEE Symposium Series on Computational Intelligence*, pages 361–368. IEEE, 2023.

50 H. Shavit and H. H Hoos. Revisiting SATZilla features in 2024. In *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, pages 27–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

51 B. M Smith. Modelling. In *Foundations of Artificial Intelligence*, volume 2, pages 377–406. Elsevier, 2006. doi:10.1016/S1574-6526(06)80015-5.

52 P. Spracklen, N. Dang, Ö. Akgün, and I. Miguel. Automated streamliner portfolios for constraint satisfaction problems. *Artificial Intelligence*, 319:103915, 2023. doi:10.1016/J.ARTINT.2023.103915.

53 S. Sun, W. Chen, L. Wang, X. Liu, and T. Liu. On the depth of deep neural networks: A theoretical view. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30(1), March 2016. doi:10.1609/aaai.v30i1.10243.

54 M. Triska and N. Musliu. An effective greedy heuristic for the social golfer problem. *Annals of Operations Research*, 194(1):413–425, 2012. doi:10.1007/S10479-011-0866-7.

55 F. Ulrich-Oltean, P. Nightingale, and J. A. Walker. Selecting SAT encodings for pseudo-boolean and linear integer constraints. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming*. LIPICS, 2022.

56 F. Ulrich-Oltean, P. Nightingale, and J. A. Walker. Learning to select SAT encodings for pseudo-boolean and linear integer constraints. *Constraints*, 28(3):397–426, 2023. doi:10.1007/S10601-023-09364-1.

57 F. Ulrich-Oltean, P. Nightingale, and J. A. Walker. IndiCon: Selecting SAT encodings for individual pseudo-boolean and linear integer constraints. In *2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 36–42. IEEE, 2024.

58 M. Vallati, L. Chrpa, and D. Kitchin. ASAP: an automatic algorithm selection approach for planning. *International Journal on Artificial Intelligence Tools*, 23(06):1460032, 2014. doi:10.1142/S021821301460032X.

59 A Vaswani. Attention is all you need. *Advances in Neural Information Processing*, 30, 2017.

60 L. Xu, F. Hutter, Holger H Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008. doi:10.1613/JAIR.2490.

61 L. Xu, F. Hutter, J. Shen, H. H Hoos, and K. Leyton-Brown. SATzilla2012: improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge*, pages 57–58, 2012.

A Neural Network Training

A.1 Model Hyper-parameters

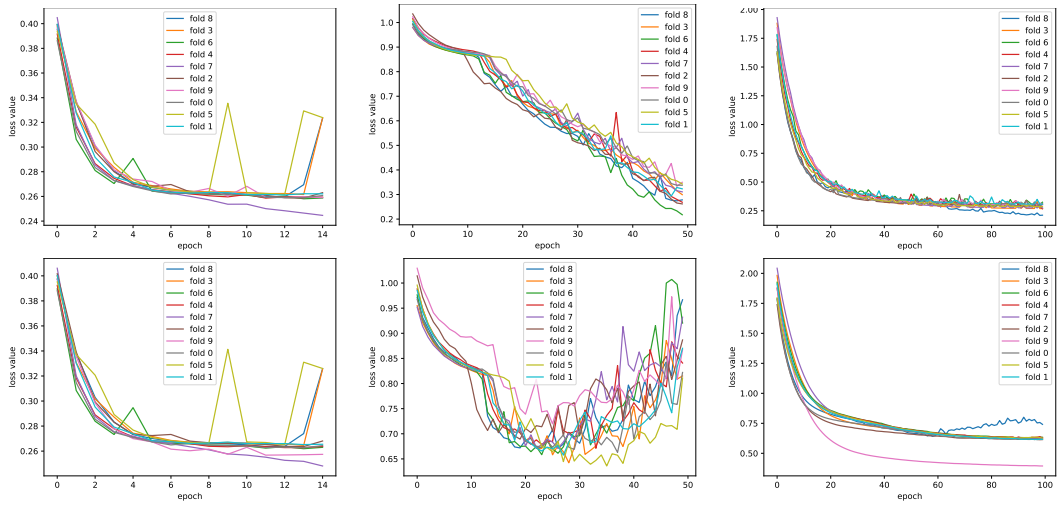
For both the B-NN and the C-NN models, we need to set the sizes of the feature and post-feature layers as they are arbitrary and their values can significantly impact both the size of the models and their ability to learn the task effectively. After some manual tuning, we set the size of the feature layer to 100 neurons and the size of the post-feature layer to 200 neurons. These values are large enough to allow a lot of flexibility in the model's parameters while still being small enough not to cause over-parametrization problems (for the post-feature layer) or issues with the external ASs (for the feature layer).

A.2 B-NN Feature Learning

For each test case, the model was trained using distinct hyper-parameter configurations, including variations in learning rates, the total number of training epochs, and batch sizes. Since we have a limited amount of training data, we observe signs of over-fitting in a number of cases. Therefore, we monitor the validation performance throughout training and retain the model that achieves the best loss value on the validation set.

Figure 8a illustrates the evolution of the loss values for both the training and validation sets at the end of each epoch for the models trained on the Car Sequencing instances. The models were trained for a total of 15 epochs using a learning rate of 7×10^{-6}. Due to memory constraints, the batch size was limited to only two elements per batch. The loss trends for the training and validation sets are generally consistent, with both following a similar overall trajectory. With the exception of a single case, the majority of networks exhibit diminishing improvements after the 6^{th} epoch. In certain instances, the loss value increases during the final epochs, suggesting overfitting. Notably, the fifth fold displays the poorest performance, characterised by a more unstable trend and a substantially higher final loss value.

Figure 8b depicts the loss values for the training and validation sets at the end of each epoch for the models trained on the Covering Array instances. The models were trained for a total of 100 epochs with a learning rate of 1×10^{-5} and a batch size of 32 items per batch. Among the case studies analysed, the models trained on the Covering Array exhibited the least favourable trend. The loss curves reveal a pronounced tendency toward overfitting, as the loss for the training set continues to decrease while the validation loss stagnates or



(a) Car Sequencing. **(b)** Covering Array. **(c)** Social Golfers.

Figure 8 Loss values for the B-NN model across all folds, shown for training (top) and validation (bottom) sets.

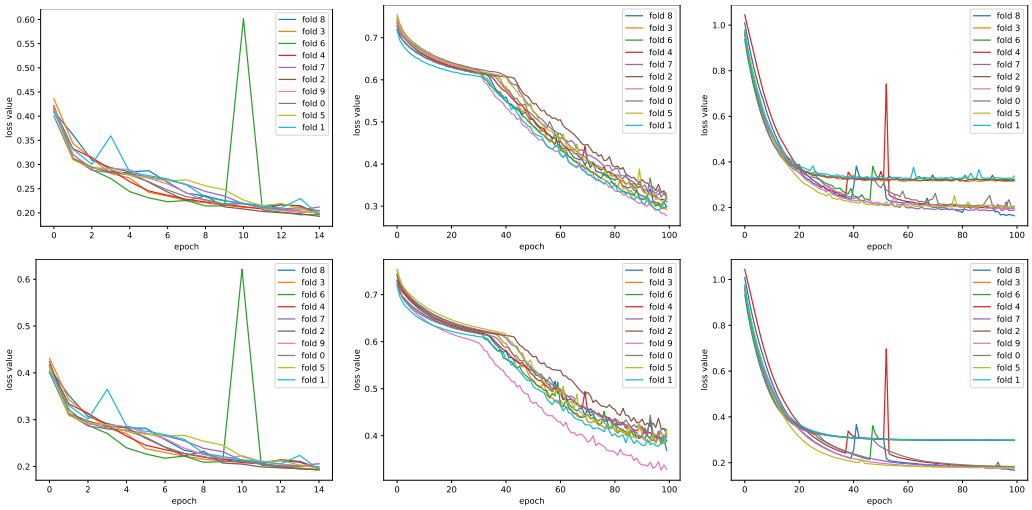
increases after approximately 25 epochs. This behaviour suggests that the models struggle to generalize effectively, indicating that either the current network architecture is suboptimal for this task or that the nature of the instances poses intrinsic challenges for this type of neural network model.

Figure 8c illustrates the loss values for the training and validation sets at the end of each epoch for the models trained on the Social Golfers instances. The models for this instances were trained for 50 epochs using a learning rate of 6×10^{-6} and a batch size of 32 items per batch. Similar to the trends observed in the Car Sequencing models, the loss values for the training and validation sets follow comparable trajectories. However, unlike the Car Sequencing models, the loss in the Social Golfers models continues to improve, albeit at a slower rate, after stabilization around the 20th epoch. Two folds exhibit distinct behaviours worth highlighting: fold 9 achieves a notably lower validation loss compared with the other folds, while fold 8 demonstrates signs of overfitting beginning around the 60th epoch, with its validation loss subsequently increasing, ultimately leading to the worst performance among all folds.

A.3 C-NN Feature Learning

These models share most of their characteristics with the B-NN model. However, instead of using the Cross-Entropy loss function, the Binary Cross-Entropy (BCE) loss function was employed due to the nature of the task, which involved multilabel classification. The optimiser remained the Adam optimizer. Each instances was trained using a distinct set of hyper-parameters. Furthermore, model performance was evaluated on the validation set at the end of each epoch. To mitigate the risk of overfitting, the model with the lowest validation loss was saved during training.

Figure 9a illustrates the evolution of the loss values for the training and validation sets at the end of each epoch for the models trained on the Car Sequencing instances. For these instances, the models were trained for 15 epochs with a learning rate of 7×10^{-6} and a batch size of 2 items per batch, a constraint imposed by memory limitations similar to the B-NN model. The overall loss trend remains consistent across all folds, with a similar



(a) Car Sequencing. **(b)** Covering Array. **(c)** Social Golfers.

Figure 9 Loss values for the C-NN model across all folds, shown for training (top) and validation (bottom) sets.

trajectory observed for both the training and validation sets. Unlike the B-NN model, where the training process often stagnates or slows down after a certain number of epochs, this model maintains a steady reduction in loss throughout all epochs. Two folds demonstrate particularly noteworthy behaviour: fold 8 experiences a small but noticeable loss spike at the 4th epoch, while fold 6 exhibits a more significant spike at the 11th epoch. Interestingly, in both cases, the loss subsequently returns to a downward trend, and the spikes are observed in both the training and validation sets.

Figure 9b illustrates the loss values for the training and validation sets at the end of each epoch for the models trained on the Covering Array instances using the C-NN model. The models for these instances were trained for 100 epochs with a learning rate of 7×10^{-6} and a batch size of 32 items per batch. The loss curves for these instances are particularly unconventional. All folds exhibit a relatively slow start, with the loss decreasing at a sluggish pace during the initial epochs, followed by a more rapid convergence in subsequent epochs. This pattern is observed in both the training and validation sets, with all folds reaching a similar loss value by the end of training. Interestingly, only the 9th fold displays a slightly lower final validation loss than the other folds. This consistent convergence across folds suggests that the peculiar shape of the loss curves is likely a result of the inherent characteristics of the data: or the loss landscape, rather than fold-specific anomalies.

Figure 9c illustrates the evolution of the loss values for the training and validation sets at the end of each epoch for the models trained on the Social Golfers instances using the C-NN model. The models for this instances were trained for 100 epochs with a learning rate of 9×10^{-6} and a batch size of 32 items per batch. The loss values for the Social golfers instances follow a trend similar to that observed in the B-NN models. Specifically, there is a rapid convergence in the early epochs, after which the rate of improvement slows. Notably, the validation loss is more stable in the C-NN model compared to its B-NN counterpart. However, similar to the B-NN models, the C-NN model exhibits a few spikes in loss during training. Unlike in the B-NN model, these spikes occur at approximately the same epochs across multiple folds, suggesting that they may be due to the structure of the loss landscape rather than the idiosyncratic behaviour of specific folds.

BFS-Based Canonical Codes for Generating Graphs with Constraint Programming

Xiao Peng
LAAS-CNRS, Université de Toulouse, Toulouse, France

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

Abstract

We consider the problem of generating all graphs that satisfy some given additional constraints (on vertex degrees, or cycle lengths, for example). Most previous works have proposed to generate canonical codes associated with adjacency matrices. In this paper, we consider canonical codes based on Breadth First Search (BFS), and we show how to generate them with Constraint Programming (CP): we introduce a set of basic constraints that must be satisfied by all canonical codes, thus breaking many symmetries, and we introduce a global constraint to break other symmetries. We illustrate the interest of our approach on connected claw-free cubic graphs, and show that it outperforms state-of-the-art CP and SAT Modulo Theory (SMT) approaches.

2012 ACM Subject Classification Mathematics of computing → Graph enumeration; Computing methodologies → Artificial intelligence

Keywords and phrases Graph Generation, Automorphisms, Symmetry Breaking

Digital Object Identifier 10.4230/LIPIcs.CP.2025.32

1 Introduction

In combinatorics and graph theory, graph enumeration involves systematically searching for all graphs that satisfy some given properties (such as a specific number of edges, vertex degree constraints, or structural features like girth). This has numerous applications in fields like chemical informatics and drug discovery, where molecules are modeled as graphs [5, 19]. To prevent us from generating isomorphic graphs, *i.e.*, graphs that are equivalent up to a renaming of their vertices, we may use canonical codes, *i.e.*, words associated with graphs such that two graphs are isomorphic if and only if they have the same canonical code: instead of enumerating graphs, we enumerate canonical codes, thus ensuring that all generated graphs are non-isomorphic.

Canonical codes may be based on adjacency matrices, *i.e.*, 2-dimensional arrays M such that $M_{ij} = 1$ if (i, j) is an edge, and 0 otherwise. A code may be obtained from M by concatenating its rows (or columns), thus obtaining a binary word. Since different vertex permutations may produce different adjacency matrices and, therefore, different codes, the lexicographically smallest one is selected as the canonical code, thus uniquely identifying the graph. To avoid exhaustively considering all vertex permutations, symmetry-breaking constraints may be used. In [4], Codish et al. introduce symmetry-breaking constraints that enforce a non-descending row order in the adjacency matrix and ensure minimality under vertex permutations within specific partitioned sets.

The Nauty algorithm [13] generates a canonical labeling through an iterative partition refinement process, where vertices are colored based on their connectivity. Once a discrete partition is reached, it serves as the canonical label. Although there is no direct work that enumerates all canonical codes using Nauty's labeling as constraints, as far as we know, the concept of symmetry breaking via structural graph information has been applied by Codish et al. [3]. They encode partition refinement into constraints, and introduce additional constraints

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

to maintain minimality in the adjacency matrix while preserving partition structure. This approach provides insights into constructing compact and complete symmetry-breaking constraints, effectively reducing the need to evaluate all vertex permutations. Further studies on complete symmetry breaking are explored in [8, 9].

In addition to the adjacency matrix, Codish et al. demonstrate in [10] how various higher-dimensional graph invariants, such as vertex degrees and the cardinality of common neighbors, can be leveraged to define symmetry-breaking constraints. They compare different combinations of these invariants to evaluate their effectiveness in reducing the search space.

Beyond *static symmetry breaking*, which relies on a predefined set of permutations, Kirchweger and Szeider introduce in [11] a novel SMT-based approach for graph generation using *dynamic symmetry breaking*. The idea is to detect symmetries in the partially generated graph to enforce adjacency matrix minimality during the solving process, thus significantly improving efficiency compared to static symmetry-breaking methods.

Another class of canonical codes relies on graph traversals, where codes are sequences of traversed edges. Graph mining algorithms, such as Gaston [15] and gSpan [20], are usually based on Depth-First-Search (DFS). In [18] and [16], canonical codes based on Breadth-First-Search (BFS) are used for particular classes of graphs (Deterministic Finite Automata in [18] and hexagonal graphs in [16]) for which graph isomorphism may be solved in polynomial-time. In [14], some BFS-based symmetry-breaking predicates are introduced for generating connected graphs.

In this paper, we extend the work of [14] and introduce new symmetry breaking constraints for BFS-based canonical codes. Indeed, BFSs naturally construct spanning trees, and the orbits formed by the vertices within a spanning tree can be exploited to derive compact symmetry breaking constraints, that are efficiently handled by CP solvers. Also, every prefix of our canonical code is a canonical code, thus enabling the construction of dynamic symmetry-breaking constraints, similar to the approach in [11].

The paper is organized as follows. In Section 2, we introduce notations. In Section 3, we introduce BFS-based codes, and define a canonical BFS-based code as the smallest possible code. In Section 4, we describe a CP model for generating BFS-based codes. In Sections 5 and 6, we study properties of canonical codes. In Section 7, we introduce a global constraint that exploits these properties to ensure canonicity. In Section 8, we evaluate our method for generating connected claw-free cubic graphs, and compare it with the approach of [10], which relies on an adjacency matrix representation, as well as with the SMT-based approach of [11] and with Nauty [13]. We also introduce a more compact graph representation for this benchmark, thus enabling us to solve larger instances.

2 Notations and definitions

We note $[i, j]$ the set of all integers ranging from i to j, $\#S$ the cardinality of a set S, and $\preceq_{lex}$ the lexicographic order for comparing sequences of integer values.

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there exists a bijection $f : V_1 \to V_2$ that preserves edges, *i.e.*, $\forall u, v \in V_1, (u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$. This problem belongs to $\mathcal{NP}$, but it is not known to be in $\mathcal{P}$ nor to be $\mathcal{NP}$-complete, and it is conjectured to be $\mathcal{NP}$-intermediate.

Throughout this paper, we consider a connected graph $G = (V, E)$ such that $\#V = n + 1$ and $\#E = n + m$, so that $0 \leq m \leq \frac{(n-2)(n-1)}{2}$. We note d_i the degree of a vertex $i \in V$.

Algorithm 1 BFS_G.

```
   Input: A graph G = (V, E)
   Output: A code associated with a BFS traversal of G
 1 for each vertex v ∈ V do initialise num[v] to -1;
 2 choose a first vertex v_0 ∈ V, set num[v_0] to 0 and add v_0 to an empty FIFO queue q;
 3 initialize i to 0, and c_f and c_b to empty sequences;
 4 while q is not empty do
 5 |   pop u from q;
 6 |   for each vertex v adjacent to u do
 7 |   |   if num[v] < 0 then // Forward edge
 8 |   |   |   add num[u] at the end of c_f, push v in q, set num[v] to i, and increment i;
 9 |   |   else if num[u] < num[v] then // Backward edge
10 |   |   |   add num[u] and num[v] at the end of c_b;
11 |   |   end
12 |   end
13 end
14 return the concatenation of c_f and c_b
```

3 Canonical BFS-based code

A code is a sequence of $n + 2m$ integer values, generated through a BFS of G as described in Algorithm 1. We use a FIFO queue q to store the vertices that have been discovered but not yet treated, and we build two sequences c_f and c_b of n and $2m$ integer values, respectively. When a vertex v is pushed in q, we store in $num[v]$ its q-number, which ranges from 0 for the first pushed vertex to n for the last pushed one. At each iteration of the while loop (Lines 4-13), a vertex u is popped from q, and we consider each vertex v adjacent to u.

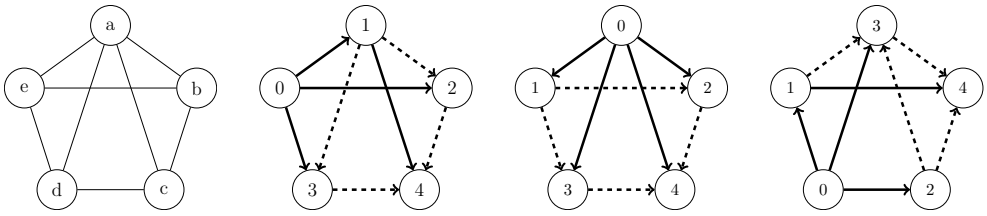
- If v is reached for the first time (Lines 7-8), $(num[u], num[v])$ is said to be a *forward edge*. $num[u]$ is the parent of $num[v]$, and it is added at the end of c_f. At the end of Algorithm 1, there are n forward edges which form a spanning tree of G.
- If v has already been reached (Lines 9-10), $(num[u], num[v])$ is said to be a *Backward edge*, and $num[u]$ and $num[v]$ are added at the end of c_b. To avoid treating twice a same edge, we add the condition that $num[u]$ must be smaller than $num[v]$.

The final code is obtained by concatenating c_f and c_b (Line 14). Hence, a code is a sequence of $n + 2m$ integer values $p_1, p_2, \ldots, p_n, x_1, y_1, x_2, y_2, \ldots x_m, y_m$ such that for each $i \in [1, n]$, (p_i, i) is a forward edge and for each $i \in [1, m]$, (x_i, y_i) is a backward edge. The first part of a code, from p_1 to p_n is called *forward code*, whereas the second part of the code, from x_1 to y_m is called *backward code*. When there is no ambiguity, commas between vertices in codes may be removed.

Different codes may be built for a same graph, depending on (i) which vertex is chosen first (Line 2) and (ii) the order considered to visit successors of u (Line 6). In order to make the algorithm deterministic, we add a parameter π which is a permutation of V used to break ties: Line 2, the chosen vertex v_0 is the first vertex of π; Line 6, successors of u are visited in the order defined by π. The code obtained when using π to break ties is denoted $\text{BFS}_G(\pi)$.

▶ **Example 1.** Let us consider the graph G displayed in Fig. 1 when the permutation π is *eabcd*, as displayed in Fig. 1(i). The first vertex pushed in q is e, and we have $num[e] = 0$. When e is popped from q, we iterate on its adjacent vertices in the order defined by π:

- when $v = a$, we set $num[a]$ to 1 and add $num[e] = 0$ to c_f, *i.e.*, $(0, 1)$ is a forward edge;
- when $v = b$, we set $num[b]$ to 2 and add $num[e] = 0$ to c_f, *i.e.*, $(0, 2)$ is a forward edge;



G (i) $\pi = eabcd$, $c_f = 0001$, $c_b = 12, 13, 24, 34$ (ii) $\pi = aebdc$, $c_f = 0000$, $c_b = 12, 13, 24, 34$ (iii) $\pi = decba$, $c_f = 0001$, $c_b = 13, 23, 24, 34$

Figure 1 A graph G (on the left), and three BFS-based codes corresponding to three different permutations π. Vertices of the three graphs on the right are labeled with their q-numbers; forward-edges are displayed in solid lines; backward-edges are displayed in dotted lines.

- when $v = d$, we set $num[d]$ to 3 and add $num[e] = 0$ to c_f, *i.e.*, $(0, 3)$ is a forward edge.

When a is popped from q, we iterate on its adjacent vertices in the order defined by π:
- when $v = e$, we do nothing because edge (e, a) has already been treated;
- when $v = b$, we add $num[a] = 1$ and $num[b] = 2$ in c_b, *i.e.*, $(1, 2)$ is a backward edge;
- when $v = c$, we set $num[c]$ to 4 and add $num[a] = 1$ to c_f, *i.e.*, $(1, 4)$ is a forward edge;
- when $v = d$, we add $num[a] = 1$ and $num[d] = 3$ in c_b, *i.e.*, $(1, 3)$ is a backward edge.

When b is popped from q, we iterate on its adjacent vertices in the order defined by π:
- when $v = e$ or a, we do nothing because edges (e, b) and (a, b) have already been treated;
- when $v = c$, we add $num[b] = 2$ and $num[c] = 4$ in c_b, *i.e.*, $(2, 4)$ is a backward edge.

When d is popped from q, we iterate on its adjacent vertices in the order defined by π:
- when $v = e$ or a, we do nothing because edges (e, d) and (a, d) have already been treated;
- when $v = c$, we add $num[d] = 3$ and $num[c] = 4$ in c_b, *i.e.*, $(3, 4)$ is a backward edge.

When c is popped from q, we iterate on its adjacent vertices in the order defined by π, *i.e.*, a, b, and d, but we do nothing as edges (a, c), (b, c), and (d, c) have already been treated. Hence $\text{BFS}_G(eabcd) = 0001, 12, 13, 24, 34$.

Given a code, we can build the corresponding graph as the code allows us to reconstitute all its edges. For example, given $\text{BFS}_G(eabcd) = 0001, 12, 13, 24, 34$, we reconstitute the set of edges $\{(0, 1), (0, 2), (0, 3), (1, 4), (1, 2), (1, 3), (2, 4), (3, 4)\}$.

As shown in Example 1, there exist different possible codes for G, depending on the considered permutation π. We define a total order on the set of all possible codes that may be associated with G by considering a lexicographic order. Among all the possible codes for G, the smallest one according to this order is called the *canonical code* of G and it is unique.

▶ **Definition 2** (Canonical code $cc(G)$)**.** *The canonical code of a graph $G = (V, E)$ is defined as $cc(G) = min\{BFS_G(\pi) : \pi$ is a permutation of $V\}$ when considering the lexicographic order $\preceq_{lex}$ to compare codes.*

In the example of Figure 1, the canonical code is $0000, 12, 13, 24, 34$.

An important property to allow an efficient enumeration of canonical codes is that any prefix of a canonical code is a canonical code: this property allows us to stop completing a code whenever its prefix is not canonical.

▶ **Theorem 3.** *Let $c = p_1 p_2 \dots p_n, x_1 y_1, \dots, x_m y_m$ be a canonical code.*
- *For each $k \in [1, n]$, the prefix $c' = p_1 \dots p_k$ is a canonical code.*
- *For each $k \in [1, m]$, the prefix $c' = p_1 \dots p_n, x_1 y_1, \dots, x_k y_k$ is a canonical code.*

Proof. Let G' be the subgraph of G that contains all edges defined by c'. Assume, for contradiction, that c' is not canonical, *i.e.*, there exists another lexicographically smaller code c'' that represents the same graph G'. We consider two cases.

Recall that our canonical code, derived from a BFS traversal, first lists the forward edges followed by the backward edges. In the first case, c' consists only of forward edges, implying that G' forms a tree. Since c'' is a valid traversal of G', it can be extended into a full traversal of G by discovering the remaining vertices, and results in a code smaller than c, which violates the canonicity of c. In the second case, c' includes backward edges. If the forward edge code $p_1 p_2 \dots p_n$ is already canonical, then the backward edges must follow a unique order, as dictated by lines 6-12 of Algorithm 1. A smaller c' would be extended to a full code smaller than c by enumerating the remaining backward edges.

Thus, in both cases, we reach a contradiction, proving that every prefix of a canonical code is also canonical. ◀

4 Basic CP model

We propose to generate codes with CP, thus allowing one to easily add other application-dependent constraints. Our CP model has the following integer variables:

- for each $i \in [1, n]$, p_i corresponds to the parent of vertex i in the spanning tree (in other words, (p_i, i) is a forward edge), and its domain is $[0, i-1]$ because the parent of i must have been discovered before i;
- for each $j \in [1, m]$, x_j and y_j correspond to the two endpoints of the jth backward edge and their domain is $[1, n]$ because an edge incident to 0 cannot be backward;
- for each $i \in [0, n]$, d_i corresponds to the degree of vertex i, and its domain is $[1, n-1]$.

The constraints are listed in Fig. 2. Constraints C1 to C7 are satisfied by any BFS-based code, even if it is not canonical:

- C1 is a consequence of the fact that the vertex whose q-number is 1 has been pushed in q just after 0 and, therefore, its parent can only be 0.
- C2 and C4 are consequences of the fact that, at each iteration of the while loop (Lines 4-13), the vertex u popped from q has a q-number increased by one (as vertices are pushed in q by increasing q-number).
- C3 is a consequence of the condition Line 9.
- C5 expresses the fact that, for each backward edge (x_i, y_i), the parent of y_i has been discovered before x_i. Indeed, let us suppose that this is not the case, *i.e.*, $p_{y_i} > x_i$. In this case, the parent of y_i would necessarily be x_i because y_i would not yet have been discovered when exploring the vertices adjacent to x_i (Lines 7-8).
- C6 and C7 relate degrees with edge variables.

C1: $p_1 = 0$
C2: $\forall i \in [2, n], p_{i-1} \leq p_i$
C3: $\forall i \in [1, m], x_i < y_i$
C4: $\forall i \in [1, m-1], x_i \leq x_{i+1}$
C5: $\forall i \in [1, m], p_{y_i} < x_i$
C6: $d_0 = \#\{i \in [1, n] | p_i = 0\}$
C7: $\forall i \in [1, n], d_i = 1 + \#\{j \in [1, n] | p_j = i\} + \#\{j \in [1, n] | x_j = i\} + \#\{j \in [1, n] | y_j = i\}$
C8: $\forall i \in [1, n], d_0 \geq d_i$
C9: $\forall i \in [1, m-1], x_i < x_{i+1} \vee (x_i = x_{i+1} \wedge y_i < y_{i+1})$
C10: $\forall i \in [1, n-1], p_i = p_{i+1} \Rightarrow \#\{j \in [1, n] | p_j = i\} \geq \#\{j \in [1, n] | p_j = i+1\}$

Figure 2 Basic CP model for generating BFS-based codes.

Constraints C8 to C10 prevent us from generating some non-canonical codes, *i.e.*, codes that are not the smallest possible ones:

- C8 comes from the fact that c starts with d_0 occurrences of 0: if there exists a vertex i such that $d_i > d_0$, then a smaller code is obtained by starting BFS from the vertex.
- If C9 is not satisfied, then a smaller code is obtained by exchanging $x_i y_i$ and $x_{i+1} y_{i+1}$.
- if C10 is not satisfied, then a smaller code is obtained by visiting $i+1$ before i.

We have experimentally compared this first CP model with the model introduced in [14] on a toy problem that aims at enumerating all graphs with k vertices and $2k-2$ edges. Both models have been implemented in Choco, and compute the same sets of solutions. However, our model is more efficient: when $k = 5$ (resp. 6, 7, and 8), it needs 0.02 (resp. 0.1, 0.4, and 4.3) seconds whereas the model of [14] needs 0.04 (resp. 0.2, 1.4, and 28.4) seconds. This comes from the fact that (i) we order sibling vertices with respect to the number of children (thanks to constraint C10) instead of sub-tree weights, and (ii) we explicitly model backward edges with x_j and y_j variables instead of using an adjacency boolean matrix.

Constraints C1 to C10 allow us to enumerate all canonical BFS-based codes, but some non-canonical codes may also be enumerated. Hence, in the next two sections we introduce two additional properties that must be satisfied by canonical codes, and that are used to propagate a global constraint that ensures the canonicity of BFS-based codes, as described in Section 7.

5 Breaking Symmetries of Forward Codes with Vertex Labels

In this section, we introduce a property which is used in Section 7 to ensure the canonicity of prefixes of forward codes. Forward code prefixes correspond to trees, and the tree isomorphism problem may be solved in polynomial time by computing vertex labels [1]. We exploit similar labels but, unlike the original algorithm of [1], we do not rename labels with integer values at each level of the tree, but define the label of a vertex i as a sequence that contains an integer value for each vertex in the subtree rooted in i. This allows us to compute labels in a more incremental way during the search.

▶ **Definition 4** (Vertex label). *Let $c_f = p_1 p_2 \dots p_k$ with $k \leq n$ be the prefix of a forward code. Let T be the tree associated with this prefix. For every vertex $i \in [0, k]$, let T_i be the subtree of T rooted at i, and Γ_i be the sequence of all vertices in T_i ordered by increasing value. The label of i, denoted l_i, is the sequence obtained by replacing every vertex j in Γ_i by $\#Ch_{c_f}(j)$ where $\#Ch_{c_f}(j)$ is the number of children of j in the tree defined by the forward code c_f.*

For example, let us consider the tree T displayed on the left of Fig. 3. The forward code associated with this tree is $c_f = 000113$. When $i = 1$, we have $\Gamma_1 = 145$ and $l_1 = 200$ because $\#Ch_{c_f}(1) = 2$ and $\#Ch_{c_f}(4) = \#Ch_{c_f}(5) = 0$. When $i = 0$, we have $\Gamma_0 = 0123456$ and $l_0 = 3201000$ because $\#Ch_{c_f}(0) = 3$, $\#Ch_{c_f}(1) = 2$, $\#Ch_{c_f}(3) = 1$, and $\#Ch_{c_f}(2) = \#Ch_{c_f}(4) = \#Ch_{c_f}(5) = \#Ch_{c_f}(6) = 0$.

Given the label l_0 of the root of a tree, we can reconstitute this tree as each value at position i in l_0 gives the number of children of vertex i. Hence, there is a bijection between each forward code and the label of the root of its associated tree. For example, we display in Fig. 3 two isomorphic trees T and T'. T corresponds to the forward code 000113 and the label of its root is 3201000, whereas T' corresponds to the forward code 000112 and the label of its root is 3210000.

An interesting property of vertex labels is that the children of a vertex in the tree associated with a canonical forward code prefix have non-decreasing labels, as stated below.

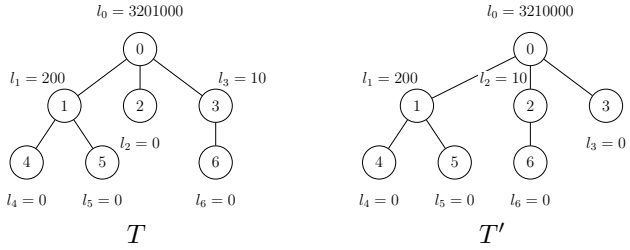


Figure 3 Vertex labels of two isomorphic trees T and T'.

▶ **Property 1.** *Let $c_f = p_1 p_2 \dots p_k$ with $k \leq n$ be the prefix of a canonical forward code. We have: $\forall i \in [1, k-1], p_i = p_{i+1} \Rightarrow l_i \succeq_{lex} l_{i+1}$.*

Proof. Let T_i and T_{i+1} be the subtrees rooted at i and $i+1$, respectively. The level of a vertex in T_i (resp. T_{i+1}) is its distance to the root i (resp. $i+1$). Let $\Gamma_i = q_1 q_2 \dots q_{\#T_i}$ be the ordered sequence of vertices in T_i, and $\Gamma_{i+1} = q'_1 q'_2 \dots q'_{\#T_{i+1}}$ be the ordered sequence of vertices in T_{i+1}. Since $p_i = p_{i+1}$, i and $i+1$ are sibling. Therefore, vertices at level k in T_i are discovered earlier than those at level k in T_{i+1}. Suppose $l_i \prec_{lex} l_{i+1}$ by contradiction, *i.e.*, $\#Ch_{c_f}(q_1) \dots \#Ch_{c_f}(q_{\#T_i}) \prec_{lex} \#Ch_{c_f}(q'_1) \dots \#Ch_{c_f}(q'_{\#T_{i+1}})$. This can occur either when (i) l_i is a prefix of l_{i+1} or (ii) it exists $c \in [1, \#T_i]$ such that $\forall j \in [1, c-1], \#Ch_{c_f}(q_j) = \#Ch_{c_f}(q'_j) \wedge \#Ch_{c_f}(q_c) < \#Ch_{c_f}(q'_c)$. In both cases, if we swap the order of i and $i+1$, then all vertices in Γ_{i+1} will be explored before those in Γ_i at the same level, resulting in a larger label l_0, which contradicts the canonicity of c_f. ◀

6 Breaking Symmetries of Backward Edges

Given a forward code prefix, we propose to exploit automorphisms in the tree associated with this prefix in order to break symmetries on backward edges. An automorphism of a tree $T = (V_T, E_T)$ is a permutation σ of V_T which is an isomorphism from T to T. Given a forward code prefix c_f, we only consider automorphisms that are still automorphisms on all possible extensions of c_f to full canonical forward codes, as defined below.

▶ **Definition 5** ($Aut(c_f)$). *Let $c_f = p_1 \dots p_k$ with $k \leq n$ be a prefix of a canonical forward code. $Aut(c_f)$ is the set of all automorphisms $\sigma : [1, k] \to [1, k]$ that are still valid on all canonical extensions of c_f,* i.e., *for any sequence $p_{k+1} \dots p_n$ such that $p_1 \dots p_n$ is canonical, the automorphism σ' such that $\forall i \in [1, k], \sigma'(i) = \sigma(i)$ and $\forall i \in [k+1, n], \sigma'(i) = i$ is an automorphism of the tree associated with $p_1 \dots p_n$,* i.e., *$\sigma' \in Aut(p_1 \dots p_n)$.*

For example, let us consider the tree T' displayed in Fig. 3. The canonical code of T' is 000112. Let us assume that $n = 9$, *i.e.*, we still have to add three forward edges to the tree. Let σ_{45} be the automorphism that only exchanges vertices 4 and 5 and leaves unchanged all other vertices, *i.e.*, $\sigma_{45}(4) = 5, \sigma_{45}(5) = 4$, and $\forall i \notin \{4, 5\}, \sigma_{45}(i) = i$. σ_{45} does not belong to $Aut(000112)$. Indeed, there exist extensions of 000112 for which σ_{45} is not an automorphism. For example, if we add the forward edges $(3, 7)$ and $(4, 8)$ to T', leading to the forward code 00011234, then σ_{45} is no longer an automorphism because the subtree rooted in 4 is no longer isomorphic to the subtree rooted in 5.

Now, let us consider the tree obtained from T' by adding the forward edge $(6, 7)$. The associated forward code is 0001126. In this case, σ_{45} belongs to $Aut(0001126)$ because it is still an automorphism on all canonical extensions of 0001126. Indeed, 4 and 5 cannot be parent (because 6, which is at the same level as 4 and 5 but with a larger q-number, already has a child).

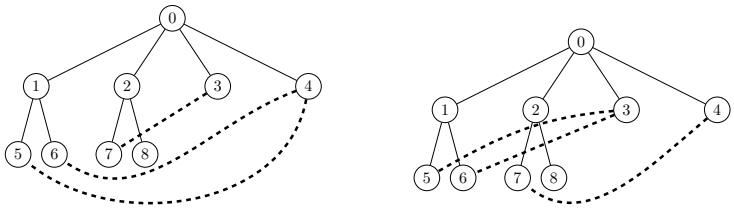


Figure 4 Example of two isomorphic graphs (backward edges are displayed with dotted lines). The code of the left-hand graph is $00001122, 37, 45, 46$ and it is not canonical because we obtain a smaller code (*i.e.*, $00001122, 35, 36, 47$) when exchanging 3 with 4, as displayed on the right.

The following property allows us to exploit automorphisms to break symmetries on backward edges.

▶ **Property 2.** *Let $c = p_1 \dots p_n, x_1y_1, \dots, x_my_m$ be a canonical code of G. The following property holds for any $k \in [1, n]$ and any $l \in [1, m]$ such that $\forall i \in [1, l], x_i, y_i \in [1, k]$:*
$\forall \sigma \in Aut(p_1 \dots p_k), x_1y_1 \dots x_ly_l \preceq_{lex} \sigma(x_1y_1 \dots x_ly_l)$
where $\sigma(x_1y_1 \dots x_ly_l)$ is obtained from $x_1y_1 \dots x_ly_l$ in two steps: (i) we compute the set of edges $S = \{(min\{\sigma(x_i), \sigma(y_i)\}, max\{\sigma(x_i), \sigma(y_i)\}) | i \in [1, l]\}$, and (ii) we sort the set S of edges in lexicographically ascending order and concatenate them into a sequence of $2l$ values.

Proof. By definition, when applying σ to the tree T associated with $p_1 \dots p_n$, we obtain a tree which is isomorphic to T and, therefore, has the same canonical code as T. For contradiction, let us assume that there exists $l \in [1, m]$ such that $x_1y_1 \dots x_ly_l \succ_{lex} \sigma(x_1y_1 \dots x_ly_l)$. In this case, the code obtained from c by replacing $x_1y_1 \dots x_ly_l$ with $\sigma(x_1y_1 \dots x_ly_l)$ is lexicographically smaller than c, which is in contradiction with the fact that c is canonical. ◀

For example, let us consider the graph G displayed in the left of Fig. 4. The permutation σ_{34} that exchanges 4 with 3 and leaves unchanged all other vertices belongs to $Aut(00001122)$. The code of G is $00001122, 37, 45, 46$. In Step 1, we compute the set S of edges obtained when applying the permutation σ_{34} to $\{(3, 7), (4, 5), (4, 6)\}$, *i.e.*, $S = \{(4, 7), (3, 5), (3, 6)\}$. Then, we sort edges of S in lexicographic order to obtain the sequence $35, 36, 47$ which is lexicographically smaller than $37, 45, 46$. Hence, $00001122, 37, 45, 46$ is not canonical.

7 Global *CanonicalCode* Constraint

Properties 1 and 2 are used to propagate the global constraint defined below.

▶ **Definition 6.** *Let $n \geq 1$ and $m \geq 0$ be integer values, and $\forall i \in [1, n], \forall j \in [1, m], p_i, x_j$, and y_j be integer variables. The global constraint $cc(p_1, \dots, p_n, x_1, y_1, \dots, x_m, y_m)$ is satisfied iff $p_1 \dots p_n x_1 y_1 \dots x_m y_m$ is a BFS-based canonical code.*

Propagation of Property 1

We exploit Property 1 to ensure that the forward code is canonical: When the domain of a variable p_k with $k \leq n$ is reduced to a singleton, if the domain of p_j is also reduced to a singleton for each $j \in [1, k-1]$, and if there exists $i \in [1, k-1]$ such that $p_i = p_{i+1}$ and $l_i \prec_{lex} l_{i+1}$, then a failure is raised. Note that this ensures that the forward code is the smallest possible one when starting the search from vertex 0. However, if there exists a vertex $i \in [1, k]$ such that $1 + \#Ch_{p_1 \dots p_k}(i) = \#Ch_{p_1 \dots p_k}(0)$, then it may be possible that a

smaller forward code exists. Hence, to fully ensure that $p_1 \dots p_k$ is canonical, we must build every tree starting from a vertex $i \in [1,k]$ such that $1 + \#Ch_{p_1 \dots p_k}(i) = \#Ch_{p_1 \dots p_k}(0)$, build the associated smallest forward code, and check that it is not smaller than $p_1 \dots p_k$.

For an efficient propagation of Property 1, we maintain a 2 dimensional array t such that, $\forall i,j \in [1,k], t[i][j] \in \{-1,0,1\}$ depending on whether $l_i \prec_{lex} l_j$, $l_i = l_j$, or $l_i \succ_{lex} l_j$.

Propagation of Property 2

We exploit Property 2 to detect some cases where the backward code is not canonical: When the domain of a variable y_l is reduced to a singleton, if the domains of x_j and y_j are also reduced to singletons for each $j \in [1,l]$, a failure is raised if there exists $\sigma \in Aut(p_1 \dots p_k)$ such that $x_1 y_1 \dots x_l y_l \succ_{lex} \sigma(x_1 y_1 \dots x_l y_l)$ where $k \in [1,n]$ is the largest value such that the domain of p_i is reduced to a singleton for each $i \in [1,k]$. We use vertex labels to compute a partition of the vertices in orbits (vertices i and j are in a same orbit if $p_i = p_j$ and $l_i = l_j$), and we use this partition to compute automorphisms. However, to compute $Aut(p_1 \dots p_k)$ when $k < n$, we need to discard automorphisms that may not be valid after the addition of $n-k$ new forward edges (as stated in Def. 5). More precisely, if two vertices u and v belong to a same orbit in $p_1 p_2 \dots p_k$, and the largest vertex in their respective subtrees T_u and T_v is smaller than p_{k+1}, then this orbit is considered to compute $Aut(p_1 \dots p_k)$ because it is preserved in $p_1 \dots p_n$, as T_u and T_v cannot be further extended with forward edges.

Canonicity Check

Properties 1 and 2 are necessary conditions for canonicity, but they are not sufficient. For example, let us consider the tree T' displayed in Fig. 3, and let us assume we have added edges $(2,4),(3,5)$, and $(4,5)$ to T'. In this case, the code $000112,24,35,45$ is not canonical, though its forward code is canonical and it satisfies Properties 1 and 2. Indeed, the canonical code is 000112,23,34,46, and it is obtained by starting the BFS from vertex 4 of T'.

Hence, we need to check if there exists another BFS that leads to a smaller code (in which case we raise a failure). More precisely, let $k \in [1,n]$ be the largest value such that the domain of p_i is reduced to a singleton for each $i \in [1,k]$, let $l \in [1,m]$ be the largest value such that the domains of both x_i and y_i are reduced to singletons for each $i \in [1,l]$, and let $G = (V,E)$ be the corresponding graph, *i.e.*, $V = [0,k]$ and $E = \{(p_i,i) | i \in [1,k]\} \cup \{(x_i,y_i) | i \in [1,l]\}$. If there exists a permutation π of $[0,k]$ such that $\text{BFS}_G(\pi) < p_1 \dots p_k x_1 y_1 \dots x_l y_l$, then we raise a failure. We exploit Constraint C8 to limit the set of permutations π to those that start with a vertex that has the same degree as 0 in G. We also exploit Constraints C9 and C10 as well as Properties 1 and 2 to break ties when choosing the next vertex v to visit (Line 6 of Algo 1). Finally, we exploit the property LexBFS introduced in [6] to avoid some BFSs (that cannot lead to canonical codes) by breaking ties when choosing the next neighbor v of u to visit (Line 6). More precisely, for each neighbor v of u, let $N_v = \{num[w] | (v,w) \in E \wedge num[w] < num[u]\}$ be the set of q-numbers of neighbors of v that have already been numbered, and let S_v be the sequence obtained by sorting elements of N_v by increasing value. At each iteration of the loop Lines 6-11, we choose the vertex v which has the smallest sequence S_v, where a sequence S_v is smaller than another sequence $S_{v'}$ if S_v is a prefix of $S_{v'}$ or if $S_v \prec_{lex} S_{v'}$ (see [6] for more details).

Table 1 Results for Problems P_1 to P_3, when considering only Constraints $C1$ to $C10$, or when combining Constraints $C1$ to $C10$ with the global constraint cc. k is the number of vertices, nb is the number of solutions, and t is the CPU time (in seconds) to enumerate these solutions. When time exceeds 200s, $t =' -'$, and nb gives the number of codes enumerated in 200s.

k	P_1				P_2				P_3			
	$C1$-$C10$		$C1$-$C10$+ cc		$C1$-$C10$		$C1$-$C10$ + cc		$C1$-$C10$		$C1$-$C10$ + cc	
	nb	t	nb	t	nb	t	nb	t	nb	t	nb	t
5	15	0.02	2	0.01	3	0.00	1	0.00	0	0.00	0	0.00
7	8,959	0.29	126	0.23	70	0.00	2	0.00	0	0.01	0	0.01
9	9,736,406	177.2	26,631	9.77	3,507	0.11	3	0.01	756,497	20.83	5,804	1.88
11	>9,740,145	-	>318,456	-	286,884	9.17	5	0.07	>8,593,667	-	>835,095	-

First Experiments on Toy Problems

We have implemented our global constraint and our CP model in Choco [17] in a straightforward way[1], using a global *count* constraint to implement Constraints C6, C7, and C10.

To evaluate the interest of our global constraint, we consider three toy problems, denoted P_1, P_2 and P_3. P_1 aims at enumerating all graphs with k vertices and $2k-2$ edges. We denote $G(k, 2k-2)$ this set of graphs. P_2 aims at enumerating all graphs of $G(k, 2k-2)$ which have one vertex of degree $k-1$ and $k-1$ vertices of degree three. P_3 aims at enumerating all graphs of $G(k, 2k-2)$ which have exactly two vertices of degree $\lceil k/2 \rceil$ while all other vertices have degrees strictly lower than $\lceil k/2 \rceil$. Constraints on vertex degrees are defined in a very straightforward way as our CP model already has variables associated with degrees.

In Table 1, we give experimental results obtained on an Intel Xeon E5-2623v3 of 3.0GHz with 32GB of RAM. For the three problems, the number of solutions when considering only Constraints C1 to C10 is much larger than the actual number of different graphs (given by column nb of $C1$-$C10$+cc). For example, when $k = 5$, $G(k, 2k-2)$ only contains 2 different graphs but we generate 15 different codes. The addition of the global constraint cc allows us to compute only canonical codes, and this strongly reduces both the number of generated solutions and the CPU time. When adding constraints on vertex degrees, the number of canonical codes and, therefore, the CPU time are strongly decreased, especially for P_2.

8 Application to the Generation of Connected Claw-Free Cubic Graphs

Cubic graphs are graphs in which each vertex has degree 3. A cubic graph is claw-free if it does not contain $K_{1,3}$ as an induced subgraph. This is equivalent to requiring that every vertex must participate in at least one triangle.

To enumerate all claw-free cubic graphs, we define a CP model composed of Constraint C1 to C10 combined with our global cc constraint. To enforce the cubic degree condition, we set the domain of each degree variable d_i to $\{3\}$, for each $i \in [0, n]$.

To ensure claw-freeness, we introduce a global constraint that ensures that each vertex is involved in at least one triangle. The propagator of this constraint maintains two sets of edges as proposed in [2]: *mandatory edges*, which must be in the solution, and *possible edges*, which may be included. We use sparse sets to efficiently maintain these sets [12].

[1] Our code is available at `https://github.com/godotshaw/bfscanonicalcode.git`

A dynamic variable selection strategy is employed to ensure that $dep_p - dep_x > 1$ is maintained throughout the search, where dep_p and dep_x are the depth levels of the last instantiated forward and backward edge variables, respectively. By prioritizing the instantiation of p-variables while interleaving backward edge variables, we avoid the enumeration of useless spanning trees. Additionally, for every $i \in [1, m]$, y_i is instantiated just after x_i.

As the procedure for checking the canonicity is rather expensive, we introduce a parameter f which allows us to control the frequency of canonicity checking: When $f = 1$, canonicity is checked after each edge assignment; when $f > 1$, it is checked every f edge assignment. Of course, when all variables are assigned, the canonicity check is performed, whatever the value of f is, in order to ensure that the global constraint is satisfied. A similar parameter is used in the SMT-based approach of [11].

Table 2 shows the results for $n \in [20, 44]$ by steps of 2, when our parameter f belongs to $\{1, 10, 20, 30\}$. When $f = 1$, run times are very often longer than when $f \geq 10$. A good tradeoff is reached when $f \in \{10, 20\}$.

In Table 2, we also display the results reported by Codish et al. in [10] (the code of this approach is not available). This approach does not break all symmetries and, therefore, it may compute redundant graphs that are isomorphic to previously enumerated graphs. For example, when $n = 32$, there are 731 different graphs whereas the approach of [10] computes 29,069 solutions. As a consequence, this approach does not scale well and cannot be used to solve larger instances within a reasonable amount of time. As a comparison, our method which only computes non-isomorphic graphs achieves a speed-up of more than 60 for $n \geq 30$ (note however that the two approaches have been run on different computers).

We have adapted the SMT-based approach described in [11] to generate claw-free cubic graphs, and we display the results obtained with this approach, on the same computer as the one used in our experiments. To handle the claw-free constraint, we directly utilize the solver's built-in feature, **"–forbidden-induced-subgraphs"**, to prohibit the predefined induced subgraph $K_{1,3}$. In [11], a parameter similar to f is used to control the frequency of **"minimality check"**, a non-polynomial operation that verifies whether the partially defined graph is canonical in their approach. We report results obtained with $f \in \{1, 10, 20, 30\}$ (the default value for this parameter is 20). The best results of SMT are obtained when $f = 10$, and our method consistently outperforms SMT across all instances.

Finally, we display results obtained with Nauty [13] (using `geng -F` to ensure claw-freeness). Our approach is always more efficient than Nauty, but it needs more memory (*e.g.*, 444584Kb instead of 22760Kb when $n = 30$).

Graph contraction

To show the versatility of our approach, we show how to exploit a property introduced in [7] to speed-up the generation of claw-free cubic graphs.

▶ **Proposition 7** (Claim A in [7]). *The vertex set V of a claw-free cubic graph G can be uniquely partitioned into k sets $V_1, \ldots, V_k$ such that $\forall i \in [1, k]$, the subgraph of G induced by V_i is either a triangle, as displayed in Fig. 5(a) or a diamond as displayed in Fig. 5 (b).*

Hence, a claw-free cubic graph can be contracted by replacing specific patterns with labeled meta-vertices. Let us first define the basic contraction operation.

▶ **Definition 8** (Contraction). *Given a graph $G = (V, E)$, a pattern graph $P = (V_P, E_P)$, and a set $S \subseteq V$ such that the subgraph of G induced by S is isomorphic to P, the contraction of G with respect to P and S is the multigraph $G' = (V', E')$ such that the occurrence of P in G is replaced with a single meta-vertex u_S, i.e., $V' = V \setminus S \cup \{u_S\}$ and $E' = \{(u, v) \in E | \{u, v\} \subseteq V \setminus S\} \cup \{(u_S, v) | \exists u \in S, (u, v) \in E\}$. The meta-vertex u_S is called a P-meta-vertex.*

Table 2 Results for generating connected claw-free cubic graphs. Each line successively gives the number n of vertices, the number of different graphs, the results of [10] (number of solutions and time), and the time of our CP model and of the SMT approach of [11] (when the frequency of canonicity checking is in $\{1/1, 1/10, 1/20, 1/30\}$), as well as Nauty [13]. All times are in seconds and best times are highlighted in gray. We display '-' when time exceeds 3600 seconds.

n	graphs	Approach of [10]		Our CP model $C1$-$C10$+cc				SMT approach of [11]				Nauty
		sols	time	1/1	1/10	1/20	1/30	1/1	1/10	1/20	1/30	
20	15	132	2.1s	0.5	0.4	0.5	0.5	1.6	0.8	1.5	1.3	0.5
22	27	307	3.9s	1.0	0.7	0.8	0.9	3.3	1.9	2.5	3.5	2.3
24	54	660	11.0s	1.2	1.2	1.3	1.6	9.4	3.6	4.2	5.3	10.5
26	94	1,835	45.4s	2.2	1.6	2.0	2.5	19.9	5.3	6.5	7.7	48.6
28	181	4,372	2.57m	3.7	2.5	2.7	4.2	33.5	10.6	15.1	18.5	218.2
30	369	10,567	6.60m	6.4	4.9	4.3	8.0	63.2	20.8	21.5	29.0	3293.7
32	731	29,069	24.28m	11.0	8.9	7.5	14.4	138.4	33.2	46.4	44.7	-
34	1,502	-	-	25.0	18.5	16.0	20.9	313.6	70.2	87.6	82.4	-
36	3,187	-	-	59.7	32.9	44.4	37.3	512.9	135.5	177.7	190.6	-
38	6,914	-	-	144.7	92.8	128.1	77.6	935.6	249.6	328.7	348.9	-
40	15,025	-	-	421.9	232.2	295.7	206.8	1936	548.8	734.0	775.0	-
42	33,687	-	-	1195	456.2	589.9	594.0	4872	1420	1526	1511	-
44	77,450	-	-	3094	1133	1142	1461	-	2979	-	3584	-

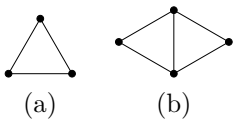


Figure 5 (a): Triangle. (b): Diamond.

E' is a multiset (and therefore G' is a multigraph) because there may exist a vertex $v \in V \setminus S$ such that several vertices of S are adjacent to v.

We consider a contracted graph obtained by contracting triangles and diamonds. In a cubic graph, two diamonds cannot share a same vertex. Also, two triangles that are not in a diamond cannot share a same vertex. To ensure a deterministic process such that the contracted graph is unique, we contract diamonds before triangles.

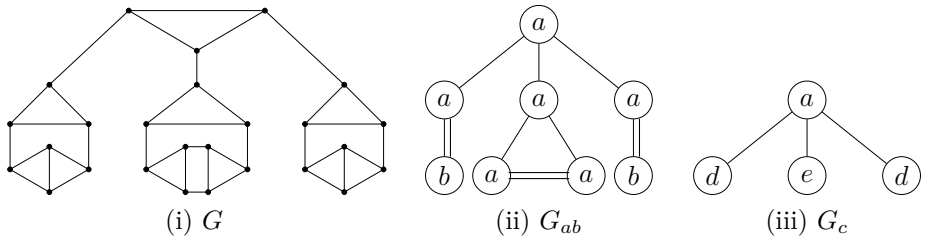
▶ **Definition 9** (Contracted graph G_{ab})**.** *Given a claw-free cubic graph G, the contracted graph G_{ab} is the graph obtained by (i) contracting every occurrence of the diamond pattern (b) in G into a b-meta-vertex, and (ii) contracting every occurrence of the triangle pattern (a) into an a-meta-vertex.*

We display in Fig. 6 an example of contracted graph G_{ab}.

Proposition 7 ensures us that G_{ab} only contains meta-vertices. As G is a cubic graph and the triangle has 3 vertices of degree 2, the degree of every a-meta-vertex is 3. Similarly, the degree of every b-meta-vertex in G_{ab} is 2 because the diamond has 2 vertices of degree 3.

▶ **Proposition 10.** *There are exactly 3 patterns that may create multi-edges in G_{ab}. These patterns, named (c), (d), and (e), are displayed in Fig. 7.*

Proof. A bridge in a connected graph is an edge such that the graph without this edge is no longer connected. There are only two possible claw-free cubic graphs that contain no bridge. These two graphs are displayed in Fig. 8 (first two graphs on the left) and they are treated as special cases. All other claw-free cubic graphs contain at least one bridge. Let G be one of these graphs and G' be the graph obtained from G by removing all bridges. G' is composed of k connected components such that each connected component is a claw-free graph that



(i) G (ii) G_{ab} (iii) G_c

Figure 6 Example of graph contraction. From the claw-free cubic graph G we obtain the multigraph G_{ab} by contracting the two diamond occurrences and then contracting the 6 triangle occurrences. From G_{ab}, we obtain G_C by contracting the occurrence of pattern (e), and the 2 occurrences of pattern (d).

does not contain bridges and that contains at least one vertex of degree 2. The multi-edges in G_{ab} are also multi-edges in G'_{ab}. Each connected component of G' is necessarily one of the three graphs displayed in Fig. 7. ◀

Hence, to remove all multi-edges, we contract G_{ab} as defined below, starting with occurrences of (e) because (c) is a subgraph of (e).

▶ **Definition 11** (Contracted graph G_C)**.** *Given a contracted graph G_{ab}, the contracted graph G_C is obtained from G_{ab} by (i) contracting every occurrence of (e) into an e-meta-vertex, then (ii) contracting every occurrence of (c) or (d) into a c-meta-vertex or a d-meta-vertex.*

We display in Fig. 6 an example of contracted graph G_C. We can show that for any initial claw-free cubic graph G, the contracted graph G_c is either one of the 5 graphs displayed in Fig. 8, which are treated as special cases, or it is a simple connected graph that does not contain multi-edges and that contains 5 different kinds of meta-vertices: the degree of meta-vertices of type a (resp. b, c, d, and e) is 3 (resp. 2, 1, 1, and 2). The meta-vertices of G_C define a partition of the vertices of G: each meta-vertex of type a (resp. b, c, d, and e) corresponds to a set of 3 (resp. 4, 6, 7, and 9) vertices of V.

▶ **Proposition 12.** *Let G and G' be two claw-free cubic graphs, and let G_C and G'_C be their corresponding contracted graphs. Then G_C and G'_C are isomorphic if and only if G and G' are isomorphic.*

Proof. To prove the proposition, we establish that the contraction process from a claw-free cubic graph G to its contracted graph G_C is bijective. First, the contraction process following Definition 9 and Definition 11 is deterministic. Additionally, we label each vertex in G_C according to the subgraph pattern from which it was contracted, ensuring that G_C retains information about the structure of G. Therefore, the contracted graph G_C is uniquely determined by G. Conversely, given G_C, we can reconstruct G uniquely by replacing each meta-vertex in G_C with its corresponding subgraph pattern. Since each subgraph pattern is symmetric with respect to the vertices of degree 2, there is no ambiguity in reconnecting the subgraphs. The connections between these subgraphs follow directly from the adjacency structure of G_C, ensuring a unique reconstruction of G.

This uniqueness in both directions (from G to G_C and from G_C to G) implies that G and G_C are isomorphically equivalent. ◀

The enumeration of claw-free cubic graphs can now be simplified to the enumeration of contracted graphs G_C (except when $n \in \{4, 6, 8, 10, 12\}$ in which case we must add the special cases displayed in Fig. 8). These contracted graphs have labels associated with vertices. Hence, we introduce a new canonical code for labeled graphs.

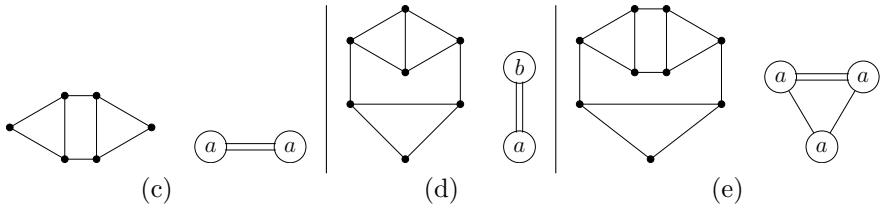


Figure 7 Patterns (c),(d),(e) and their corresponding contracted graph G_{ab}.

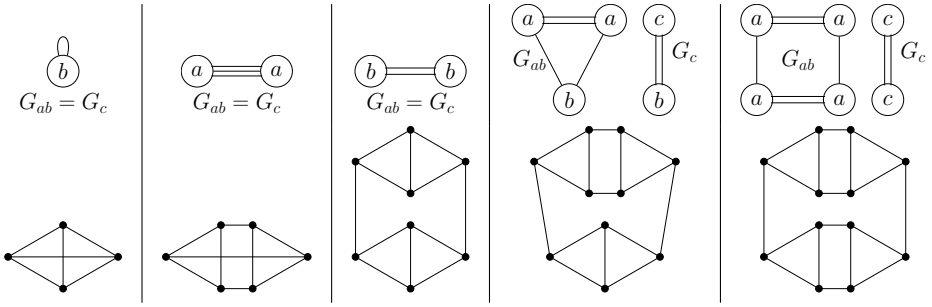


Figure 8 Claw-free cubic graphs for which G_C contains multi-edges or loops: for each graph G on the bottom row, we display its associated contracted graphs G_{ab} and G_C on the top row.

▶ **Definition 13** (Canonical code for a labeled graph). *Let $G = (V, E)$ be a connected graph, L be a finite set of labels, and $l : V \to L$ be a vertex labeling function. A code is a sequence $c \cdot l_0 l_1 \dots l_n$, where $n = \#V - 1$, c is a BFS-based code of G and $\forall i \in [0, n], l_i \in L$ is the label of the vertex whose q-number is i. The canonical code, denoted $ccl(G)$, is the lexicographically smallest among all possible codes generated by different BFS traversals of G.*

Under this definition, the canonical code for the contracted graph in Fig. 6 is represented as $000, adde$. Given a connected claw-free cubic graph G of order n, the size of the corresponding contracted graph G_c is not fixed but instead depends on the types and frequencies of patterns present in G. Hence, we introduce a new integer variable N_x for each $x \in \{a, b, c, d, e\}$ that corresponds to the number of x-meta-vertices. We also introduce two new integer variables n_C and m_C that correspond to the number of variables and edges of the contracted graph G_C. Finally, for each $i \in [0, n_C - 1]$, we introduce a variable l_i which represents the label of the ith meta-vertex of G_C. The following constraints must be satisfied:

- $n_C = N_a + N_b + N_c + N_d + N_e$
- $m_C = \frac{3n}{2} - (3N_a + 5N_b + 8N_c + 10N_d + 13N_e)$
- $n = 3N_a + 4N_b + 6N_c + 7N_d + 9N_e$
- Connectedness: $n_C \leq m_C + 1$
- Occurrence of vertices with the same label:
  $\forall k \in \{a, b, c, d, e\}, N_k = \#\{i \mid l_i = k,\ i \in [0, n_C - 1]\}$
- Degree constraints:
  $\forall i \in [0, n_C - 1],\ d_i = 3 \Leftrightarrow l_i = a \wedge d_i = 2 \Leftrightarrow l_i \in \{b, c\} \wedge d_i = 1 \Leftrightarrow l_i \in \{d, e\}$

In this model, the 5 special graphs shown in Fig 8 are not addressed. However, they are accounted for when counting the number of solutions for $n \in \{4, 6, 8, 10, 12\}$.

We add to these constraints the constraints C1 to C10 of Fig. 2, while replacing n and m with n_C and m_C. We also extend our global canonicity constraint by ensuring that the label sequence is the smallest possible one under all possible BFS traversals.

Table 3 presents the solving time for enumerating all contracted graphs corresponding to all claw-free cubic graphs for $n \in \{20, \ldots, 60\}$ when the frequency parameter f is set to 1 (greater values for f do not improve results). Since the contracted graph is significantly smaller than the original claw-free cubic graph, we observe a notable reduction in solving times. For instance, when $n = 36$, generating 6914 solutions takes only 2.2 seconds, whereas enumerating the primary claw-free cubic graphs requires 32.9 seconds. With this approach, we can generate graphs up to $n = 60$ within 9 hours.

Table 3 Results for generating all claw-free cubic graphs when using contracted graphs.

n	graphs	time	n	graphs	time	n	graphs	time
20	15	0.1s	34	1,502	1.6s	48	418,112	131.2s
22	27	0.2s	36	3,187	2.2s	50	1,005,927	293.9s
24	54	0.3s	38	6,914	3.2s	52	2,412,987	756.0s
26	94	0.3s	40	15,025	5.9s	54	5,934,636	1,877.8s
28	181	0.4s	42	33,687	11.8s	56	14,823,532	4,526.9s
30	369	0.7s	44	77,450	23.6s	58	37,005,614	11,836.1s
32	731	0.8s	46	177,465	54.1s	60	94,412,125	30,948.9s

9 Conclusion

We introduce a new canonical graph encoding based on a BFS traversal, which is well-suited for constraint-based approaches. Using this encoding, we define a CP model that enumerates all non-isomorphic graphs satisfying specific constraints by generating their corresponding canonical codes. The key aspect of this encoding is identifying a spanning tree that yields the lexicographically smallest representation of the graph. We demonstrate how this encoding enables the formulation of fundamental static symmetry-breaking constraints. Additionally, we leverage the crucial property that every prefix of a canonical code must also be canonical, allowing us to define a global constraint that dynamically enforces symmetry-breaking, ensuring the canonicity of the generated codes.

We tested our approach on connected claw-free cubic graphs, which are regular and inherently challenging for BFS-based methods. However, our experimental results demonstrate that our approach outperforms existing state-of-the-art techniques based on adjacency matrix representations. Beyond this, we explored the properties of local structures within this class of graphs that allow us to contract graphs into more compact labeled graphs. Our BFS-based canonical code may be easily extended to labeled graphs, and this allows us to enumerate graphs of larger order, up to 60 vertices. As part of our future work, we aim to refine our encoding for such labeled graphs and develop dedicated propagation strategies.

Our approach is well-suited for enumerating graphs with a limited diameter, such as diameter-2-critical graphs and required-girth extremal graphs [11]: In these cases, the number of canonical spanning trees remains manageable, allowing them to be precomputed and used for static symmetry breaking. We plan to publish further results on these aspects in the near future, as they could not be included in this work due to space limitations.

References

1 Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

2 Nicolas Beldiceanu, Irit Katriel, and Xavier Lorca. Undirected forest constraints. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Re-*

search (OR) Techniques in Constraint Programming, pages 29–43. Springer, 2006. doi:10.1007/11757375_5.

3 Michael Codish, Graeme Gange, Avraham Itzhakov, and Peter J Stuckey. Breaking symmetries in graphs: the nauty way. In *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings 22*, pages 157–172. Springer, 2016. doi:10.1007/978-3-319-44953-1_11.

4 Michael Codish, Alice Miller, Patrick Prosser, and A Stuckey. Breaking symmetries in graph representation, 2013.

5 Laurianne David, Amol Thakkar, Rocío Mercado, and Ola Engkvist. Molecular representations in ai-driven drug discovery: a review and practical guide. *Journal of cheminformatics*, 12(1):56, 2020. doi:10.1186/S13321-020-00460-5.

6 Michel Habib and Christophe Paul. A simple linear time algorithm for cograph recognition. *Discrete Applied Mathematics*, 145(2):183–197, 2005. doi:10.1016/J.DAM.2004.01.011.

7 Michael A Henning and Christian Löwenstein. Locating-total domination in claw-free cubic graphs. *Discrete Mathematics*, 312(21):3107–3116, 2012. doi:10.1016/J.DISC.2012.06.024.

8 Marijn JH Heule. Optimal symmetry breaking for graph problems. *Mathematics in Computer Science*, 13:533–548, 2019. doi:10.1007/S11786-019-00397-5.

9 Avraham Itzhakov and Michael Codish. Breaking symmetries in graph search with canonizing sets. *Constraints*, 21:357–374, 2016. doi:10.1007/S10601-016-9244-Z.

10 Avraham Itzhakov and Michael Codish. Breaking symmetries with high dimensional graph invariants and their combination. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 133–149. Springer, 2023. doi:10.1007/978-3-031-33271-5_10.

11 Markus Kirchweger and Stefan Szeider. Sat modulo symmetries for graph generation and enumeration. *ACM Transactions on Computational Logic*, 2024.

12 Vianney Le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-Sets for Domain Implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, Uppsala, Sweden, September 2013. URL: https://hal.science/hal-01339250.

13 Brendan D McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of symbolic computation*, 60:94–112, 2014. doi:10.1016/J.JSC.2013.09.003.

14 Vyacheslav Moklev and Vladimir Ulyantsev. Bfs enumeration for breaking symmetries in graphs, 2018. arXiv:1804.02273.

15 Siegfried Nijssen and Joost N Kok. The gaston tool for frequent subgraph mining. *Electronic Notes in Theoretical Computer Science*, 127(1):77–87, 2005. doi:10.1016/J.ENTCS.2004.12.039.

16 Xiao Peng and Christine Solnon. Using canonical codes to efficiently solve the benzenoid generation problem with constraint programming. In *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

17 Charles Prud'homme and Jean-Guillaume Fages. Choco-solver: A java library for constraint programming. *Journal of Open Source Software*, 7(78):4708, 2022. doi:10.21105/joss.04708.

18 Vladimir Ulyantsev, Ilya Zakirzyanov, and Anatoly Shalyto. Bfs-based symmetry breaking predicates for DFA identification. In *Language and Automata Theory and Applications - 9th International Conference, LATA 2015*, volume 8977 of *Lecture Notes in Computer Science*, pages 611–622. Springer, 2015. doi:10.1007/978-3-319-15579-1_48.

19 David Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of chemical information and computer sciences*, 28(1):31–36, 1988. doi:10.1021/CI00057A005.

20 Xifeng Yan and Jiawei Han. Closegraph: mining closed frequent graph patterns. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 286–295, 2003. doi:10.1145/956750.956784.

Constraint-Based In-Station Train Dispatching

Andreas Schutt
Data61, CSIRO, Melbourne, Australia

Matteo Cardellini
DIBRIS, University of Genova, Italy

Jip J. Dekker
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Daniel Harabor
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia

Marco Maratea
Department of Mathematics and Informatics, University of Calabria, Rende, Italy

Mauro Vallati
Department of Computer Science, University of Huddersfield, UK

Abstract

In-station dispatching is the problem of planning the movements of scheduled trains inside a railway station. Effective solutions for in-station dispatching are important for maximising the utilisation of railway infrastructure and for mitigating the impact of incidents and delays in the broader network.

In this paper, we explore a constraint-based approach to perform in-station train dispatching. Our extensive empirical analysis of multiple modelling, search strategy, and solver choices, performed over synthetically generated, yet realistic, data, shows that our method outperforms the existing planning-based state-of-the-art approach. In addition, we present different optimisation criteria, which can be effortless defined thanks to the constraint-based approach.

2012 ACM Subject Classification Applied computing → Operations research; Computing methodologies → Planning and scheduling

Keywords and phrases in-station train dispatching, train scheduling, railway scheduling, constraint programming, mixed-integer programming

Digital Object Identifier 10.4230/LIPIcs.CP.2025.33

Supplementary Material *Dataset*: `https://github.com/ShortestPathLab/train_dispatching_benchmark`

1 Introduction

Railways play a significant economical role in our society for transporting either goods or passengers, but the increasing volume of people and freight transported on railways is congesting the networks [4]. Train traffic control in railway networks deals with the problem of finding appropriate routes for trains to respect a given timetable, and is usually divided into two main areas: line dispatching [20] and in-station train dispatching [6].[1] The former considers the overall railway network and the routing of trains between different railway stations. The latter is focused on the routing of trains inside a specific station; to deal with delays and disruptions and minimise their overall negative impacts, such as cost penalties for the rail operator and inconveniences for passengers.

[1] We note that both or a combination of both problems are sometimes referred as train or railway scheduling problem in the operation research and constraint programming community.

© Andreas Schutt, Matteo Cardellini, Jip J. Dekker, Daniel Harabor, Marco Maratea, and Mauro Vallati;
licensed under Creative Commons License CC-BY 4.0

Despite its importance, the in-station train dispatching problem is often handled manually by experienced operators in charge of one or more connected stations [21, 17, 6]. These operators monitor the situation inside a station and instruct train conductors about which path to follow and when. Operators receive limited support from railway control systems (e.g., RailSys, a popular software package originally developed for Deutsche Bahn) which focus mainly on operational performance indicators and operator support services, such as simulation and validation to measure the quality of human-driven decision-making (see, e.g., [12]). These limitations exist, in part, because in-station train dispatching is a hard complex problem falling in the class of job-shop scheduling problems [19]. Typically, there are three key decisions to be made for each train: what time it should enter the station, which path (including the platform) it should take, and how long it should dwell at the platform. A main complication is that each train needs to be routed through the station such that its occupation of track segments does not overlap with that of other trains.

Several notable works appear in the research literature, which attempt to address the in-station dispatching problem automatically. Mannino and Mascis [21] developed a specialised algorithm that offers real-time re-scheduling over a 1.5 hour planning horizon, but limited to just a few (≤ 8) trains. Their dataset is no longer available. Meanwhile, Kumar et al. [17] consider a CP-based optimisation method which tackles up to 23 trains and substantially larger station layouts. Unfortunately, this approach requires several minutes to solve one instance, decisions for most of the trains are already fixed a priori, and solutions are limited to a small time horizon (10 minutes). Their datasets are not available. Recently, Cardellini et al. [6] made a significant step with a method based on automated planning. Their approach models the problem using PDDL+ [13] and uses a modified version of the planning engine ENHSP [25, 26] for solving. This system can tackle more trains (30+) over a larger time horizon (2+ hours) with solutions computed in just seconds. For confidentiality reasons, their datasets are also not available.
In this paper, we make substantial new contributions to in-station train dispatching:

1. We develop a new constraint-based model using the modelling language MiniZinc [28]. MiniZinc has several compelling advantages compared to PDDL+: it is easier to understand, it does not change for every instance, and it offers the opportunity to explore a range of solution approaches, including different objectives, solving technologies and search strategies.

2. We generate and make available a new benchmark dataset, which is synthetic but based on real historical data. Modelled on a typical medium-sized Italian railway station, we generate a range of problem instances with up to 50 trains over a several hour time horizon – substantially larger than problems previously appearing in the literature.

3. We analyse the performance of our approach using a range of solvers and two different objectives: *makespan* and sum of (individual train) *endtimes*. Results show that our method is competitive in terms of runtime to the PDDL+ planning approach in [6]: we solve more problems overall, we can usually find a first solution with a comparable runtime of just seconds, and we solve most problems to optimality given a few minutes.

We report results for a wide range of solving technologies including constraint programming (CP) and mixed-integer programming (MIP) solvers, i.e., Chuffed, Google's OR-Tools solver CP-SAT, IBM ILOG CP Optimizer, and Gurobi. We believe this is the largest evaluation and empirical comparison yet undertaken for in-station train dispatching.

2 Related Work

In the broader topic of railway traffic control, most works deal with the problem of finding appropriate routes for trains, usually regarding a given official timetable [30]. By contrast, we focus on the problem of in-station train dispatching, where the main challenge is optimising the movement of trains inside stations. Given the hard combinatorial nature of the problem, many works have focused on related sub-problems or on a more abstract formulation of the overall problem. Rodriguez [24] formulated a CP model for performing train scheduling at a junction, which shares some characteristics of a station, but does not include platforms and stops. Other works such as [7, 3, 8] focused on the problem of assigning trains to available platforms, given the timetable and a set of operational constraints, without focusing on the paths that trains should follow to reach the platforms. Meanwhile, Caprara et al. [5] analysed a different perspective and focused on the identification and evaluation of recovery strategies to be applied in case of delays of one or more trains arriving at the controlled station. These strategies include actions such as the use of backup platforms or alternative paths, and are assumed to be readily applicable.

A few works have directly addressed the in-station train dispatching problem. We discussed these works in Section 1 but give more details here. Mannino and Mascis [21] introduced a MIP model for controlling a metro station. Their approach, evaluated on data from a medium-sized Milanese metro station, can effectively manage delays by re-scheduling trains in real-time (1 second per solve iteration). Unfortunately, the approach is limited to small numbers of trains. In Kumar et al. [17] authors consider a CP model for performing in-station train dispatching for a busy Indian terminal. Their approach can deal with a large railway station and more trains, but at the cost of considering only very short time horizons (less than 10 minutes). The approach depends on station-specific optimisations, which means the model and associated gains are not easily generalised to other settings. Another drawback is the approach used to model non-overlap constraints, which ensure paths assigned to trains are collision-free. Their proposed model employs sub-paths of train routes and platforms where a sub-path is a path from one signal/semaphore (see Section 3) to another one (not necessarily the next one) and, thus, it is a higher abstract level than a track segment. Using this approach, trains need to wait until an entire sub-path is released before they can use any track segment that is part of the sub-path. This eliminates some plans and could be avoided if (as in our approach) the non-overlapping constraints were modelled over track segments. A final difference is evaluation, where reported results are all produced by a single solver: IBM ILOG CP Optimizer.

The most recent and closer work to ours is from Cardellini et al. [6]. In this paper, as already mentioned in the introduction, the authors presented an approach based on automated planning with PDDL+, a mixed discrete-continuous extension of the standard PDDL language. A modified version of the PDDL+ planning engine ENHSP is then used for solving. The approach was validated on historical data of a medium-sized railway station from the North-West of Italy provided by Rete Ferroviaria Italiana (RFI), but because of confidentiality issues the authors could not share the data.

Other related CP methods are proposed in [14, 22]. Geske [14] proposed a CP approach for simulating the train timetable for parts of the German railway network, which combines line and in-station train dispatching. The CP model uses the global constraints cumulative and diffn for modelling the non-overlapping constraints over track segments, and the method stops once a solution is found. Masoud et al. [22] implemented a CP model in IBM ILOG CP Optimizer for scheduling coal trains between mines and a port in Australia using the

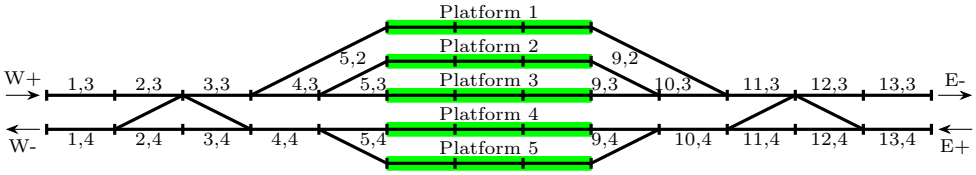


Figure 1 The diagram of the railway station under consideration (adapted from `https://github.com/matteocarde/icaps2021`).

global constraint disjunctive for modelling non-overlapping constraints on tracks between junction points. Their method minimises the makespan while also considering coal demand constraints at the port. A similar problem, and associated CP-model, is developed for freight-rail capacity-evaluation in [15]. We also note that the MiniZinc Challenge [28] has two CP models "train" and "train-scheduling" for line dispatching, but there is no information on their background and origin.

Compared to the CP models in the literature, we model the non-overlapping constraints over individual track segments, which allows a "tight" scheduling of trains leading to fewer delays. We also explore the usage of redundant constraints to strengthen the model, which has not been done before to the best of our knowledge in this context, and investigate the impact of three solution objectives and four solvers on the solving performance.

3 Problem Description and Model

First, we provide the necessary background for the considered in-station dispatching problem. Next, we introduce our basic constraint-based model and additional redundant constraints to strengthen propagation. Then we introduce different solution objectives and finally we discuss search strategies.

3.1 Problem Description

Figure 1 provides an example structure of a railway station. A railway station can be represented as a graph, composed by a set of connected track segments, i.e., the minimal controllable rail units. Their status can be checked via track circuits, that provide information about occupation of the segment and about corresponding timings. Track segments can be divided into two classes, stopping points and interlocking. A stopping point is a track segment in which a train can stop: this is possible if there is a connected platform, i.e., a point in the station used to embark/disembark the train (in green in Figure 1), or at the entrance and the exit points of the railway station (indicated as E and W in the figure). Entry points are segments where the train stops before being allowed to enter into the station (or queues behind other trains); similarly, exit points allow the train to leave the station and enter the outside railway network.

Sequences of connected track segments are organised in itineraries: this is manually done by experts at the specific railway station. While track segments are the minimal controllable units of a station, itineraries describe paths that the trains will follow in order to move within the station. Semaphores are positioned in the station at specific points, and that are the only points at which trains are allowed to stop, usually signalling the beginning and the end of an itinerary. Intuitively, trains are only allowed to stop at the end of an itinerary, where semaphores are placed. For example, all trains in Figure 1 consists of two itineraries. The first one from the station entrance to the end of the platform in their direction and the second one, the first track segment after the end of the platform till the station exit.

A track segment can be occupied by a single train at the time. For safety reasons, a train is required to reserve an itinerary, and this can be done only if the itinerary is currently not being used by another train. While a train is navigating the itinerary, the track segments left by the train are released. This is done to allow trains to early reserve itineraries even if they share a subset of the track segments. A train occupies a track segment for a duration that depends on many variables; e.g., type of the train, length of the train, position of the segment within the itinerary, and weather. Based on historical data, when available, corresponding values can be estimated for planning purposes. A train going through the controlled railway station is running a route in the station, by reserving an itinerary and moving through the corresponding track segments.

Finally, a timetable is the schedule that includes information about when trains arrive at the controlled station, when they arrive at a platform, and the time when they leave a platform. Figure 2 shows a non-overlapping schedule of three trains moving across the station from Figure 1. We show temporal reservations for itineraries leading to and leaving from each platform. In this example, the orange train overtakes the red train via Platform 3. Currently, human train dispatchers rely on tools to visualise the conditions of the controlled station. When making re-scheduling decisions they use data such as occupied track segments (similar to Figure 2), timetable information and extensive communication with train drivers and personnel in the station. Dispatchers operate in real-time and mostly on a reactive basis, according to the arrival time of trains. They are guided by an intuitive understanding about the quality of potential decisions and their knock-on effects.

3.2 Constraint-based Model

The in-station train dispatching problem can be characterised by the tuple (T, E), where $T = \{1, 2, \dots\}$ is the set of *trains* and $E = \{1, 2, \dots\}$ is the set of track *segments* in the station. A train $t \in T$ is specified by its earliest start time $\underline{s}_t^T \in \mathbb{Z}$, and a set of possible *routes* $R_t \subseteq \mathbb{Z}$ through the station.

For the purposes of our model, we distinguish between four train types $y_t^T \in \{\mathsf{std}, \mathsf{ori}, \mathsf{des}, \mathsf{van}\}$. Here std is a *standard* train that enters and exits the station, to and from the external network; ori is an *origin* train that start at a platform from the beginning of the planning horizon and subsequently exits the station; des is a *destination* train that enters the station, terminates at a platform, and occupies the platform for the remaining planning horizon, and; van is a *vanishing* train that enters the station, terminates at a platform, and "vanishes" from the station after completing its stop. A vanishing train models the case when the train leaves the station into the train yard or side-track, which is assumed to always have enough capacity.

A route $r \in R_t$ of a train t is a sequence of itineraries forming the path of the train through the station. Its grounded representation is modelled as a sequence of temporal *reservation blocks* $B_r = \{i, i+1, \dots, j\}$, where $i, j \in \mathbb{Z}$ and $i < j$, and the integers represent the indices of the blocks. A block $b \in B_r$ of route r specifies a track segment $e_b^B \in E$ and a reservation duration $d_b^B \in \mathbb{Z}_0^+$, excluding dwell time (defined below). A train occupies a block b starting from the sum of the end time of its previous block in the sequence B_r and a start offset time $o_b^B \in \mathbb{Z}$. If b is the first block in the sequence, the start offset time is zero.

When moving through a platform, a train has a minimal dwell time $\underline{l}_r^R \in \mathbb{Z}_0^+ = \{0, 1, \dots\}$. We note that even if a train does not take passengers at the station, it might be allowed to stop at the platform, e.g., waiting for a green signal or a passing of another train through a different platform. However, the minimal dwell time is zero. Similarly, we also define a minimal duration for a train to move through the station $d_r^R \in \mathbb{Z}$, excluding the dwell time.

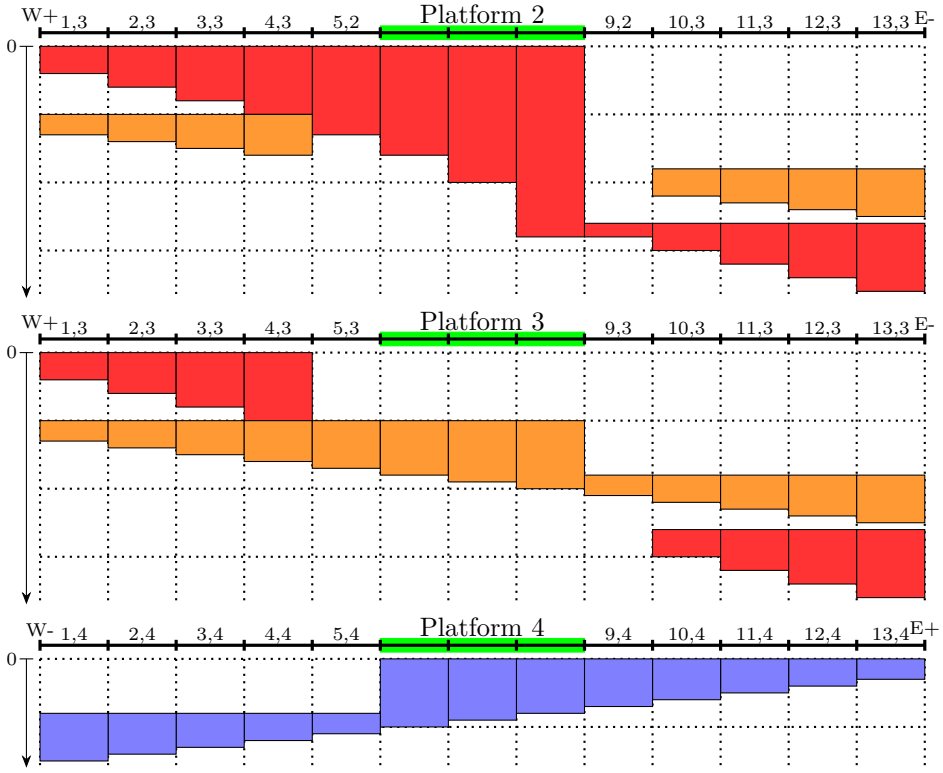


Figure 2 Three Gantt charts of a train schedule of three trains across the station shown in Figure 1, in which the x-axis corresponds to track segments of one route through the station and the y-axis the time. The red, orange, and blue train enters the station at W+, W+, and E+, exits at E-, E-, and W-, and drives via Platform 2, 3, and 4, respectively. The coloured rectangles show the reservation times of the corresponding track segment in the station shown in Figure 1 for each train.

As dwell times are optional, we use a Boolean parameter $p_b^B \in \{\top, \bot\}$ which is true ($\top$) if the block b can contain a non-zero dwell time (i.e., if the train can stop at the block's segment e_b^B) and false ($\bot$) otherwise; the parameter $t_b^B \in T$ refers to the corresponding train. We denote by B the union of all blocks, i.e., $\bigcup_{t \in T} \bigcup_{r \in R_t} B_r$.

Decision Variables. The problem has three variables for each train t, which are the start time variable $\mathbf{s}_t^T$ modelling when trains enter the station or depart from the platform for origin trains, the route variable $\mathbf{r}_t^T$ for selecting the route through the station, and the dwell time variable $\mathbf{w}_t^T$ for modelling the duration a train stops at the platform. In addition, we have optional start time variables $\mathbf{s}_b^B$ and duration variables $\mathbf{d}_b^B$ for each block b, which are used for modelling non-overlapping constraints for each segment in the station. An optional variable is a variable that might be a part of a solution or not. It can be understood as a pair of the integer variables and a Boolean variable that indicates whether the variable is present or absent in a solution. In our case, whether a train traverses a particular segment depends solely on which route has been selected in a solution. Thus, related block variables to that route should be present in the solution while the others absent. All variables are (optional) integer variables.

Constraints. Equations (1–20) show all the constraints of our basic model: (1–5) are the core constraints, (6–11) link the different decision and block variables together, and (12–20) impose bounds on the variables. The constants $\underline{H}$ and $\overline{H}$ represent the start and end time of the planning horizon. The start time is simply $\min_{t \in T} \underline{s}_t^T$, whereas for the end time we compute a trivial upper bound by scheduling the trains on a single track starting with origin trains and ending with destination trains.

$$\mathsf{disjunctive}(Start, Dur) \qquad \forall e \in E : Start = \{\tilde{\mathbf{s}}_b^B \mid b \in B : e_b^B = e\}, \; Dur = \{\tilde{\mathbf{d}}_b^B \mid b \in B : e_b^B = e\} \tag{1}$$

$$\tilde{\mathbf{s}}_b^B = \mathrm{occurs}(\mathbf{s}_b^B) \odot \underline{H} \quad \tilde{\mathbf{d}}_b^B = \mathbf{d}_b^B + (\mathbf{s}_b^B - \underline{H}) \qquad \forall b \in B : p_b^B = \top \wedge y_{t_b^B}^T = \mathsf{ori} \tag{2}$$

$$\tilde{\mathbf{s}}_b^B = \mathbf{s}_b^B \quad \tilde{\mathbf{d}}_b^B = \mathbf{d}_b^B + \overline{H} \qquad \forall b \in B : p_b^B = \top \wedge y_{t_b^B}^T = \mathsf{des} \tag{3}$$

$$\tilde{\mathbf{s}}_b^B = \mathbf{s}_b^B \quad \tilde{\mathbf{d}}_b^B = \mathbf{d}_b^B \qquad \forall b \in B : p_b^B = \bot \vee y_{t_b^B}^T \in \{\mathsf{std}, \mathsf{van}\} \tag{4}$$

$$\mathbf{s}_{t_1}^T \leq \mathbf{s}_{t_2}^T \qquad \forall (t_1, t_2) \in O \tag{5}$$

$$\mathbf{r}_t^T = r \rightarrow \mathbf{s}_t^T = \mathbf{s}_{\min(B_r)}^B \qquad \forall t \in T, \forall r \in R_t \tag{6}$$

$$\mathbf{r}_t^T = r \leftrightarrow \mathrm{occurs}(\mathbf{s}_{min(B_r)}^B) \qquad \forall t \in T, \forall r \in R_t \tag{7}$$

$$\mathbf{d}_b^B = d_b^B \qquad \forall b \in B : p_b^B = \bot \tag{8}$$

$$\mathbf{d}_b^B = d_b^B + \mathbf{w}_{t_b^B}^T \qquad \forall b \in B : p_b^B = \top \tag{9}$$

$$\mathbf{s}_b^B = \mathbf{s}_{b-1}^B \oplus d_{b-1}^B \oplus o_b^B \qquad \forall t \in T, \forall r \in R_t, \forall b \in B_r \setminus \{\min(B_r)\} : \neg(p_{b-1}^B = \top \wedge p_b^B = \bot) \tag{10}$$

$$\mathbf{s}_b^B = \mathbf{s}_{b-1}^B \oplus d_{b-1}^B \oplus o_b^B \oplus \mathbf{w}_t^T \qquad \forall t \in T, \forall r \in R_t, \forall b \in B_r \setminus \{\min(B_r)\} : p_{b-1}^B = \top \wedge p_b^B = \bot \tag{11}$$

$$\mathbf{r}_t^T = r \rightarrow \underline{l}_r^R \leq \mathbf{w}_t^T \qquad \forall t \in T, \forall r \in R_t : p_r^R = \top \tag{12}$$

$$\mathbf{r}_t^T = r \rightarrow \mathbf{w}_t^T = 0 \qquad \forall t \in T, \forall r \in R_t : p_r^R = \bot \tag{13}$$

$$\mathbf{w}_t^T \leq \max_{r \in R_t} \underline{l}_r^R \qquad \forall t \in T : y_t^T = \mathsf{van} \tag{14}$$

$$\mathbf{w}_t^T = 0 \qquad \forall t \in T : y_t^T = \mathsf{ori} \tag{15}$$

$$\underline{s}_t^T \leq \mathbf{s}_t^T \leq \overline{H} \qquad \forall t \in T \tag{16}$$

$$\min(R_t) \leq \mathbf{r}_t^T \leq \max(R_t) \qquad \forall t \in T \tag{17}$$

$$0 \leq \mathbf{w}_t^T \leq \overline{H} - \underline{H} \qquad \forall t \in T \tag{18}$$

$$\underline{H} \leq \mathbf{s}_b^B \leq \overline{H} \qquad \forall b \in B \tag{19}$$

$$0 \leq \mathbf{d}_b^B \leq \overline{H} - \underline{H} \qquad \forall b \in B \tag{20}$$

Constraints (1–4) ensure train reservations are non-overlapping. We model this using the global constraint disjunctive for each segment $e \in E$ in our network (1). The disjunctive constraint guarantees a non-overlapping execution of tasks specified by their start time variables and their duration variables. In our case, we create an optional task by optional start time variables $\tilde{\mathbf{s}}_b^B$ and duration variables $\tilde{\mathbf{d}}_b^B$ for each block b that corresponds to the segment e. Both variables are assigned using their block variables $\mathbf{s}_b^B$ and $\mathbf{d}_b^B$ accounting for the type of train and whether there is a stop at the block. There are three cases to consider, depending on the type of train.

- The first case (Constraint 2) covers an origin train at the stop, that is, at the platform. In this case, the corresponding segment must be blocked from the beginning of the planning horizon $\underline{H}$ until the train leaves it. Note that the return value of the function occurs($\mathbf{s}_b^B$)

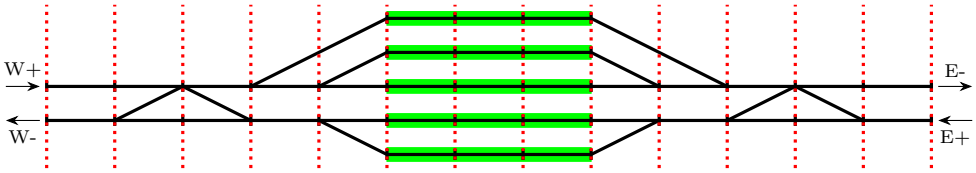


Figure 3 The diagram shows 13 columns of "parallel" track segments separated by red vertical lines of the railway station shown in Figure 1.

is 1 if the variable $\mathbf{s}_b^B$ is present, and absent otherwise. Additionally, we note that the result of the multiplication operator $\odot$ is only present if both multipliers are present. Thus, $\tilde{\mathbf{s}}_b^B$ is only present if $\mathbf{s}_b^B$ is present.

- The second case (Constraint 3) models destination trains having their stop, for which we artificially extend their task's duration by the end of the planning horizon $\overline{H}$, so that the platform is blocked for the remaining planning horizon.
- The last case (Constraint 4) covers all other cases.

Constraint (5) imposes a partial order on trains, in which order they can enter the station from the same entry point (e.g., W+ and E+ in Figure 1), where O is the set of all train pairs (t_1, t_2), for which an order must be imposed. For example, if two trains t_1 and t_2 are entering the station at the same entry point, then t_1 must enter before t_2 if $\underline{s}_{t_1}^T \leq \underline{s}_{t_2}^T$.

Constraints (6–7) ensure that if a route r is chosen (i.e., $\mathbf{r}_r^T = r$) then the start time of the first block of the route $\mathbf{s}_b^B$ equals to the train's start time and $\mathbf{s}_b^B$ is only present in this case. Constraints (8–9) model their duration, which is the reservation time if there is no stop or the reservation time plus the train's dwell time otherwise. Constraints (10–11) connect the start time variables of blocks in the same route where $b-1$ is the immediate predecessor of b and the addition operator $\oplus$ only returns the sum if both summands are presents; otherwise, the result is absent.

Constraint (12) refines the lower bound on the trains' dwell time in the case the route has a stop, while (13) assigns it to zero in the case that there is no stop. Constraint (14) imposes an upper bound on the train's dwell time if the train is vanishing, whereas the dwell time is set to zero for origin trains (15). Constraints (16–20) provide general lower and upper bounds on the decision and block's variables.

3.3 Redundant Constraints

Redundant constraints are additions to a model that do not change the solution space, but which might provide additional propagation for a CP solver. A potential weakness of our model are the non-overlapping constraints (1), for which the start time variables are optional when the train has multiple routes through the station. To introduce redundant constraints we observe that railway stations have a "regular" pattern (e.g., tracks are parallel, "parallel" track segments have similar length and similar position in their itinerary). We model these "parallel" track segments using the global constraint cumulative [1]. For instance, Figure 3 shows a possible separation of track segments by vertical lines. These vertical lines split the station in 13 *columns*, for which one can impose a redundant constraint for each. We note that a segment can be part of multiple columns.

Let C be the set of columns, $E_c^C \subseteq E$ a set of segments in the column c and $T_c^C \subseteq T$ the set of trains that have at least one route through the column. Then, we model the redundant constraints as follows for a column $c \in C$ with $|T_c^C| > |E_c^C|$.

$$\mathsf{alternative}(\mathbf{s}_t^c, \mathbf{d}_t^c, S, D) \qquad \begin{aligned} \forall t \in T_c^C : S = \{\mathbf{s}_b^B \mid b \in B : e_b^B \in E_c^C \wedge t_b^B = t\}, \\ D = \{\mathbf{d}_b^B \mid b \in B : e_b^B \in E_c^C \wedge t_b^B = t\} \end{aligned} \tag{21}$$

$$\min D \leq \mathbf{d}_t^c \qquad \begin{aligned} \forall t \in T_c^C : D = \{d_b^B + (p_b^B = \top \ ? \ \mathbf{w}_t^T : 0) \\ \mid b \in B : e_b^B \in E_c^C \wedge t_b^B = t\} \end{aligned} \tag{22}$$

$$\mathsf{cumulative}(S, D, U, |E_c^C|) \qquad \begin{aligned} S = \{\tilde{\mathbf{s}}_t^c \mid t \in T_c^C\}, \quad D = \{\tilde{\mathbf{d}}_t^c \mid t \in T_c^C\}, \\ U = \{1 \mid t \in T_c^C\} \end{aligned} \tag{23}$$

$$\tilde{\mathbf{s}}_t^c = \mathrm{occurs}(\mathbf{s}_t^c) \odot \underline{H} \qquad \tilde{\mathbf{d}}_t^c = \mathbf{d}_t^c + (\mathbf{s}_t^c - \underline{H}) \qquad \forall t \in T_c^C : (\exists b \in B : t_b^B = t \wedge p_b^B = \top) \wedge y_t^T = \mathsf{ori} \tag{24}$$

$$\tilde{\mathbf{s}}_t^c = \mathbf{s}_t^c \qquad \tilde{\mathbf{d}}_t^c = \mathbf{d}_t^c + \overline{H} \qquad \forall t \in T_c^C : (\exists b \in B : t_b^B = t \wedge p_b^B = \top) \wedge y_t^T = \mathsf{des} \tag{25}$$

$$\tilde{\mathbf{s}}_t^c = \mathbf{s}_t^c \qquad \tilde{\mathbf{d}}_t^c = \mathbf{d}_t^c \qquad \begin{aligned} \forall t \in T_c^C : (\forall b \in B : t_b^B \neq t \vee p_b^B = \bot) \\ \vee \quad y_t^T \in \{\mathsf{std}, \mathsf{van}\} \end{aligned} \tag{26}$$

Constraint (21) creates one mandatory "master" task with the start time variable $\mathbf{s}_t^c$ and duration variable $\mathbf{d}_t^c$ for each train $t \in T_c^C$ passing through the column c by linking both variables to their corresponding blocks' optional start time variables and duration variables from the different routes of the train t. The global constraint alternative is used, which ensures that at most one of the blocks' optional start time variable is present in a solution and, if so, then $\mathbf{s}_t^c = \mathbf{s}_{b'}^B$ and $\mathbf{d}_t^c = \mathbf{d}_{b'}^B$ where b' is the "present" block (i.e., the block belonging to the chosen route); otherwise, $\mathbf{s}_t^c$ is absent. In our case, $\mathbf{s}_t^c$ is present, which forces that exactly one of the blocks' start time variable is present. Constraint (22) imposes a minimum duration on the "master" task duration, but it might be redundant to (21) if the solver natively supports the global constraint alternative. Constraint (23) ensures that at most $|E_c^C|$ trains use one of the segments in E_c^C at any time by setting a task with a start variable $\tilde{\mathbf{s}}_t^c$, duration variable $\tilde{\mathbf{d}}_t^c$, and a resource requirement of 1 for each train t. These variables are then connected to their "master" task's variables $\mathbf{s}_t^c$ and $\mathbf{d}_t^c$ in the constraints (24–26) depending whether the train can have a stop in the column and its train type similar to the constraints (2–4) on page 7.

So far, we characterise a column c via its set of "parallel" segments E_c^C, but we did not specify how we determine this set because it may not contain all "parallel" segments. There are cases in which "parallel" segments belong to different columns because there is no overlap in trains using these "parallel" segments. We determine the set of columns by computing the connected components in an undirected graph for each set of "parallel" segments, where a node represents one "parallel" segment and two nodes are connected if there exists a train having two routes using the corresponding "parallel" segments. Such a partitioning of "parallel" segments leads to stronger redundant constraints.

3.4 Solution Objectives

While the planning method in [6] is tailored to only provide a good first solution and cannot be easily adapted to an optimisation method, for limits of both modelling and solving phases, our model is flexible in this sense, and we can explore different objectives while using the same constraint-based model. We explore three distinct objectives.

Listing 1 The standard search strategy.

```
ann: std = seq_search([
    int_search(start, smallest, indomain_min),
    int_search(route, smallest, indomain_min),
    int_search(dwell, smallest, indomain_min)  ]);
```

Listing 2 The fixed-order search strategy where the array `est` contains the trains' t earliest start times $\underline{s}_t^T$.

```
array [int] of int: order = arg_sort(est);
ann: fixed = int_search(
    [[start[order[i]], route[order[i]], dwell[order[i]]][j]
    | i in index_set(order), j 1..3],
    input_order, indomain_min );
```

Listing 3 The priority search strategy.

```
ann: prio = priority_search( start,
    [ int_search([start[t], route[t], dwell[t]],
                                  input_order, indomain_min)
    | t in T],   smallest, indomain_min );
```

satis first solution subjected to (1–20) and the redundant constraints (21–26) for columns under consideration

makespan $\min \max_{t \in T}(\mathbf{s}_t^T + d^R_{\mathbf{r}_t^T} + \mathbf{w}_t^T)$ subjected to (1–20) and the redundant constraints (21–26) for columns under consideration

endtimes $\min \sum_{t \in T}(\mathbf{s}_t^T + d^R_{\mathbf{r}_t^T} + \mathbf{w}_t^T)$ subjected to (1–20) and the redundant constraints (21–26) for columns under consideration

While the objective **satis** is the same as the planning method, **makespan** optimises the trains at the end of the planning horizon so that all trains exit the station as earliest as possible with less interference to the next planning horizon, and **endtimes** optimises the total amount of end times, which relates to minimising the total amount of delay. We highlighted that we only need to change one line of MiniZinc to change the solution objective. Thus, it would be easy to explore more objectives, e.g., minimising delays and the lexicographical optimisation of minimising the makespan and then the sum of end times.

3.5 Search Strategies

In CP, search strategies can have a significant impact on the solution quality and the solving efficiency. We explore different search strategies only over all the decision variables. For readability, we describe the strategies in the MiniZinc language as search annotations, in which the MiniZinc arrays `start`, `route`, and `dwell` contain the start time ($\mathbf{s}_t^T$), the route ($\mathbf{r}_t^T$), and the dwell time ($\mathbf{w}_t^T$) variables for all trains t, respectively.

standard The standard search (Listing 1) assigns all start time variables before assigning the route and dwell time variables. The search incentivises the earliest start of all trains while keeping the route and dwell time flexible by selecting the train with the earliest possible start time in $\mathbf{s}_t^T$ and assigning this time to the variable.

fixed-order The fixed-order search (Listing 2) groups the three decision variables of a train together and assigns them first in the order of start time, route, and dwell time before fixing the variables of the next train. The trains are sorted in non-descending order of

the trains' earliest start time $\underline{s}_t^T$. Compared to the standard search, it fixes all variables of train, meaning that all corresponding block start time variables become present, and a solver can perform stronger propagation of the non-overlapping constraints earlier in the search. However, the fixed order of the trains does not incentivise the earliest start of all trains as the standard search.

priority The priority search (Listing 3) combines the standard and fixed-order search by grouping the train's decision variables as in the fixed-order search and selecting the train with the earliest possible start time in $\mathbf{s}_t^T$. However, the priority search [11] is not part of MiniZinc distribution and is only supported by Chuffed among the four considered solvers.

free The "free" search is a solver-specific generic search, which differs between solvers. For some solvers, it can be combined with a user-defined search at the solver's discretion (e.g., Chuffed [9, 10] switches between both searches when restarting).

restart For learning solvers as, e.g., Chuffed, a search can be combined with a restart policy, in which the solver restarts the search after a given number of failures are encountered in the search. We explore a geometry restart policy with a factor 1.5 and a base 100, in which the first restart is triggered after 100 failures and the n-th restart after $100 \cdot 1.5^{n-1}$.

4 Experiments

We carried out numerous experiments to test different modelling and solver choices, and compare the planning method in [6] on instances of up to 50 trains. We describe our setup in terms of hardware, instance generation procedure and solver selection and configuration.

Hardware. Our model was implemented in MiniZinc 2.9.2 and all solvers were executed using MiniZinc with default parameters unless otherwise stated. All experiments were conducted on a computational cluster, where each method was given exclusive access to a single core of an Intel Xeon Platinum 8260 CPU at 2.4GHz, 16 GB of RAM. Due to confidentiality, the planning method was run on a single core of a MacBook Pro Mid-2015 machine having a 2.5GHz Intel Core i7 (4870HQ) with 16 GB of RAM. According to the CPU Benchmark at `www.cpubenchmark.net` the MacBook Pro processor achieves a 10% lower mark for single thread computations compared to our cluster's processors. Thus, we conservatively assume the cluster processor is twice as fast as the other one for runtime comparison. We imposed a runtime limit of 300 seconds. The full list of our experimental results are available in Section A.

Instance Data. We took the railway station as depicted in Figure 1, which represents a typical medium-size Italian railway station. For this station, we created 141 realistic instances derived from real-world instances, ranging from one train up to 50 trains. These instances are available at `https://github.com/ShortestPathLab/train_dispatching_benchmark`. For each train, we uniformly chose the train type, the entry/exit points, the platform at which an origin starts, the arrival time of the train, selected from the integer interval $[0, 200 \cdot |T|]$ where a time unit represents one second, and whether or not the train has a stop at the platform taking passengers. The number of origin, destination, and vanishing trains are capped at 5 each. Their sizes are distributed as follows:

1–19 trains: (114 instances) We generate six instances for each number of trains. The maximal optimal makespan for these instances is about 70 minutes.

20–50 trains: (27 instances) We generate three instances for each number of trains with a step size of 5 (i.e., 20, 25, 30, etc.) and a further three instances with 21 and 22 trains. The maximal optimal makespan for these instances is about 170 minutes (almost 3 hours).

In addition, we collect 9 instances of between 1 and 5 trains, available from `https://github.com/matteocarde/icaps2021`. These instances were created by the authors of [6] for the same station (they do not appear in the paper). The maximal optimal makespan for these instances is about 7 minutes.

Solvers. Since our model is written in MiniZinc, we only chose solvers with a MiniZinc interface. We selected Google's OR-Tools solver CP-SAT version 9.12.4544 [23], the best performing CP solvers in recent MiniZinc challenges [28], Chuffed version 0.13.2 [9, 10], a CP solvers known for performing well on scheduling problems (see e.g. [16, 27, 29]), IBM ILOG CP Optimizer version 12.0.1 [18], a commercial CP solver also known for performing very well on scheduling problems, especially when optional tasks are involved, and its usage in the related work [22, 17], and Gurobi version 12.0.1 as the best performing MIP solver in recent MiniZinc challenges. We note that modelling in the solver's interface and having the full control over it may lead to a better performance, but at the cost of implementing equivalent models in different solver interfaces, and maintaining each of them accordingly, which is onerous, and requires a certain expertise level in each of these solvers. Moreover, for MIP solvers, MiniZinc uses advanced linearisation techniques producing models, which are often on-par with a tailored approach [2], and, for CP solvers, MiniZinc's high-level modelling language is close to the native ones in general. Thus, we do not expect a significant performance difference for Chuffed, CP-SAT and Gurobi. For CP Optimizer, this may not be the case, because this solver has a limited MiniZinc interface which does not support e.g., its global constraint alternative.

4.1 Search Strategies and Redundant Constraints

Tables 1 and 2 show the results of different combinations of solvers and search strategies, and different options for using redundant constraints. Table 1 lists the results for the objective **makespan** while Table 2 for **endtimes**. If a search strategy has the postfix "+" then it was combined with the restart policy **restart**. For each combination (a row in the table), we highlight the best performing option for the redundant constraints in *italic*, where an option is better than another one if it finds solutions for more instances (i.e., the sum of optimal and suboptimal solutions) or its average runtime is faster in the case of a tie. If this combination is also the best performing across all search strategies for a solver, then it is shown in **bold**. We test these five options for redundant constraints: (*none*) no redundant constraints, (*border*) only redundant constraints on columns bordering the external railway network, (*platform*) only redundant constraints on columns with track segments at platforms, (*b+p*) options (*border*) and (*platform*), and (*all*) redundant constraints on all columns.

Search Strategies. CP-SAT's free search is faster on average and optimally solved more instances than the standard and fixed-order search, independent of the redundant constraint option and solution objective. We note that CP-SAT pre-terminated with an error in the instance parsing stage for five instances, while it could find a solution for all other 145 instances. For Chuffed, there are more search options and a different performance outcome compared to CP-SAT. Only the fixed-order search with/without restart and priority search with restart can find a solution for every instance in the given time limit, and their performance is similar for the different redundant constraint options and solution objectives.

Table 1 Comparison of the of redundant constraints when minimising the *makespan*. For each method, we present the number of proven optimal instances (opt), the number of instances for which a solution is found (sat), and the average runtime of all instances in seconds (time).

		none			border			platform			b + p			all		
Solver	Search	opt	sat	time	opt	sat	time	opt	sat	time	opt	sat	time	opt	sat	time
Planner			137	28												
Chuffed	standard	110	116	84	*111*	*116*	*82*	109	115	86	110	115	85	110	115	86
	standard+	*110*	*117*	*84*	111	116	82	109	115	86	110	115	85	110	115	86
	fixed-order	136	150	37	**140**	**150**	**30**	135	150	39	138	150	31	138	150	32
	fixed-order+	135	150	38	*139*	*150*	*31*	134	150	39	138	150	31	138	150	32
	priority	126	139	55	*132*	*140*	*46*	126	139	55	131	140	46	130	139	47
	priority+	133	150	44	*138*	*150*	*34*	133	150	46	137	150	35	137	150	36
	free	*137*	*142*	*44*	132	137	51	135	140	47	132	137	54	126	136	62
CP Opt.	free	**139**	**150**	**32**	131	142	47	42	42	218	42	42	218	40	40	228
CP-SAT	standard	*132*	*141*	*44*	129	133	49	129	134	53	128	132	52	128	132	55
	fixed-order	134	145	39	*138*	*145*	*33*	134	145	40	138	145	34	137	145	35
	free	142	145	26	**142**	**145**	**25**	141	145	31	142	145	26	142	145	28
Gurobi	free	136	140	36	**138**	**140**	**32**	132	136	46	135	137	44	129	131	65

To our surprise, priority search without restart did not find a solution for all the instances, which indicates that Chuffed's propagation might not be strong enough to propagate the decisions on one train to other trains. Chuffed's free search optimally solved about 20 instances more than any of its other searches when minimising the sum of end times, but had problems finding a solution for some instances. Moreover, Chuffed has a wide variability on the performance for the different search strategies for minimising the makespan, so that Chuffed's virtual best solver optimally proves 145 instances and only requires an average runtime of 15 seconds while only using redundant constraints on border columns (full results are available in Section A). It shows that no search strategy dominates the other ones in this case. We note that this virtual best solver significantly outperforms any other solver.

Redundant Constraints. It is not the best choice for any solver to have them on all columns or only on the platform columns. This indicates that redundant constraints are not beneficial on all columns, which means that are not bottlenecks as, e.g., platform columns, in dispatching trains. We note that each additional redundant constraint comes with a cost for the solver, which needs to handle the additional variables and constraints during the solving process. For example, the flattening time that MiniZinc needs to convert the model with an instance increased by about 30% and 50% for CP solvers and Gurobi, respectively, which normally correlates to an increase in the problem size. Interestingly, what option is the best depends on solver and solution objectives, which, we suspect, is related to the different propagation strength of the solvers' global constraints and their implemented free search. For CP Optimizer, we see drastic performance deterioration when more redundant constraints are used, which points to that the additional variables created by the redundant constraints negatively impact its search decisions. There is a clear picture across the remaining solvers when minimising the makespan, that having the redundant constraints only for border columns will lead to the best performance of a solver. There is no uniform picture when

Table 2 Comparison of the of redundant constraints when minimising the *sum of end times*. For each method, we present the number of proven optimal instances (opt), the number of instances for which a solution is found (sat), and the average runtime of all instances in seconds (time).

		none			border			platform			b + p			all		
Solver	Search	opt	sat	time	opt	sat	time	opt	sat	time	opt	sat	time	opt	sat	time
Planner			137	28												
Chuffed	standard	97	117	110	*101*	*117*	*106*	98	115	109	101	115	106	99	115	107
	standard+	97	117	110	*101*	*117*	*107*	98	116	109	101	115	106	99	114	107
	fixed-order	103	150	101	*109*	*150*	*89*	104	150	100	*109*	*150*	*89*	108	150	92
	fixed-order+	103	150	101	108	150	90	104	150	101	*109*	*150*	*90*	107	150	92
	priority	102	139	101	108	140	92	103	139	99	*108*	*140*	*90*	107	139	91
	priority+	102	150	101	108	150	90	103	150	100	**110**	**150**	**87**	109	150	89
	free	122	141	72	128	137	61	*125*	*141*	*70*	126	134	66	121	132	71
CP Opt.	free	**106**	**150**	**107**	107	139	108	41	42	220	42	42	220	39	40	228
CP-SAT	standard	*112*	*141*	*82*	116	133	74	111	133	86	114	132	78	114	132	82
	fixed-order	111	145	87	*117*	*145*	*72*	109	145	91	116	145	76	115	145	79
	free	126	145	56	**126**	**145**	**55**	125	145	59	126	145	57	126	145	59
Gurobi	free	**133**	**140**	**45**	136	138	38	129	137	54	132	135	47	130	132	60

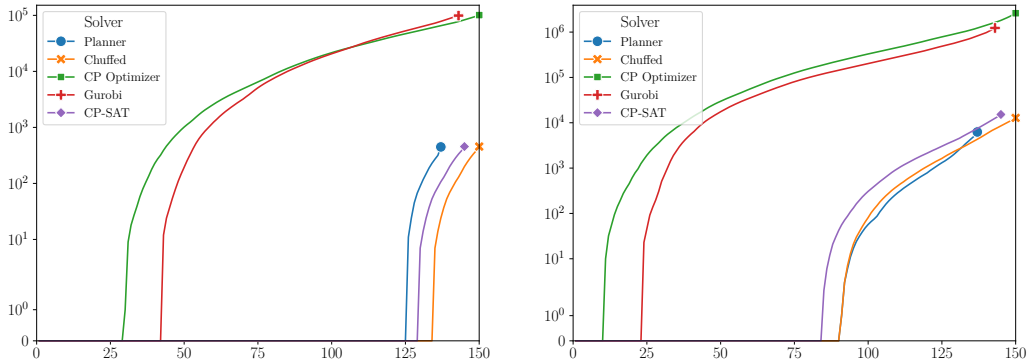
minimising the sum of end times, The best option is (*b+p*) for Chuffed, (*border*) for CP-SAT and Gurobi, and (*none*) for CP Optimizer, which shows that the differences in the solvers' architectures can differently impact their performance for different modelling choices.

Other Observations. The state-of-the-art planning approach in [6] was only able to find solutions for 137 instances whereas Chuffed found solutions for all instances and closed 147 across all cofigurations. CP solutions are competitive to the planning approach in terms of the makespan, sum of end times and computational runtime (see Appendix A for full details). As expected, the problem of minimising the sum of end times is more difficult to solve, which is reflected by the lower number of proven optimal solutions and higher average runtime for any solver because minimising the makespan only minimises the end time of the last train in the schedule.

4.2 Comparison to the State of the Art

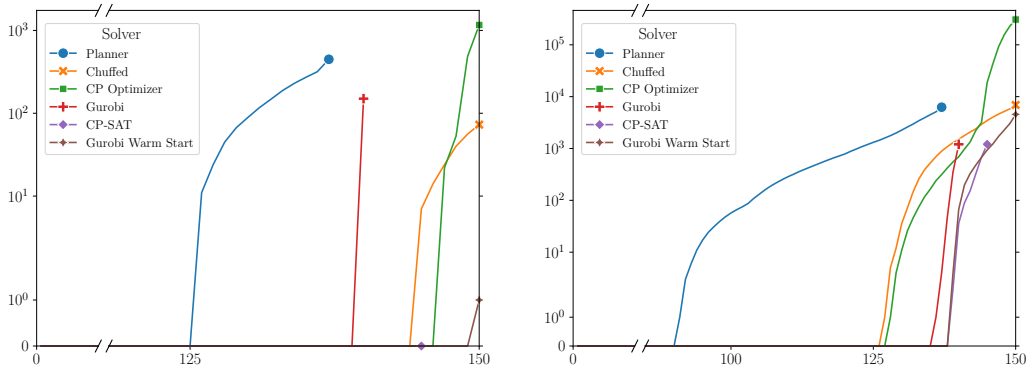
The planner [6] applies a dedicated search heuristic method for finding a first solution with a minimal flow-on impact beyond the station. First, we compare the planner to the best solver combination on the quality of the first solution found regarding their makespan and their sum of end times, and then to the best solution found within the runtime limit.

For the first comparison, the solvers used the solution objective **satis**, and these solver combinations: Chuffed and CP-SAT with fixed-order search and no redundant constraints, and CP Optimizer and Gurobi with free search and no redundant constraints. Figure 4 shows the cumulative differences between the best known objective value for the makespan (Figure 4a) and the sum of end times (Figure 4b) and the solvers' first solution. While the planner does not find a solution for 13 instances, when it found a solution it was always so good as the solutions of other solvers and better than the other solutions for some instances in terms of the makespan and the sum of end times. Unsurprisingly, Chuffed and CP-SAT using the fixed-order search, which incentivised scheduling trains as earliest as possible,



(a) Comparing the makespan. **(b)** Comparing the sum of end times.

Figure 4 The cumulative difference (y-axis) between the first solution found by each method and the best known solution for each solution objective. The x-axis shows the number of instances.



(a) Minimising the makespan. **(b)** Minimising the sum of end times.

Figure 5 The cumulative difference (y-axis) after 5 minutes of optimisation by each method and the best known solution for each solution objective. The x-axis shows the number of instances.

are competitive to the planner. While CP Optimizer and Gurobi first solution quality are understandingly much worse. Overall, Chuffed performs the best in comparison to the other solvers and planner, because it quickly finds a solution for each instance while incurring sometimes a small cost in the solution quality compared to the planner. We note that when only considering the instances for that CP-SAT does not terminate in an error, CP-SAT is the best performing solver.

The considered planner, as all systems solving PDDL+ planning instances, works in satisfying mode, i.e. it terminates after finding the first solution. It cannot be easily modified to operate in the so-called anytime mode, where better solutions are found over time. In our approach, we only have to change the solution objective to one of the two minimisation objectives to find better solutions over time. We used the best solver combinations identified in the previous section, which are highlighted in **bold** in Tables 1 and 2. In addition to these combinations, we also use Gurobi's option to warm start their search by passing in the first solution found by Chuffed using the search strategy *fixed-order* and no redundant constraints to overcome Gurobi's difficulty to finding a solution for all instances. Figure 5 shows plots of the cumulative differences between the best known objective value for the makespan (Figure 5a) and the sum of end times (Figure 5b) for the solvers' best found

solution within five minutes. It is clear that providing more solving time, all solvers find much better solutions than the planner on the harder instances. This highlights that a significant reduction of train delays can be made when the search continues after a first solution is found. Warm starting Gurobi with Chuffed's first solution has not only solved the issue with finding a solution for all instances, but also significantly increased its performance so that it is the best performing solver for both objectives. For minimising the makespan (the sum of end times) it optimally solved 146 (137) instances and found a feasible solution for the remaining 4 (13) instances while having an average runtime of 18 (36) seconds.

Discussion. The solutions obtained show that the solution found by the planning approach are sub-optimal for harder instances indicating the potential in the reduction of the flow-on impacts of delays beyond a station. The best solver for optimising both objective was a combination of Chuffed and Gurobi: Chuffed finds the first solution, which is then injected to Gurobi as a warm start. In terms of redundant constraints, they are an overhead when the solver is only asked to find a solution. When optimisation is required, then imposing redundant constraints on the track segments bordering to the railway network beyond the station is the most beneficial in terms of proving optimality and runtime.

5 Conclusion

We present a constraint-based model for solving the in-station train dispatching problem and explore different options for redundant constraints to strengthen the model, which has not been explored in this context before. Written in the solver-independent modelling language MiniZinc, our model allows us to evaluate multiple leading CP solvers including Google's OR-Tools, CP-SAT, Chuffed, IBM ILOG CP Optimizer as well as the MIP solver Gurobi. In comparison to our approach previous works suffered one or more short-comings: (i) inflexible or train specific models, which are difficult to modify constraints or the solution objective, (ii) not a fine granularity for modelling the non-overlapping constraints for trains (for reducing the problem complexity), and (iii) limited experimental studies on only one solver.

We evaluate our model on synthetic instances generated from real-world data of a railway station in Italy, containing up to 50 trains over a planning horizon of up to almost three hours on the larger instances. Experimental results show that our CP-based approach (Chuffed) can produce first solutions in seconds (similar to SOTA), which is suitable for close-to-real-time decision-making. If more time is available then warm-starting the search (Gurobi) with the first solution (from Chuffed) improves solution cost, up to optimal in just a few minutes for most instances. To our best knowledge, we conduct the largest comparison of solvers for this problem, and optimally solved the largest number of trains yet reported in the literature. Our benchmark dataset is available at `https://github.com/ShortestPathLab/train_dispatching_benchmark` to further work in the area.

References

1 Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and computer modelling*, 17(7):57–73, 1993.

2 Gleb Belov, Peter J. Stuckey, Guido Tack, and Mark Wallace. Improved linearization of constraint programming models. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 49–65, Cham, 2016. Springer International Publishing. `doi:10.1007/978-3-319-44953-1_4`.

3 Alain Billionnet. Using integer programming to solve the train-platforming problem. *Transportation Science*, 37:213–222, 2003. doi:10.1287/TRSC.37.2.213.15250.

4 Joseph Bryan, Glen Elliot Weisbrod, and Carl Douglas Martland. *Rail freight solutions to roadway congestion: final report and guidebook*, volume 586. Transportation Research Board, 2007.

5 Alberto Caprara, Laura Galli, Leo Kroon, Gábor Maróti, and Paolo Toth. Robust train routing and online re-scheduling. In *Proceedings of ATMOS workshop*, pages 24–33, 2010. doi:10.4230/OASICS.ATMOS.2010.24.

6 Matteo Cardellini, Marco Maratea, Mauro Vallati, Gianluca Boleto, and Luca Oneto. In-station train dispatching: A PDDL+ planning approach. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 450–458, 2021. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/15991.

7 Dorotea De Luca Cardillo and Nicola Mione. k l-list λ colouring of graphs. *European Journal of Operational Research*, 106:160–164, 1998. doi:10.1016/S0377-2217(98)00299-9.

8 Partha Chakroborty and Durgesh Vikram. Optimum assignment of trains to platforms under partial schedule compliance. *Transportation Research Part B: Methodological*, 42:169–184, 2008.

9 Geoffrey Chu. *Improving Combinatorial Optimization*. PhD thesis, Department of Computing and Information Systems, University of Melbourne, 2011.

10 Geoffrey Chu, Peter J Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, a lazy clause generation solver, 2018. URL: https://github.com/chuffed/chuffed.

11 Thibaut Feydy, Adrian Goldwaser, Andreas Schutt, Peter J Stuckey, and Kenneth D Young. Priority search with MiniZinc. In *ModRef 2017: The Sixteenth International Workshop on Constraint Modelling and Reformulation*, 2017.

12 Ian Fox et al. The network modelling, timetabling and fuel saving computer programs on the market. Technical report, Australian Rail Track Corporation, 2017.

13 Maria Fox and Derek Long. Modelling mixed discrete-continuous domains for planning. *J. Artif. Intell. Res.*, 27:235–297, 2006. doi:10.1613/JAIR.2044.

14 Ulrich Geske. Railway scheduling with declarative constraint programming. In Masanobu Umeda, Armin Wolf, Oskar Bartenstein, Ulrich Geske, Dietmar Seipel, and Osamu Takata, editors, *Declarative Programming for Knowledge Management*, pages 117–134, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

15 Daniel Harabor and Peter J Stuckey. Rail capacity modelling with constraint programming. In *Integration of AI and OR Techniques in Constraint Programming: 13th International Conference, CPAIOR 2016, Banff, AB, Canada, May 29-June 1, 2016, Proceedings 13*, pages 170–186. Springer, 2016. doi:10.1007/978-3-319-33954-2_13.

16 Stefan Kreter, Andreas Schutt, and Peter J Stuckey. Using constraint programming for solving RCPSP/max-cal. *Constraints*, 22:432–462, 2017. doi:10.1007/S10601-016-9266-6.

17 Rajnish Kumar, Goutam Sen, Samarjit Kar, and Manoj Kumar Tiwari. Station dispatching problem for a large terminal: A constraint programming approach. *Interfaces*, 48(6):510–528, 2018. doi:10.1287/inte.2018.0950.

18 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG. *Constraints*, 23:210–250, 2018. doi:10.1007/S10601-018-9281-X.

19 Leonardo Lamorgese and Carlo Mannino. An exact decomposition approach for the real-time train dispatching problem. *Operations Research*, 63(1):48–64, 2015. doi:10.1287/OPRE.2014.1327.

20 Leonardo Lamorgese, Carlo Mannino, and Mauro Piacentini. Optimal train dispatching by benders'-like reformulation. *Transportation Science*, 50:910–925, 2016. doi:10.1287/TRSC.2015.0605.

21 Carlo Mannino and Alessandro Mascis. Optimal real-time traffic control in metro stations. *Operations Research*, 57:1026–1039, 2009. doi:10.1287/OPRE.1080.0642.

22 Mahmoud Masoud, Erhan Kozan, Geoff Kent, and Shi Qiang Liu. A new constraint programming approach for optimising a coal rail system. *Optimization Letters*, 11:725–738, 2017. doi:10.1007/s11590-016-1041-5.

23 Laurent Perron. Operations research and constraint programming at Google. In *International conference on principles and practice of constraint programming*, pages 2–2. Springer, 2011.

24 Joaquín Rodriguez. A constraint programming model for real-time train scheduling at junctions. *Transportation Research Part B: Methodological*, 41:231–245, 2007.

25 Enrico Scala, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramírez. Interval-based relaxation for general numeric planning. In *Proceedings of ECAI*, pages 655–663, 2016. doi:10.3233/978-1-61499-672-9-655.

26 Enrico Scala, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramírez. Subgoaling techniques for satisficing and optimal numeric planning. *J. Artif. Intell. Res.*, 68:691–752, 2020. doi:10.1613/JAIR.1.11875.

27 Andreas Schutt, Thibaut Feydy, Peter J Stuckey, and Mark G Wallace. Solving RCPSP/max by lazy clause generation. *Journal of scheduling*, 16:273–289, 2013. doi:10.1007/S10951-012-0285-X.

28 Peter J Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The MiniZinc challenge 2008–2013. *AI Magazine*, 35(2):55–60, 2014. doi:10.1609/AIMAG.V35I2.2539.

29 Ria Szeredi and Andreas Schutt. Modelling and solving multi-mode resource-constrained project scheduling. In *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings 22*, pages 483–492. Springer, 2016. doi:10.1007/978-3-319-44953-1_31.

30 Ruifan Tang, Lorenzo De Donato, Nikola Bešinović, Francesco Flammini, Rob M.P. Goverde, Zhiyuan Lin, Ronghui Liu, Tianli Tang, Valeria Vittorini, and Ziyulong Wang. A literature review of artificial intelligence applications in railway systems. *Transportation Research Part C: Emerging Technologies*, 140:103679, 2022.

A Full Results

The following three tables show the full list of our experimental results. Table 3 lists all results for the objective **satis**, i.e., finding a solution. Table 4 lists the results for the objective **makespan**, whereas Table 5 the results for **endtimes**.

Table 3 Comparison of all solving methods when instructed to find the *first solution*. For each method, the number of instances for which a solution is found (sat), the average runtime of all instances in seconds (time), the average makespan, and the average sum of end times.

solver	search	column	sat	time	makespan	end sum
Planner		none	137	28	2219	26844
Chuffed	fixed-order	none	150	5	2699	37449
		border	150	6	2699	37449
		platform	150	6	2699	37449
		b + p	150	6	2699	37449
		all	150	7	2699	37449
		virt. best	150	5	2699	37449
	priority	none	139	5	2291	24957
		border	141	9	2405	28829
		platform	139	6	2291	24957
		b + p	140	7	2349	26797
		all	139	6	2294	25023
		virt. best	142	8	2416	28987
	priority+	none	150	7	2700	37432
		border	150	7	2700	37435
		platform	150	8	2700	37434
		b + p	150	7	2700	37433
		all	150	7	2700	37433
		virt. best	150	7	2700	37430
	free	none	143	8	2836	40346
		border	137	11	2410	27614
		platform	142	8	2763	37174
		b + p	136	13	2307	24669
		all	136	12	2315	24958
		virt. best	143	15	2612	33174
	virt. best	none	150	8	2699	37418
		border	150	7	2699	37421
		platform	150	9	2699	37419
		b + p	150	7	2699	37420
		all	150	7	2699	37420
		virt. best	150	7	2699	37418
CP Opt.	free	none	150	5	3370	54742
		border	140	13	2743	32029
		platform	42	6	637	1937
		b + p	42	6	640	1980
		all	40	30	648	2000

CP-SAT	fixed-order	none	145	7	2770	38680
		border	145	8	2770	38680
		platform	145	8	2770	38680
		b + p	145	9	2770	38680
		all	145	10	2770	38680
		virt. best	145	7	2770	38680
	free	none	145	7	2847	44652
		border	145	8	2898	46322
		platform	145	8	2899	46328
		b + p	145	9	2898	46322
		all	145	12	2990	42546
		virt. best	145	10	2792	39730
	virt. best	none	145	7	2770	38679
		border	145	8	2769	38679
		platform	145	8	2769	38679
		b + p	145	9	2769	38679
		all	145	10	2770	38678
		virt. best	145	8	2770	38676
Gurobi	free	none	143	13	3064	35304
		border	142	12	2970	33647
		platform	135	7	2702	25345
		b + p	137	11	2792	26519
		all	132	21	2566	23028

Table 4 Comparison of all solving methods when optimsing the *makespan*. For each method, the number of proven optimal instances (opt), the number of instances for which a solution is found (sat), the average runtime of all instances in seconds (time), and the average makespan for the solved instances.

solver	search	column	opt	sat	time	makespan
Planner		none	0	137	28	2219
Chuffed	standard	none	110	116	84	1884
		border	111	116	82	1884
		platform	109	115	86	1874
		b + p	110	115	85	1874
		all	110	115	86	1874
		virt. best	111	116	81	1884
	standard+	none	110	117	84	1903
		border	111	116	82	1884
		platform	109	115	86	1874
		b + p	110	115	85	1874
		all	110	115	86	1874
		virt. best	111	117	81	1903
	fixed-order	none	136	150	37	2696
		border	140	150	30	2697
		platform	135	150	39	2696

		b + p	138	150	31	2697
		all	138	150	32	2697
		virt. best	142	150	27	2696
	fixed-order+	none	135	150	38	2696
		border	139	150	31	2697
		platform	134	150	39	2696
		b + p	138	150	31	2697
		all	138	150	32	2697
		virt. best	141	150	28	2696
	priority	none	126	139	55	2288
		border	132	140	46	2345
		platform	126	139	55	2288
		b + p	131	140	46	2345
		all	130	139	47	2290
		virt. best	132	141	45	2358
	priority+	none	133	150	44	2697
		border	138	150	34	2697
		platform	133	150	46	2697
		b + p	137	150	35	2697
		all	137	150	36	2697
		virt. best	138	150	34	2697
	free	none	137	142	44	2384
		border	132	137	51	2250
		platform	135	140	47	2325
		b + p	132	137	54	2247
		all	126	136	62	2256
		virt. best	139	142	39	2381
	virt. best	none	145	150	19	2696
		border	147	150	15	2696
		platform	145	150	20	2696
		b + p	147	150	17	2696
		all	146	150	18	2696
		virt. best	148	150	12	2696
CP Opt.	free	none	139	150	32	2704
		border	131	142	47	2342
		platform	42	42	218	572
		b + p	42	42	218	572
		all	40	40	228	591
CP-SAT	standard	none	132	141	44	2567
		border	129	133	49	2279
		platform	129	134	53	2307
		b + p	128	132	52	2264
		all	128	132	55	2264
		virt. best	132	141	44	2567
	fixed-order	none	134	145	39	2767
		border	138	145	33	2767
		platform	134	145	40	2767
		b + p	138	145	34	2767

		all	137	145	35	2767
		virt. best	138	145	32	2767
	free	none	142	145	26	2767
		border	142	145	25	2782
		platform	141	145	31	2780
		b + p	142	145	26	2784
		all	142	145	28	2781
		virt. best	143	145	22	2767
	virt. best	none	142	145	25	2767
		border	143	145	22	2767
		platform	142	145	28	2767
		b + p	143	145	23	2767
		all	143	145	24	2767
		virt. best	143	145	21	2767
Gurobi	free	none	136	140	36	2259
		border	138	140	32	2253
		platform	132	136	46	2132
		b + p	135	137	44	2163
		all	129	131	65	2012
	warm start	none	144	150	23	2696
		border	146	150	18	2696

Table 5 Comparison of all solving methods when optimising the *sum of end times*. For each method, the number of proven optimal instances (opt), the number of instances for which a solution is found (sat), the average runtime of all instances in seconds (time), and the average sum of end times for the solved instances.

solver	search	column	opt	sat	time	end sum
Planner		none	0	137	28	26844
Chuffed	standard	none	97	117	110	16906
		border	101	117	106	16906
		platform	98	115	109	16419
		b + p	101	115	106	16419
		all	99	115	107	16419
		virt. best	103	117	103	16906
	standard+	none	97	117	110	16906
		border	101	117	107	16906
		platform	98	116	109	16726
		b + p	101	115	106	16419
		all	99	114	107	16404
		virt. best	103	117	104	16906
	fixed-order	none	103	150	101	37421
		border	109	150	89	37417
		platform	104	150	100	37421
		b + p	109	150	89	37417
		all	108	150	92	37417

		virt. best	113	150	82	37416
	fixed-order+	none	103	150	101	37421
		border	108	150	90	37417
		platform	104	150	101	37421
		b + p	109	150	90	37417
		all	107	150	92	37417
		virt. best	113	150	83	37416
	priority	none	102	139	101	24936
		border	108	140	92	26759
		platform	103	139	99	24936
		b + p	108	140	90	26759
		all	107	139	91	24985
		virt. best	110	141	86	26934
	priority+	none	102	150	101	37402
		border	108	150	90	37393
		platform	103	150	100	37402
		b + p	110	150	87	37394
		all	109	150	89	37396
		virt. best	110	150	86	37391
	free	none	122	141	72	26609
		border	128	137	61	24411
		platform	125	141	70	26403
		b + p	126	134	66	21366
		all	121	132	71	20913
		virt. best	130	141	56	25999
	virt. best	none	125	150	65	37386
		border	128	150	58	37381
		platform	127	150	64	37386
		b + p	127	150	58	37382
		all	124	150	62	37383
		virt. best	130	150	53	37380
CP Opt.	free	none	106	150	107	39396
		border	107	139	108	24375
		platform	41	42	220	1582
		b + p	42	42	220	1582
		all	39	40	228	1707
CP-SAT	standard	none	112	141	82	31811
		border	116	133	74	23440
		platform	111	133	86	23439
		b + p	114	132	78	23192
		all	114	132	82	23192
		virt. best	116	141	73	31811
	fixed-order	none	111	145	87	38652
		border	117	145	72	38645
		platform	109	145	91	38652
		b + p	116	145	76	38645
		all	115	145	79	38646
		virt. best	117	145	72	38645

	free	none	126	145	56	38584
		border	126	145	55	38586
		platform	125	145	59	38794
		b + p	126	145	57	38678
		all	126	145	59	38590
		virt. best	127	145	53	38579
	virt. best	none	126	145	56	38584
		border	126	145	55	38586
		platform	125	145	59	38591
		b + p	126	145	57	38590
		all	126	145	59	38590
		virt. best	127	145	53	38579
Gurobi	free	none	133	140	45	23632
		border	136	138	38	21880
		platform	129	137	54	21109
		b + p	132	135	47	20053
		all	130	132	60	18786
	warm start	none	134	150	43	37379
		border	137	150	36	37377

DynamicSAT: Dynamic Configuration Tuning for SAT Solving

Zhengyuan Shi
The Chinese University of Hong Kong, Hong Kong
National Center of Technology Innovation for EDA, Nanjing, China

Wentao Jiang
Ningbo University, China
National Center of Technology Innovation for EDA, Nanjing, China

Xindi Zhang[1]
Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Science, Beijing, China

Jin Luo
Peking University, Beijing, China
National Center of Technology Innovation for EDA, Nanjing, China

Yun Liang
Peking University, Beijing, China

Zhufei Chu
Ningbo University, China

Qiang Xu[1]
The Chinese University of Hong Kong, Hong Kong
National Center of Technology Innovation for EDA, Nanjing, China

Abstract

Boolean Satisfiability (SAT) problem serves as a foundation for solving numerous real-world challenges. As problem complexity increases, so does the demand for sophisticated SAT solvers, which incorporate a variety of heuristics tailored to optimize performance for specific problem instances. However, a major limitation persists: a configuration that performs well on one instance may lead to inefficiencies on others. While previous approaches to automatic algorithm configuration set parameters prior to runtime, they fail to adapt to the dynamic evolution of problem characteristics during the solving process. We introduce DynamicSAT, a novel SAT solver framework that dynamically tunes configuration parameters during solving process. By adjusting parameters on-the-fly, DynamicSAT adapts to changes arising from clause learning, elimination, and other transformations, thus improving efficiency and robustness across diverse SAT instances. We demonstrate that DynamicSAT achieves significant performance gains over the state-of-the-art solver on 2024 SAT Competition Benchmark.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Mathematics of computing → Combinatorial optimization

Keywords and phrases Boolean satisfiability problem, configuration tuning, multi-armed bandit

Digital Object Identifier 10.4230/LIPIcs.CP.2025.34

Supplementary Material *Software (Source Code)*: `https://github.com/cure-lab/DynamicSAT` archived at `swh:1:dir:3c22d81c15345c48063f09c569fed3c286889034`

Funding This work was supported in part by the Hong Kong Research Grants Council (RGC) under Grant No. 14212422, 14202824, and C6003-24Y, in part by National Technology Innovation Center for EDA, China and in part by the NSFC under Grant No. 62274100, No. T2293700 and No. T2293701.

[1] The corresponding authors are: Xindi Zhang and Qiang Xu

© Zhengyuan Shi, Wentao Jiang, Xindi Zhang, Jin Luo, Yun Liang, Zhufei Chu, and Qiang Xu;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

The Boolean Satisfiability (SAT) problem determines whether a given propositional logic formula can be satisfied, recognizing as the first NP-hard problem [8]. Its importance extends beyond theoretical computer science, as many real-world problems, including hardware design [29], planning [22], and scheduling [14], can be efficiently transformed into SAT instances in polynomial time. These problems are then tackled using SAT solvers, which provide a practical framework for addressing challenges across various domains. As the complexity of practical applications continues to grow, so does the demand for more sophisticated and efficient SAT-solving techniques.

The mainstream SAT solvers opt for Conflict-Driven Clause-Learning (CDCL) [28] as the core searching algorithm. In CDCL framework, the problem instances is represented as Conjunctive Normal Form (CNF), consisting of a series of clauses. To enhance the efficiency of CDCL, substantial progress has been made in the SAT community, including advanced SAT solvers [31, 38, 5, 10] and their corresponding techniques [12, 3, 25, 36, 26]. For example, branching techniques [31, 25, 37] guide the solving process by selecting variables and branching orders. Besides, restart technique [12] has been introduced to periodically reset the solving process, helping solvers escape local optima and overcome stagnation in the search space. Clause management strategies [3] focus on simplifying the underlying problem by identifying and removing redundant or non-essential clauses, thus improving computational efficiency and guiding the solver toward viable solutions. These techniques form the backbone of modern SAT solvers, enabling them to handle increasingly complex and diverse applications.

Unfortunately, a well-known challenge remains: techniques that perform effectively on one class of instances may perform poorly on another [7, 24]. To address this variability, modern SAT solvers offer a range of configurable parameters, allowing users to customize heuristics and optimize solver performance for specific problem types. Therefore, algorithm configuration tuning becomes a promising direction in the SAT community, which automates the selection of suitable heuristic parameters, eliminating the need for manual updates. Representative techniques [20, 15, 9] determining the suitable configuration parameters prior to the solving process, have yielded advance across a wide range of problem solving.

Within the CDCL framework, the problem instance undergoes continuous transformation throughout the solving process. The initial CNF instance becomes increasingly dissimilar to its original state due to critical clause operations, such as clause elimination [13], which removes redundant clauses, and clause learning [34], which introduces new clauses into the instance. As illustrated in Figure 1, the cosine similarity between the intermediate instance and the initial instance decreases significantly over time. Consequently, this paper raises a critical question: *Does this divergence indicate that the initial configuration may no longer remain optimal during runtime, resulting in degraded solver performance?*

To address this question, we propose **DynamicSAT**, a novel solution that dynamically tunes the solver configuration during the solving process instead of maintaining the initial configuration throughout. Specifically, we model the configuration tuning process as a decision-making problem and incorporate a general reward function tied to solving efficiency. As a plug-in for modern SAT solvers, DynamicSAT adjusts arbitrary configuration parameters on-the-fly to maximize the reward metric. We employ a Multi-Armed Bandit (MAB) algorithm to make optimal decisions, where tuning configuration parameters corresponds to selecting action arms.

DynamicSAT is not merely a plug-in framework but a methodology to evaluate the necessity of dynamic tuning for SAT solving heuristics. We evaluate the effectiveness of solvers enhanced with DynamicSAT and compare them with baseline solvers. Specifically, we

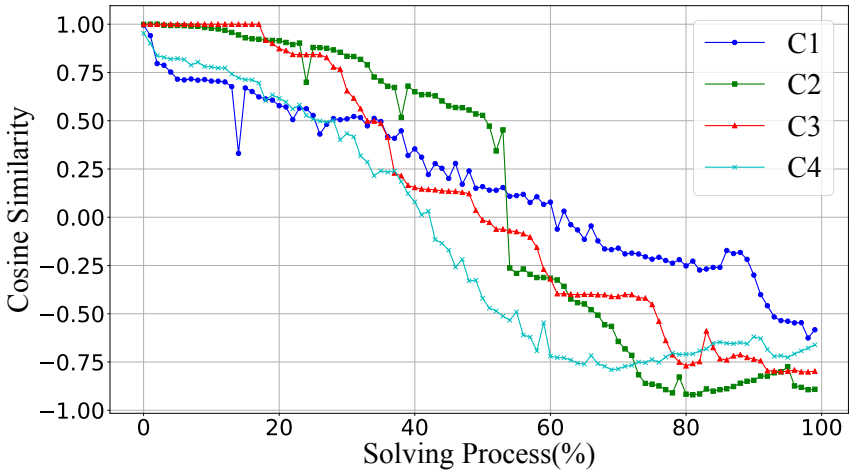


Figure 1 CNF changes during solving: We randomly select 4 problem instances (C1–C4). For each instance, we extract 100 intermediate CNFs at equal intervals throughout the solving process and generate their feature vectors using SATZilla [39]. The y-axis represents the cosine similarity between the feature vectors of each extracted intermediate CNF and the initial CNF.

tune the configurations of various modern heuristics, including chronological backtracking [32], clause vivification [23], and restart strategies [12]. The experiments are conducted on the 2024 SAT Competition benchmarks [19]. Our results demonstrate that DynamicSAT significantly improves solver efficiency, both in terms of the number of problems solved and the total solving time. Notably, our approach only adjusts the configurations of a single heuristic `compact` achieves 19.23% Par2 score reduction compared to Kissat-4.0.0, the champion of the 2024 SAT Competition [4].

We summarize the contributions of our work as follows.

- We observe and formally define the dynamic changes of problem instances during SAT solving, which may lead to the initial configuration becoming inefficient during the solving process.
- We propose a general Multi-Armed Bandit (MAB)-based approach to dynamically tune the configuration parameters of the solving algorithm during runtime. This method is versatile and can be applied to any configuration tuning of SAT solvers.
- We conduct experiments using various sets of tunable configuration parameters. The experimental results demonstrate that our approach can be easily implemented as a plug-in into baseline solver, achieving solving time reductions of up to 21.37%.

2 Related Work

2.1 Conflict-Driven Clause-Learning

The Boolean Satisfiability (SAT) problem is a fundamental problem in computer science and logic, which involves determining whether a given propositional formula is satisfiable. Specifically, the SAT problem is defined over a propositional formula in Conjunctive Normal Form (CNF). Let X be a set of propositional variables. A literal l is either a variable $x \in X$ or its negation $\neg x$. The clause C_i is defined as the disjuction of literals and CNF formula F is a conjunction of m clauses.

$$F = C_1 \wedge C_2 \wedge \ldots \wedge C_m \tag{1}$$

Solving the SAT problem is formulated as determining whether there exists an assignment $A = l_1, l_2, \ldots, l_n$ that leads the formula F to evaluate to logic-1 (*i.e.*, F is satisfied). If no such assignment exists, the formula F is said to be unsatisfiable.

Modern Boolean Satisfability solvers primarily rely on Conflict-Driven Clause-Learning (CDCL) as their core algorithm [28], supplemented by various heuristics to efficiently address this NP-hard problem. In a CDCL solver, the problem instance is represented in Conjunctive Normal Form (CNF), which consists of a conjunction of clauses, where each clause is a disjunction of literals.

Over the past decades, numerous solving heuristics have been developed to accelerate the systematic exploration of potential solutions while leveraging learned conflicts to prune the search space. For example, clause vivification [23] reduces the search space by simplifying clauses through partial resolution, effectively strengthening constraints to eliminate redundancies. Chronological backtracking [32] revisits decisions in the reverse chronological order, avoiding expensive backtracking to arbitrary points in the decision tree. Besides, restart strategy [12] periodically restart the search process to escape local minima and improve convergence on hard instances. While these heuristics and their corresponding configuration parameters significantly enhance solver performance, determining the optimal configuration before or during SAT solving remains a critical challenge.

2.2 Algorithm Configuration in SAT Solver

Currently, SAT solvers employ a range of parameterized heuristics, with solver performance on a given instance being highly dependent on the choice of parameters [18]. Identifying an optimal parameter configuration for specific instances is critical to improving performance. One representative approach focuses on determining the best configuration for a set of CNF instances from a training set [17, 2, 16]. These methods often use diverse search strategies, such as iterated local search [17], genetic algorithms [2], and model-based optimization [16], to optimize configurations for a given set of instances. However, this search process requires time-intensive trials to evaluate configuration candidates, making these methods inefficient for application on a per-instance basis.

For instances with intrinsic differences, solving them under distinct configurations is generally more effective than using a single configuration across all instances. Instance-specific algorithm configuration (ISAC) [20] addresses this by clustering CNF instances based on their features using g-means clustering. Each cluster, composed of similar CNFs, then applies a search-based configuration strategy to identify optimal parameter settings. When encountering a new instance, ISAC assigns it to the nearest cluster and applies that cluster's configuration to the instance. Malitsky *et al.* [27] further enhance ISAC by introducing adaptive clustering, allowing the model to adjust clusters as new instances are added.

While these approaches have demonstrated performance gains by selecting appropriate parameters for individual instances, they do not address the dynamic changes that occur throughout the SAT solving process. As the characteristics of a CNF evolve during solving, the most effective strategy may also shift [7].

2.3 Multi-Armed Bandit

The Multi-Armed Bandit (MAB) problem [21] is a fundamental framework in reinforcement learning and decision-making under uncertainty. It models scenarios where an agent iteratively selects actions (termed "arms", $A = \{a_1, a_2, \ldots, a_T\}$) to maximize cumulative rewards $\sum r_t(a_t)$ while balancing exploration (gathering information about under-tested

arms) and exploitation (leveraging known high-reward arms). Common approaches to solving MAB problems include algorithms such as ϵ-greedy, Upper Confidence Bound (UCB), and Thompson Sampling. These methods estimate the expected reward for each arm while efficiently managing the exploration-exploitation trade-off. For instance, UCB selects arms based on an optimistic estimate of their potential, while Thompson Sampling uses Bayesian inference to sample actions according to their probability of being optimal.

Since there is no explicit functional relationship between maximizing total expected reward and the selection of arms, MAB is particularly well-suited for addressing black-box optimization problems. This makes MAB-based approaches highly effective for dynamically adjusting configurations or strategies during the solving process, where accurately modeling the reward function is challenging. Previous work, such as Kissat-MAB [7], employs MAB to dynamically select between two state-of-the-art SAT solving heuristics: Variable State Independent Decaying Sum (VSIDS) and the Conflict History-Based (CHB) branching heuristic. Similarly, [24] formulates the decision of whether or not to trigger a solver reset as a MAB problem. However, these approaches focus narrowly on optimizing specific configurations rather than providing a generalized framework for handling arbitrary heuristics with minimal engineering effort.

In this work, we model configuration tuning as a MAB problem and propose a general reward and action formulation that supports a wide range of heuristic configuration parameters. This approach offers a versatile solution for dynamically optimizing SAT solver performance while minimizing the need for extensive engineering effort.

3 Methodology

3.1 Problem Statment

The Boolean Satisfiability (SAT) problem can be defined defined over a propositional formula in conjunctive normal form (CNF). Let the initial CNF instance be denoted as Eq (1), where C represents a clause, and m is the number of clauses. We denote the CNF formula at variable decision step t as F_t and the initial CNF is F_0. During the execution of a Conflict-Driven Clause Learning (CDCL) SAT solver, the CNF formula evolves dynamically due to unit propagation and clause learning, clause elimination mechanisms.

Unit Propagation. At each step t, the partial assignment A_t simplifies F_t through Boolean constraint propagation [30]. Formally, let $A_t = \{l_1, l_2, \ldots, l_n\}$ be the literal assigned at step t. After unit propagation, the clauses satisfied by A_t (*i.e.*, $\exists l \in C \cap A_t$) are discarded, while the unsolved clauses retain only unassigned literals:

$$F_t^{UP} = UnitProp(F_t, A_t) = \Big\{ C \setminus \{\neg l \mid l \in A_t\} \mid C \in F_t, A_t \not\models C \Big\} \tag{2}$$

Clause Learning. Following conflict analysis, the solver derives new clauses via resolution during conflict analysis [28]. Let $\mathcal{L}_t$ denote the set of learned clauses added at step t. The augmented CNF F_t^{CL} becomes Eq (3). Each learned clause $C_{learnt} \in \mathcal{L}_t$ is logically implied by F_t, where $F_t \models C_{learnt}$, but contains fewer literals than the clauses representing conflicts.

$$F_t^{CL} = F_t \cup \mathcal{L}_t \tag{3}$$

Clause Elimination. To maintain tractability, the solver periodically removes clauses deemed irrelevant by activity-based heuristics. Let $\mathcal{E}_t$ denote the set of clauses eliminated at step t, which is selected by their activity scores [3]. The CNF after elimination becomes:

$$F_t^{CE} = F_t \setminus \mathcal{E}_t \tag{4}$$

Combining the above three mechanisms, the CNF at the next step $t+1$ is defined as below. This induces a non-stationary problem instance, where F_t structurally diverges from F_0 through iterative clause modifications.

$$F_{t+1} = UnitProp(F_t, A_t) \cup \mathcal{L}_t \setminus \mathcal{E}_t \tag{5}$$

To better visualize this phenomenon, we quantify the structural divergence by analyzing the evolution of intermediate CNF features. We first randomly select 4 representative instances (C1–C4) from SAT competition benchmarks [19] and sample 100 intermediate CNF formulas F_t at uniform decision intervals (0–100% solving progress). We then employ SATZilla [39], a well-known tool has been successfully used in previous algorithm selection approaches [40, 35], to obtain feature vectors of F_t and F_0. Feature vectors represent CNF instances by incorporating various features, such as basic syntactic features (*e.g.*, clause/variable ratios), graph-based metrics (*e.g.*, nodes degree statistics of variable-clause graph) and behavioral indicators from short solver probes. We denote the feature vector $h_t = Norm(\phi(F_t))$, where $\phi(\cdot)$ represents SATZilla's feature mapping and $Norm(\cdot)$ performs zero-mean normalization. The similarity between F_t and F_0 is calculated as cosine similarity between h_t and h_0:

$$sim(F_t, F_0) = \frac{h_t \cdot h_0}{\|h_t\|\|h_0\|} \tag{6}$$

As shown in Figure 1, all four instances exhibit a decay of similarity, confirming that F_t becomes increasingly dissimilar to F_0. Consequently, this dynamic alteration can render the initial configuration inadequate for handling the evolving CNF in its intermediate state. Consequently, there is a clear motivation for dynamically adjusting the configuration parameters during the solving process.

3.2 Overview of DynamicSAT

Given the structural divergence quantified by $sim(F_t, F_0)$, the initial configuration Ω_0 becomes suboptimal for F_t. We formalize on-the-fly parameter tuning as a decision-making problem where the action is how to adjust the parameters and the goal is to reduing solving time. To achieve this, we adopt the MAB algorithm to dynamically optimize solver behavior.

Figure 2 demonstrates the proposed MAB-based solving process, where the initial problem instance is denoted as F_0 and the sampled intermediate instances are $F_1, F_2, F_3, \ldots$. When the dynamic changes in the CNF instance meet a predefined trigger condition $\Delta\|F\| \geq \theta$ (see Section 3.3.1), the MAB-based configuration tuning process is initiated, consisting of three stages: Sampling, Decision, and Free (see Section 3.4). The DynamicSAT serves as a plug-in within the SAT solver and repeats iteratively until the solver terminates to reduce solving time.

3.3 MAB Formulation

3.3.1 Trigger Condition

To determine when the problem instance has diverged sufficiently to necessitate parameter reconfiguration, we formalize a trigger condition based on the structural evolution of the CNF formula F_i.

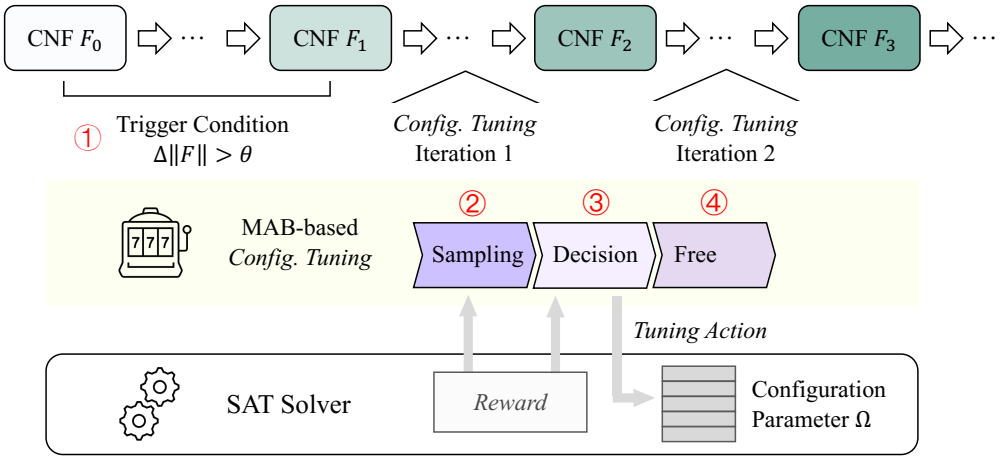


Figure 2 Overview of Dynamic Configuration Tuning.

As discussed in Section 3.1, the CDCL solver dynamically modifies F_i through unit propagation, clause learning and clause elimination, resulting in continuous additions and removals of clauses. These operations directly alter the formula's logical and topological properties, which we quantify using the number of added and removed clauses as metrics of structural change.

Let F_i to F_j denote CNF instances at steps i and j, respectively. We define:

- $\#add(F_i, F_j)$: The number of clauses added to F_i to derive F_j. These include learned clauses generated during conflict analysis.
- $\#rm(F_i, F_j)$: The number of clauses removed from F_i to derive F_j due to clause elimination and unit propagation.

The cumulative clause change between F_i and F_j is then defined as:

$$\Delta\|F_{i\to j}\| = \#add(F_i, F_j) + \#rm(F_i, F_j) \tag{7}$$

Given the current intermediate instance F_j and the instance from the previous timestamp F_i, the MAB-based configuration tuning approach is triggered if the clause changes $\Delta\|F_{i\to j}\|$ exceed the threshold θ. Once triggered, the timestamp is updated to instance F_j, ensuring that subsequent comparisons measure changes relative to F_j. It should be denoted that the initial change is defined as $\Delta\|F_{0\to j}\|$, where F_0 represents the initial instance.

3.3.2 Action Space

To enable adaptive configuration tuning in DynamicSAT, we formalize the action space as a set of discrete adjustments to solver parameters. Each action (*i.e.*, arm) a_i corresponds to a specific combination of parameter updates, which the MAB framework selects based on the evolving state F_i.

For Boolean parameters, which toggle specific heuristics on or off (e.g., enabling chronological backtracking), the tunable value v^{bool} is defined as a binary choice:

$$v^{bool} = \{0, 1\} \tag{8}$$

For integer parameters, which control thresholds or limits (e.g., backtracking depth), adjustments are constrained to three discrete actions increase its original value by 10%, decrease its by 10%, and keep it stable. It is denoted in Eq. (9), where p and q are the lower and upper bounds, respectively.

$$v^{int} = \begin{cases} \min(v^{int} * (1 + 10\%), q) \\ \max(v^{int} * (1 - 10\%), p) \\ v^{int} \end{cases} \tag{9}$$

Then, our DynamicSAT with multi-arms supports tuning multiple options to explore synergistic interactions. The action space $\|a\|$ for tuning multiple parameters is the product of individual adjustments. For example, consider tuning the chronological backtracking heuristic [32], which provides Boolean option --chrono and int option --chronolevels in the integrated SAT solver. The combined action space for these parameters is $\|a\| = 2 \cdot 3 = 6$, where each arm in action a represents a unique decision of two parameters.

3.3.3 Reward

MAB strategies rely on a reward function to evaluate the performance of the selected actions. In our framework, we define the reward to reveal how the previous action benefit for the solving process. To enhance the flexibility of DynamicSAT, we adopt a general metric that evaluates the impact of configuration tuning on reducing the SAT solver's search space. Specifically, we assess the quality of recently learned clauses.

As reward, we utilize the recent Literals Blocks Distance (LBD) value [3], a metric effectively reflects the quality of learned clauses. We define the reward of action in timestamp i as $r(a_i)$ in Eq. (10), where $\text{Avg}(\text{LBD}(i-d, i))$ represents the average LBD value observed between timestamps $i - d$ to i and d is the number of variable decision intervals in this period. Additionally, U is a constant number. Consequently, such reward function is designed to assign a higher reward when the arm selected in the previous evaluation round leads the solver to learn clauses of higher quality, thereby incentivizing actions that reduce the searching space during SAT solving.

$$r(a_i) = U - \text{Avg}(\text{LBD}(i-d, i)) \tag{10}$$

3.4 MAB Strategy

We opt for Upper Confidence Bound (UCB) algorithm [1] to implement our MAB-based configuration tuning. For each arm a, there are the following relevant parameters:

- $N(a)$ represents the number of times arm a is selected in the previous of evaluations.
- $\mathbb{E}(a) = \frac{\sum r(a)}{\#a}$ represents the expected reward of arm a over the previous of evaluations, where $\#a$ is the total number of arm a_k selected.

The confidence upper bound for each action can be expressed as Eq. (11), where i is accumulated times of MAB decision in the current configuration tuning iteration until now.

$$UCB(a) = \mathbb{E}(a) + 2\sqrt{\frac{ln(i)}{N(a)}} \tag{11}$$

The UCB algorithm selects the arm that maximizes the value on the right side of the above formula as the current action, and reduces its uncertainty as the number of selections increases, balancing exploration and exploitation.

Algorithm 1 Pseudo-code of implementing DynamicSAT in variable decision function.

Input: The number of variable decision times t

```
// Main program of variable decision
...
// DynamicSAT code starts below:
Define the interval of MAB decision making as d
// 1. Condition to trigger the decision process
if Δ‖F_{i→j}‖ ≥ θ then
    trigger ← True                                  // Set trigger to True
    no_sampling ← μ                                 // Set sampling frequency
    no_decision ← λ                                 // Set decision frequency
    i ← 0                                 // Initialize of MAB decisions counter
end
else
    trigger ← False                                 // Keep trigger to False
end
// Process when triggered
if trigger then
    // 2. Sampling Stage
    while no_sampling > 0 and t%d == 0 do
        Get r(a_i)                                  // Get reward for previous action
        Update E(a_i)                               // Update expected value in MAB
        no_sampling ← no_sampling −1
        i ← i + 1
        a_i ← Randomly select action                // Randomly select for exploration
        Update Ω with action a_i                    // Update the solving configuration
    end
    // 3. Decision Stage
    while no_sampling == 0 and no_decision > 0 and t%d == 0 do
        Get r(a_i)                                  // Get reward for previous action
        Update E(a_i)                               // Update expected value in MAB
        no_decision ← no_decision −1
        i ← i + 1
        // Make decision based on UCB value
        a_i ← arg max(UCB)                                          // See Eq. (11)
        Update Ω with action a_i                    // Update the solving configuration
    end
    // 4. Free Stage
    if no_sampling == 0 and no_decision == 0 then
        trigger ← False
        Clear UCB value
    end
end
```

In DynamicSAT, we divide each configuration tuning iteration into three stages: Sampling, Decision and Free, as shown in Algorithm 1. This approach ensures a balance between exploration and exploitation while maintaining computational efficiency and flexibility. We define the hyperparameters θ represents the formula change threshold, d is the intervals of MAB decision making and μ is the sampling frequency. In the default setting, we assign $\mu = 1000$, $\theta = 30\%$ and $d = 100$, which are explored in Section 4.2.

Once the trigger condition satisfied (Line 3), the MAB process initiates exploration by dynamically selecting actions through random sampling (Line 18). For each action a_i, the reward $r(a_i)$ is collected according to Eq. (10) (Line 14) and expected value $\mathbb{E}(a_i)$ is iteratively updated (Line 15). This stage prioritizes exploration to gather sufficient data for the current intermediate instance, ensuring the policy avoids premature convergence to suboptimal configurations.

Then, once exploration concludes (*no_sampling=0*), the algorithm shifts to exploitation. Here, the updated policy guides the selection of action arms to maximize the reward, using UCB value (Line 26, Eq. (11)). Configuration parameters Ω are adjusted based on the arm with the highest UCB value.

After both sampling and decision, the MAB-based process is disabled (Line 29). The solver keeps the current optimal configuration until the next trigger condition is met. Besides, the UCB value of each action is cleared before the next iteration.

4 Experiments

4.1 Experimental Settings

Our DynamicSAT can be seamlessly integrated into the modern SAT solvers with minimal engineering effort, while also supporting configuration tuning for arbitrary heuristics. We opt for Kissat-4.0.0 (Kissat-sc24), the winner of SAT Competition 2024 [4] as the baseline solver. To demonstrate the effectiveness of DynamicSAT, we introduce specific variants (named `DSAT-[heuristic]`) that integrate with the baseline solver and allow for the dynamic tuning of multiple heuristic options. For instance, the variant named `DSAT-chrono` refers to the solver where the options related to chronological backtracking [32] are dynamically tunable during runtime. Specifically, the baseline solver provide two options: `--chrono` and `--chronolevels`.

In our MAB setting, sampling occurs at intervals of every p decision rounds, with the decision phase triggered when the number of processed clauses reaches θ of the total clauses in the original CNFs. For each MAB iteration, DynamicSAT conducts sampling for s times before making decisions based on the UCB value. To ensure consistency, we use the same random seed across all experiments, eliminating the effect of randomness. All experiments are conducted on an Intel(R) Xeon(R) CPU E7-8860 v4 @ 2.20GHz with 128GB memory. The timeout is set to 5,000 seconds of CPU time.

We test the effectiveness of DynamicSAT on the 2024 SAT Competition Benchmark [19] and evaluate using three metrics.

- **Problem Solved** (#Solved): The number of problem instances that can be solved within time limit. A higher value indicates better solver performance.
- **Solving Time of Solvable Instance** (Time): The average runtime for successfully solved instances by baseline solver, which investigate whether DynamicSAT negatively impacts the performance on these relative easy instances. A lower value indicates better solver performance.

Table 1 The results of hyperparameters exploration.

Initial Sampling Frequency			Clause Change Threshold			Decision Interval		
μ	#Solved	Par2	θ	#Solved	Par2	d	#Solved	Par2
10	308	2,852.60	10%	300	3,038.82	10	305	2,841.36
100	303	2,953.52	**30%**	**305**	**2,910.31**	**100**	**310**	**2,721.20**
1,000	**310**	**2,824.76**	50%	303	2,964.96	1,000	308	2,751.27
10,000	305	2,875.46	100%	302	2,966.27	10,000	308	2,760.65

- **Average Par2 Score** (Par2): The Par2 score sums the solving time for successfully solved instances and applies a penalty of twice the time limit for instances that timeout. In our experiments, we report the average Par2 score. A lower Par2 score indicates better solver performance.

4.2 Hyperparameter Analysis of DynamicSAT

DynamicSAT's performance hinges on three critical hyperparameters: the initial sampling frequency (μ), the clause change threshold (θ), and the decision interval (d). We analyze their impact using `DSAT-chrono` on the the benchmark, with results summarized in Table 1. All experiments fix two hyperparameters while varying the third, with results summarized in Table 1.

Initial Sampling Frequency. The initial sampling frequency μ determines how many arms (actions) are tested during the sampling stage when the MAB framework is first activated. A higher μ increase early-stage exploration, refining reward estimates $\mathbb{E}(a)$, but increases computational overhead. From Table 1, we observe that $\mu = 1,000$ achieves the best performance: it solves 310 instances (vs. 308 for $\mu = 10$) with the lowest Par2 score 2,824.76. Smaller values ($\mu = 10$ and $\mu = 100$) under-explore, leading to suboptimal policy convergence, while $\mu = 10,00$ over-samples, wasting runtime on redundant exploration. Therefore, moderate early-stage sampling benefits reward estimation accuracy without sacrificing exploitation efficiency, where we assign $\mu = 1,000$ as the default setting.

Clause Change Threshold. The clause change threshold θ triggers MAB-based configuration tuning when the normalized divergence $\Delta\|F_{i\to j}\| \geq \theta$. First, a lower $\theta = 10\%$ results increasing sensitivity to formula dynamics but risks overreacting to noise. For instance, the variant with $\theta = 10\%$ demonstrates the worst performance, solving only 300 instances. Second, a higher threshold $\theta = 50\%$ or $\theta = 100\%$ is too insensitive to changes, resulting in suboptimal performance. Third, at $\theta = 30\%$, DynamicSAT solves 305 instances with a Par2 of 2,910.31, outperforming the others. This suggests $\theta = 30\%$ optimally filters transient clause fluctuations while capturing critical structural shifts.

Decision Interval. The decision interval d indicates the number of variable decision steps (*i.e.*, the times entry main program of variable decision in SAT solver) between consecutive MAB decision. Since our reward function depends on the quality of learned clauses over a period, shorter intervals may result in insufficient clause learning, while longer intervals risk delayed adaptation. From Table 1, $d = 100$ achieves the best Par2 score (2,721.20) while solving 310 instances. In contrast, $d = 10$ degrades performance by 4.42% in Par2 score due to the frequent reward access, while $d = 1,000$ and $d = 10,000$ causes stale policies, resulting suboptimal performance. Therefore, $d = 100$ is adopted as the default in the subsequent experiments.

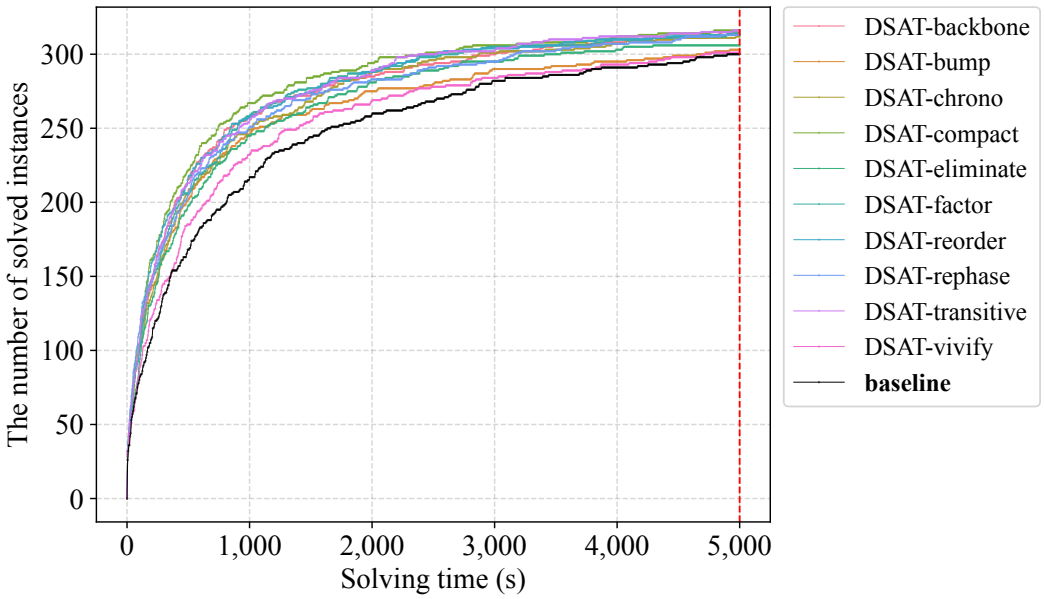


Figure 3 Number of solved instances as a function of solving time.

4.3 Effect of DynamicSAT

To quantify the impact of integrating DynamicSAT on various in-processing heuristics[1], we choose 10 heuristic strategies randomly and evaluate the corresponding DynamicSAT variants. The complete list of tunable heuristics and their corresponding options is provided in Table 7 in Appendix A.1. Table 2 compares their performance against the baseline solver on the 2024 SAT Competition Benchmark. It should be noted that the runtime includes DynamicSAT, even though it occupies a small portion of the overall time (see discussion in Appendix A.3). We conclude three observations from the results.

First, the results reveal that all 10 solver variants powered by our proposed DynamicSAT consistently outperform the baseline solver across all metrics. On average, DynamicSAT achieves a 3.63% increase in #Solved (310.90 vs. 300), 32.30% solving time reduction (568.49 vs. 839.70) on solvable instances and 13.42% Par2 reduction (2,709.68 vs. 3,129.77), confirming the robustness of our approach across diverse heuristics. Besides, Figure 3 further illustrates the superiority of DynamicSAT: the curves of all 10 variants rise steadily over time, consistently surpassing the baseline's plateau.

Second, the most significant improvements arise from heuristics directly interacting with the clause database. Notably, the top-performing variant `DSAT-compact` solve 5.33% more instances than the baseline, with a 19.23% reduction in Par2 score, demonstrating its superior efficiency and effectiveness. Simiarlly, `DSAT-transitive` also achieves 5.33% more instances solved. These strategies adaptively prune redundant clauses and simplify implication graphs during critical phases of structural divergence [6]. Since these heuristics are more sensitive to the structural CNF changes, tuning their corresponding parameters ensures optimal configuration.

[1] We do not adjust the pre-processing heuristics (e.g., `fastel` and `lucky`), as they are executed only once before the solving process.

Table 2 Performance comparison between baseline and DynamicSAT.

	#Solved	Impv.	Time	Red.	Par2	Red.
baseline	300		839.70		3,129.77	
DSAT-backbone	315	5.00%	473.87	43.57%	2,616.12	16.41%
DSAT-bump	303	1.00%	715.23	14.82%	2,916.44	6.82%
DSAT-chrono	312	4.00%	595.24	29.11%	2,691.16	14.01%
DSAT-compact	**316**	**5.33%**	**444.16**	**47.10%**	**2,528.06**	**19.23%**
DSAT-eliminate	306	2.00%	687.71	18.10%	2,828.60	9.62%
DSAT-factor	314	4.67%	500.34	40.41%	2,618.74	16.33%
DSAT-reorder	313	4.33%	471.75	43.82%	2,616.13	16.41%
DSAT-rephase	313	4.33%	540.49	35.63%	2,678.69	14.41%
DSAT-transitive	**316**	**5.33%**	471.89	43.80%	2,574.10	17.75%
DSAT-vivify	301	0.33%	784.17	6.61%	3,028.80	3.23%
Avg.	**310.9**	**3.63%**	**568.49**	**32.30%**	**2,709.68**	**13.42%**

Table 3 Performance comparison between baseline and DynamicSAT on SAT instances.

Solver	#Solved	Impv.	Time	Red.	Par2	Red.
baseline	149		660.49		1,462.65	
DSAT-backbone	154	3.36%	376.50	43.00%	987.05	32.52%
DSAT-bump	149	0.00%	488.71	26.01%	1,210.50	17.24%
DSAT-chrono	152	2.01%	527.31	20.16%	1,071.29	26.76%
DSAT-compact	155	4.03%	353.18	46.53%	**849.92**	**41.89%**
DSAT-eliminate	150	0.67%	630.54	4.53%	1,223.24	16.37%
DSAT-factor	155	4.03%	356.00	46.10%	885.90	39.43%
DSAT-reorder	153	2.68%	365.93	44.60%	981.02	32.93%
DSAT-rephase	153	2.68%	366.75	44.47%	982.87	32.80%
DSAT-transitive	**156**	**4.70%**	**347.25**	**47.43%**	853.90	41.62%
DSAT-vivify	153	2.68%	537.54	18.61%	1,122.96	23.22%
Avg.	**153.0**	**2.68%**	**434.97**	**34.14%**	**1,016.87**	**30.48%**

Third, heuristics focusing on localized variable operations exhibit marginal gains, where `DSAT-vivify` and `DSAT-bump` only reduce Par2 score by 3.23% and 6.82%, respectively. Such strategies, which optimize literal-level properties or activity scores, lack systemic alignment with the solver's evolving state [25, 23]. Their localized impact and insensitivity to structural CNF shifts prevent them from benefiting meaningfully from dynamic adaptation. Moreover, we further analyze that DSAT-vivify performs particularly worse on UNSAT cases, as detailed in Section 4.4.

Therefore, our experiments highlight that DynamicSAT excels when the tunable heuristics interact with global structures but struggles with contextually isolated optimizations. Our DynamicSAT also supports tuning more options with complex interactions, which is considered as the future work and briefly discussed in Appendix A.2.

Table 4 Performance comparison between baseline and DynamicSAT on UNSAT instances.

Solver	#Solved	Impv.	Time	Red.	Par2	Red.
baseline	151		1,016.53		1,778.76	
DSAT-backbone	161	6.62%	569.94	43.93%	1,003.39	43.59%
DSAT-bump	154	1.99%	938.74	7.65%	1,510.69	15.07%
DSAT-chrono	160	5.96%	662.28	34.85%	1,102.08	38.04%
DSAT-compact	**161**	**6.62%**	**533.93**	**47.48%**	**925.36**	**47.98%**
DSAT-eliminate	156	3.31%	744.12	26.80%	1,285.17	27.75%
DSAT-factor	159	5.30%	642.77	36.77%	1,109.66	37.62%
DSAT-reorder	160	5.96%	576.17	43.32%	1,009.37	43.25%
DSAT-rephase	160	5.96%	711.94	29.96%	1,159.21	34.83%
DSAT-transitive	**160**	**5.96%**	**594.88**	**41.48%**	1,033.06	41.92%
DSAT-vivify	148	-1.99%	1,027.53	-1.08%	1,869.57	-5.11%
Avg.	**157.9**	**4.57%**	**700.23**	**31.12%**	**1,200.76**	**32.49%**

4.4 Failure Case Analysis

We investigate the reasons behind the the negative impacts observed with `DSAT-vivify` in this subsection. To further dissect the efficacy of DynamicSAT, we analyze its performance separately on satisfiable (SAT) and unsatisfiable (UNSAT) instances from the 2024 SAT Competition Benchmark. This distinction is critical, as SAT and UNSAT instances impose fundamentally different demands on solvers: SAT solutions require efficient search space navigation, while UNSAT proofs depend on conflict analysis and clause learning [33].

For satisfiable instances, DynamicSAT variants demonstrate substantial efficiency gains. In Table 3, adjusting heuristics `compact` achieves 41.89% Par2 reduction, while `DSAT-transitive` reduces the average solving time by 47.43% compared to the baseline. Although some variants, such as `DSAT-bump` and `DSAT-eliminate`, show only minor improvements in the number of solved instances, they remain effective by achieving significant reductions in both Par2 scores and solving time. For example, `DSAT-bump` does not solve any additional instances but still reduces the Par2 score by 17.24% and the solving time by 26.01%.

In contrast, unsatisfiable instances reveal a more nuanced performance landscape. While most DynamicSAT variants still outperform the baseline, variable-centric heuristics like `DSAT-vivify` exhibit limited gains, which aligns with the results in Section 4.3. To be specific, the worst-performing variant, `DSAT-vivify`, solves 1.99% fewer instances than the baseline solver and shows negative gains in solving time and Par2 score, with increases of 1.08% and 5.11%, respectively.

We attribute such limited efficacy to the inherent optimizations on the corresponding heuristic. As reported in the solver specification [4], the optimized and simplified vivification is one of the most significant update appeared in the baseline solver. This suggests that human expertise has largely saturated vivification's optimization potential, while dynamic tuning it may gain diminish for highly engineered tactics. To further prove our hypothesis, we integrate DynamicSAT into the kissat solver without meticulous optimization on vivification, using the version `sc2022-bulky` [11] submitted to SAT Competition 2022 (denoted as kissat-sc22). The intergated variant is named as `DSAT-vivify-sc22`. The following experiments are conducted on the 2024 SAT Competition Benchmark under the same environment and hyperparameters as described earlier.

Table 5 Performance comparison between kissat-sc22, kissat-sc24 and DSAT-vivify-sc22.

Solver	#Solved	Impv.	Time	Red.	Par2	Red.
Kissat-sc22	294		944.95		3,344.54	
Kissat-sc24	300	2.04%	1,087.49	-15.08%	3,129.77	**6.42%**
DSAT-vivify-sc22	295	0.34%	787.34	16.68%	3,155.88	**5.64%**

Table 5 summarizes the results. On the one hand, the variant with DynamicSAT `DSAT-vivify-sc22` achieves a 16.68% solving time reduction for solvable instances and a 5.64% reduction in Par2 score across the entire benchmark compared to `Kissat-sc22`, which highlights the effectiveness of DynamicSAT. On the other hand, the upgrade from `Kissat-sc22` to `Kissat-sc24` achieves a 2.04% increase in #Solved and a 6.42% reduction in Par2 score. Although `DSAT-vivify-sc22`, which simply combines DynamicSAT into `Kissat-sc22`, does not significantly improve #Solved, it achieves a 5.64% Par2 reduction, which is comparable to the improvement seen in `Kissat-sc24`. Our experimental results underscore the complementary roles of human expertise and automated adaptation in solver design, demonstrating that DynamicSAT can significantly enhance performance even without meticulous heuristic-specific optimization.

4.5 Instance Family Analysis

We present a breakdown analysis of instance families alongside the corresponding solver variants. Detailed results are provided in Table 8 in the Appendix A.4. Below, we discuss the performance trends observed for representative tuning heuristics.

First, `DSAT-backbone` and `DSAT-transitive` are the variants that support dynamic tuning of binary clause–related heuristics. The former focuses on analyzing binary clause backbones, while the latter applies transitive reduction to binary clauses. In families with a high density of binary clauses, such as Hamiltonian, Maxsat, and Miter, these heuristics yield substantial improvements. For instance, in the Hamiltonian family, the baseline solver achieves an average Par2 score of 404.76. `DSAT-backbone` reduces this score to 243.36 (a 39.88% improvement), and `DSAT-transitive` further lowers it to 198.25 (a 51.02% improvement). In contrast, `DSAT-chrono` only achieves an 8.93% Par2 reduction on Hamiltonian instances, and `DSAT-eliminate` even increases the Par2 score by 70.96%.

Second, for Cryptography instances, the baseline Par2 score is 1,310.32, and most tuning approaches yield marginal or negative improvements – for example, `DSAT-chrono` worsens performance by 57.13% and `DSAT-rephase` increases the Par2 score by 9.50%. Surprisingly, `DSAT-backbone` reduces the Par2 score by 68.39% on this instance family. Although cryptography instances do not present an abundance of explicit binary clauses, many variable relationships can be modeled as equality or inequality conditions that reflect the XOR structure. During the clause learning process, the solver adds numerous binary clauses representing these conditions. By dynamically tuning the backbone heuristic, the solver efficiently identifies and exploits these critical binary relationships, which leads to significant performance gains. This result highlights the adaptive capability of DynamicSAT to tailor tuning strategies to the dynamic properties of the instance family.

Third, our further experiments with a separated analysis across instance families align with the results in Section 4.4, where the variant `DSAT-vivify` does not significantly outperform the baseline. While `DSAT-vivify` reduces the Par2 score across most instance families, it struggles with solving Random Circuit, Miter, and Hamiltonian instances. Specifically, it even increases Par2 score on Random Circuit problems by 133.37%.

As a result, most DynamicSAT variants achieve remarkable performance on a broad range of problems. Moreover, our results clearly expose the specific instance families where DynamicSAT variants are less effective. In future work, we aim to refine our heuristics and further adapt the DynamicSAT framework to better address the limitations identified in these challenging instance families.

5 Conclusion

This paper introduces DynamicSAT, a framework for dynamically tuning configuration parameters on-the-fly within a SAT solver. Our work builds on the observation that CNF instances undergo significant changes during the solving process, resulting in a substantial difference between intermediate instances and the initial problem formulation. To address this, DynamicSAT employs a multi-armed bandit (MAB)-based agent to explore optimal configurations that adapt to the evolving characteristics of CNF instances. By integrating DynamicSAT into a baseline SAT solver, we achieve Par2 reductions of up to 19.23% on the 2024 SAT competition benchmarks. DynamicSAT is flexible and can tune arbitrary configurations of various SAT solvers with minimal engineering effort. We believe that this approach could become an efficient and general tool for advancing SAT solving frameworks.

As future work, several avenues can be explored to enhance DynamicSAT. First, it would be beneficial to investigate more effective settings in our MAB agent, including exploring alternative reward functions and considering other MAB algorithms (e.g. ϵ-greedy and Thompson Sampling). Additionally, we should explore more effective approaches to simultaneously tune multiple heuristics, leveraging their synergistic effects to further enhance solver performance across diverse instance families. Finally, we aim to generalize the proposed DynamicSAT framework and integrate it into other modern SAT solvers.

References

1 Rajeev Agrawal. Sample mean based index policies by o (log n) regret for the multi-armed bandit problem. *Advances in applied probability*, 27(4):1054–1078, 1995.

2 Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, pages 142–157, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-04244-7_14.

3 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Twenty-first international joint conference on artificial intelligence*. Citeseer, 2009.

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, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

6 Ronen I Brafman. A simplifier for propositional formulas with many binary clauses. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(1):52–59, 2004. doi:10.1109/TSMCB.2002.805807.

7 Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. Combining vsids and chb using restarts in sat. In *27th International Conference on Principles and Practice of Constraint Programming*, 2021.

8 Stephen A Cook. The complexity of theorem-proving procedures. In *Logic, automata, and computational complexity: The works of Stephen A. Cook*, pages 143–152. 2023. doi:10.1145/3588287.3588297.

9 Stefan Falkner, Marius Lindauer, and Frank Hutter. Spysmac: Automated configuration and performance analysis of sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 215–222. Springer, 2015. doi:10.1007/978-3-319-24318-4_16.

10 ABKFM Fleury and Maximilian Heisinger. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. *SAT COMPETITION*, 2020:50, 2020.

11 Armin Biere Mathias Fleury. Gimsatul, isasat, kissat. *SAT COMPETITION*, 10, 2022.

12 Carla P Gomes, Bart Selman, Henry Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98(1998):431–437, 1998.

13 Marijn Heule, Matti Järvisalo, Florian Lonsing, Martina Seidl, and Armin Biere. Clause elimination for sat and qsat. *Journal of Artificial Intelligence Research*, 53:127–168, 2015. doi:10.1613/JAIR.4694.

14 Andrei Horbach. A boolean satisfiability approach to the resource-constrained project scheduling problem. *Annals of Operations Research*, 181:89–107, 2010. doi:10.1007/S10479-010-0693-2.

15 Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5*, pages 507–523. Springer, 2011. doi:10.1007/978-3-642-25566-3_40.

16 Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, pages 507–523, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-25566-3_40.

17 Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, October 2009. doi:10.1613/JAIR.2861.

18 Frank Hutter, Marius Lindauer, Adrian Balint, Sam Bayless, Holger Hoos, and Kevin Leyton-Brown. The configurable sat solver challenge (cssc). *Artificial Intelligence*, 243:1–25, 2017. doi:10.1016/j.artint.2016.09.006.

19 Markus Iser and Christoph Jabs. Global Benchmark Database. 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 18:1–18:10, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SAT.2024.18.

20 Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac–instance-specific algorithm configuration. In *ECAI 2010*, pages 751–756. IOS Press, 2010. doi:10.3233/978-1-60750-606-5-751.

21 Michael N Katehakis and Arthur F Veinott Jr. The multi-armed bandit problem: decomposition and computation. *Mathematics of Operations Research*, 12(2):262–268, 1987. doi:10.1287/MOOR.12.2.262.

22 Henry Kautz and Bart Selman. Unifying sat-based and graph-based planning. In *IJCAI*, volume 99, pages 318–325, 1999. URL: http://ijcai.org/Proceedings/99-1/Papers/047.pdf.

23 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*, 279:103197, 2020. doi:10.1016/J.ARTINT.2019.103197.

24 Chunxiao Li, Charlie Liu, Jonathan Chung, Piyush Jha, Vijay Ganesh, et al. A reinforcement learning based reset policy for cdcl sat solvers. *arXiv preprint arXiv:2404.03753*, 2024.

25 Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers. In *Hardware*

and Software: Verification and Testing: 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings 11, pages 225–241. Springer, 2015. doi:10.1007/978-3-319-26287-1_14.

26 Hongduo Liu, Peng Xu, Yuan Pu, Lihao Yin, Hui-Ling Zhen, Mingxuan Yuan, Tsung-Yi Ho, and Bei Yu. Neuroselect: Learning to select clauses in sat solvers. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6, 2024. doi:10.1145/3649329.3656250.

27 Yuri Malitsky, Deepak Mehta, and Barry O'Sullivan. Evolving instance specific algorithm configuration. In *Proceedings of the International Symposium on Combinatorial Search*, volume 4, pages 133–140, 2013. doi:10.1609/SOCS.V4I1.18296.

28 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.

29 João P Marques-Silva and Karem A Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of the 37th Annual Design Automation Conference*, pages 675–680, 2000. doi:10.1145/337292.337611.

30 David A McAllester. Truth maintenance. In *AAAI*, volume 90, pages 1109–1116, 1990. URL: http://www.aaai.org/Library/AAAI/1990/aaai90-164.php.

31 Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001. doi:10.1145/378239.379017.

32 Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In *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 21*, pages 111–121. Springer, 2018. doi:10.1007/978-3-319-94144-8_7.

33 Chanseok Oh. Between sat and unsat: the fundamental difference in cdcl sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 307–323. Springer, 2015. doi:10.1007/978-3-319-24318-4_23.

34 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.

35 Hadar Shavit and Holger H Hoos. Revisiting satzilla features in 2024. In *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, pages 27–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

36 Zhengyuan Shi, Min Li, Yi Liu, Sadaf Khan, Junhua Huang, Hui-Ling Zhen, Mingxuan Yuan, and Qiang Xu. Satformer: Transformer-based unsat core learning. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–4. IEEE, 2023. doi:10.1109/ICCAD57390.2023.10323731.

37 Zhengyuan Shi, Hongyang Pan, Sadaf Khan, Min Li, Yi Liu, Junhua Huang, Hui-Ling Zhen, Mingxuan Yuan, Zhufei Chu, and Qiang Xu. Deepgate2: Functionality-aware circuit representation learning. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2023. doi:10.1109/ICCAD57390.2023.10323798.

38 Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.

39 Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008. doi:10.1613/JAIR.2490.

40 Lin Xu, Frank Hutter, Jonathan Shen, Holger H Hoos, and Kevin Leyton-Brown. Satzilla2012: Improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge*, pages 57–58, 2012.

A Appendix

A.1 Tunable Options in Kissat Solver

See Table 7. For each heuristic, our DynamicSAT tunes the corresponding configurations, which include all the relevant options in the baseline solver.

A.2 Effect of DynamicSAT on Multiple Configuration Tuning

We employ DynamicSAT for the simultaneous tuning of options across multiple heuristics. Given the large number of possible combinations of tunable heuristics, we focus on two representative sets:

- `DSAT-Set1`. Combines variable decision heuristics `reorder` and `rephase`. These heuristics exhibit high interdependence in variable decision-making, as `reorder` alters the sequence of decision variables while `rephase` directly impacts the polarity choices during branching.
- `DSAT-Set2`. Integrates heuristics related to binary clause simplification. The `backbone` heuristic targets stable variable assignments in binary clauses, while the `transitive` heuristic simplifies binary clauses through transitive reduction.

Table 6 Performance of DynamicSAT with multiple configuration tuning.

Solver	#Solved	Impv.	Time	Red.	Par2	Red.
baseline	300		839.70		3,129.77	
DSAT-reorder	313	4.33%	471.75	43.82%	2,616.13	16.41%
DSAT-rephase	313	4.33%	540.49	35.63%	2,678.69	14.41%
DSAT-Set1	315	5.00%	542.36	35.41%	2,616.70	16.39%
DSAT-backbone	315	5.00%	473.87	43.57%	2,616.12	16.41%
DSAT-transitive	316	5.33%	471.89	43.80%	2,574.10	17.75%
DSAT-Set2	317	0.63%	478.80	42.98%	2,586.11	17.37%

Table 6 compares the performance of these sets against their individual components and the baseline. We observe that jointly tuning interdependent heuristics improves the number of solved instances. Specifically, `DSAT-Set1` solves 315 instances, outperforming the individual tuning of single heuristic (`DSAT-reorder` and `DSAT-rephase`), which solve 313 instances each. However, the results also indicate that jointly tuning heuristics does not consistently show a positive synergistic effect on Par2 reduction and solving time reduction compared to tuning individual heuristics.

To further investigate the reasons behind the observed performance patterns, we analyze the number of solved instances as a function of a partial range of solving time. Figure 4 reveals that while `DSAT-Set1` ultimately solves more instances than individual heuristics, its early-stage inefficiency limits overall gains. For solving time $\leq$ 1,000s, standalone heuristics (`DSAT-reorder`, `DSAT-rephase`) resolve more instances. For example, `DSAT-reorder` solves 258 instances within the 1,000s time limit, while `DSAT-Set1` only solves 251 instances. Beyond 4,000 seconds, `DSAT-Set1` overtakes individual heuristics. This dichotomy stems from the insufficient exploration in the MAB-based framework under tight time constraints. When integrating multiple heuristics, `DSAT-Set1` faces a larger action space due to the increased number of tunable options. As solving time increases, the accumulated exploration allows the MAB framework to refine more accurate expected rewards, enabling `DSAT-Set1` to select

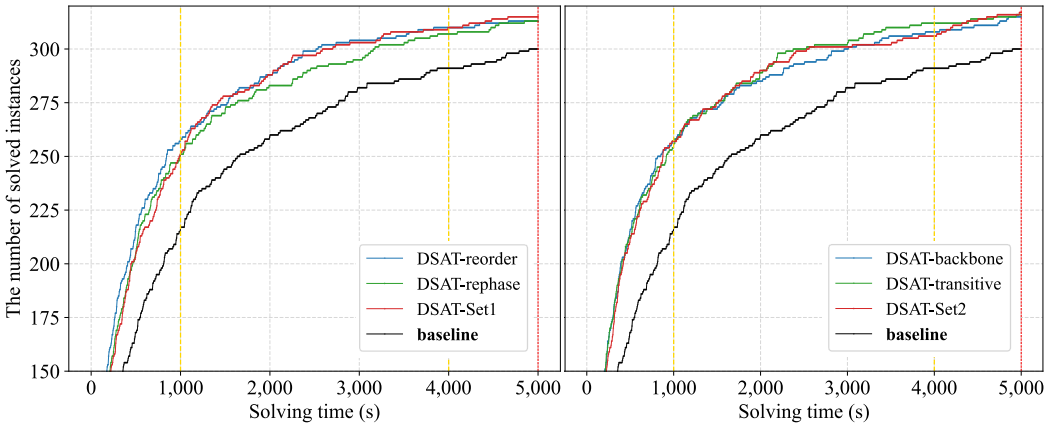


Figure 4 Performance comparison between tuning single and multiple heuristics.

more suitable actions. The same observation can be found in `DSAT-Set2`. Therefore, the experimental results suggest that DynamicSAT could benefit more on hard problem instances given the multiple tunable heuristics.

Due to page limitations, we only investigate the effectiveness and generalization ability of DynamicSAT across various heuristics in this work, leaving the exploration of synergies for future work.

A.3 Runtime Analysis

In this subsection, we quantify the computational overhead introduced by the MAB-based configuration tuning process relative to the overall SAT solving runtime. To this end, we instrument our solver implementation to record runtime spent in dynamic configuration tuning, including the Sampling, Decision, and Free stages of the MAB agent (see Line 12-32 in Algorithm 1). The experiments are conducted using the `DSAT-compact` variant across the 400 instances in the 2024 SAT Competition Benchmark. The experimental settings remain consistent with those described in previous sections.

Out of the 400 instances, our solver successfully solves 316 cases, which are included in the subsequent runtime analysis. We compare the total runtime of our solver with that of the baseline DynamicSAT system and present the results in Figure 5. The runtime of DynamicSAT exhibits a near-linear relationship with the overall solver runtime, indicating consistent computational behavior across instances. Furthermore, the cumulative time spent specifically on DynamicSAT is only 227.87 seconds, compared to a total solver runtime of 171,222.46 seconds across all 316 solved instances. This accounts for merely 0.13% of the total execution time, indicating that the overhead introduced by the MAB-based tuning mechanism is negligible in practice.

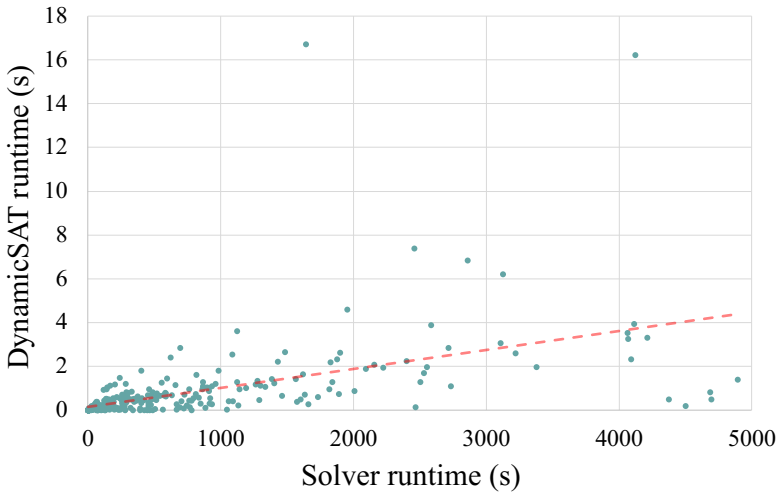


Figure 5 Runtime comparison between solver and DynamicSAT.

A.4 Instance Families

See Table 8. We only list the representative families with a significant number of problem instances. The families containing fewer instances and uncategorized instances, are not included in this analysis.

Table 7 Tunable Heuristic Configurations and Parameters.

Heuristic	Configuration	Type	Range	Default	Function
backbone	backbone	Bool	[0,1]	1	Enable backbone analysis
	backboneeffort	Int	[0,1e5]	20	Effort level
	backbonemaxrounds	Int	[1,1e6]	1k	Maximum analysis rounds
	backbonerounds	Int	[1,1e6]	100	Base analysis rounds
bump	bump	Bool	[0,1]	1	Enable clause bump
	bumpreasons	Bool	[0,1]	1	Track bump reasons
	bumpreasonslimit	Int	[1,10]	10	Reason list size limit
	bumpreasonsrate	Int	[1,10]	10	Reason decay rate
chrono	chrono	Bool	[0,1]	1	Chronological backtracking
	chronolevels	Int	[0,1e7]	100	Maximum jumped levels
compact	compact	Bool	[0,1]	1	Compaction strategy
	compactlim	Int	[0,100]	10	Compaction threshold
eliminate	eliminate	Bool	[0,1]	1	Enable eliminate strategy
	eliminatebound	Int	[0,8k]	16	Variable elimination bound
	eliminateclslim	Int	[1,100]	100	Clause size limit
	eliminateeffort	Int	[0,100]	100	Elimination effort
	eliminateinit	Int	[0,500]	500	Initial elimination interval
	eliminateint	Int	[10,500]	500	Stable elimination interval
	eliminateocclim	Int	[0,2k]	2k	Literal occurrence limit
	eliminaterounds	Int	[1,1e3]	2	Maximum elimination rounds
factor	factor	Bool	[0,1]	1	Enable factor strategy
	factorcandrounds	Int	[0,2]	2	Candidate selection rounds
	factoreffort	Int	[0,1e6]	50	Factorization effort
	factorhops	Int	[1,10]	3	Hyper binary resolution hops
	factoriniticks	Int	[700,1e6]	700	Initial conflict ticks
	factorsize	Int	[5,100]	5	Maximum factor size
	factorstructural	Bool	[0,1]	0	Structural factorization
reorder	reorder	Bool	[0,1]	1	Enable reorder strategy
	reorderinit	Int	[0,1e5]	10k	Initial reorder interval
	reorderint	Int	[1,1e5]	10k	Stable reorder interval
	reordermaxsize	Int	[2,256]	100	Maximum cache size
rephase	rephase	Bool	[0,1]	1	Enable rephase strategy
	rephaseinit	Int	[10,1e5]	1k	Initial phase reset interval
	rephaseint	Int	[10,1e5]	1k	Stable phase reset interval
transitive	transitive	Bool	[0,1]	1	Enable transitive strategy
	transitiveeffort	Int	[0,2k]	20	Transitive reduction effort
	transitivekeep	Bool	[0,1]	1	Keep transitive clauses
vivify	vivify	Bool	[0,1]	1	Enable clause vivification
	vivifyeffort	Int	[0,1e3]	100	Vivification effort level
	vivifyirred	Int	[1,100]	1	Irredundant clause effort

Table 8 Par2 Score comparison of different variants across instance families in 2024 SAT Competition Benchmark.

Family	Count	baseline	DSAT-backbone		DSAT-bump		DSAT-chrono		DSAT-compact		DSAT-eliminate	
			Par2	Red.	Par2	Red.	Par2	Red.	Par2	Red.	Par2	Red.
Cryptography	14	1,310.32	414.24	68.39%	659.13	49.70%	2,058.86	-57.13%	1,145.65	12.57%	1,216.48	7.16%
Scheduling	28	1,183.02	830.43	29.80%	700.52	40.79%	815.77	31.04%	1,056.72	10.68%	814.91	31.12%
Maxsat	13	2,014.63	907.72	54.94%	1,184.61	41.20%	1,162.92	42.28%	717.49	64.39%	1,061.59	47.31%
Software verification	13	609.20	408.20	32.99%	390.84	35.84%	929.61	-52.60%	347.31	42.99%	398.08	34.65%
Random circuits	15	1,243.49	1,029.34	17.22%	1,114.68	10.36%	905.38	27.19%	913.58	26.53%	1,068.19	14.10%
Miter	40	1,010.72	618.64	38.79%	948.85	6.12%	606.90	39.95%	580.50	42.57%	688.30	31.90%
Hamiltonian	41	404.76	243.36	39.88%	546.41	-35.00%	368.63	8.93%	233.82	42.23%	691.99	-70.96%
Argumentation	21	1,657.23	858.34	48.21%	2,359.05	-42.35%	1,013.27	38.86%	831.37	49.83%	1,044.30	36.98%
Independent set	7	2,373.78	1,985.18	16.37%	2,087.52	12.06%	1,788.86	24.64%	1,788.56	24.65%	1,828.35	22.98%
Family	Count	baseline	DSAT-factor		DSAT-reorder		DSAT-rephase		DSAT-transitive		DSAT-vivify	
			Par2	Red.	Par2	Red.	Par2	Red.	Par2	Red.	Par2	Red.
Cryptography	14	1,310.32	563.59	56.99%	1,296.83	1.03%	1,434.81	-9.50%	546.71	58.28%	508.96	61.16%
Scheduling	28	1,183.02	815.79	31.04%	797.84	32.56%	887.24	25.00%	909.80	23.10%	1,045.13	11.66%
Maxsat	13	2,014.63	919.30	54.37%	822.33	59.18%	1,157.78	42.53%	850.46	57.79%	1,310.09	34.97%
Software verification	13	609.20	431.53	29.16%	364.91	40.10%	475.02	22.03%	394.05	35.32%	518.77	14.84%
Random circuits	15	1,243.49	925.03	25.61%	958.47	22.92%	1,003.53	19.30%	886.37	28.72%	2,901.92	-133.37%
Miter	40	1,010.72	637.64	36.91%	576.27	42.98%	729.32	27.84%	591.76	41.45%	1,232.54	-21.95%
Hamiltonian	41	404.76	328.48	18.85%	274.95	32.07%	305.72	24.47%	198.25	51.02%	472.29	-16.68%
Argumentation	21	1,657.23	914.95	44.79%	822.04	50.40%	1,099.66	33.64%	943.72	43.05%	1,524.36	8.02%
Independent set	7	2,373.78	1,931.14	18.65%	1,847.57	22.17%	1,870.54	21.20%	1,869.59	21.24%	2,044.19	13.88%

Unite and Lead: Finding Disjunctive Cliques for Scheduling Problems

Konstantin Sidorov[1]
Delft University of Technology, The Netherlands

Imko Marijnissen[1]
Delft University of Technology, The Netherlands

Emir Demirović
Delft University of Technology, The Netherlands

Abstract

Constraint programming solvers have seen much success in scheduling problems owing to their efficient reasoning over constraints to solve complex problems in practice. Many algorithms have been proposed for propagating information from a *single* constraint. However, inferring and exchanging information across multiple constraints can provide deeper insight into the *global* structure of a problem. In this work, we propose to exchange information amongst constraints by inferring the disjointness of tasks in scheduling problems from *many* constraints. We do this by (i) augmenting existing propagators, such as the `Cumulative` and nogoods, to report when pairs of tasks are disjoint, and (ii) leveraging this information by introducing the `SelectiveDisjunctive` propagator which generates a lower bound on the earliest completion time of cliques of disjoint tasks to determine conflicts. This allows us to aggregate disjointness information spanning multiple constraints to gain a better global overview of the problem, as well as more precise local information. We also identify a problem structure where an LCG solver reasoning over `Cumulative` constraints separately, without any reformulations, requires an exponential amount of time to prove infeasibility, which we both justify theoretically and show empirically; on the other hand, our approach solves those instances in polynomial time. On particular known RCPSP and RCPSP/max benchmarks, our approach significantly reduces the number of conflicts required to prove optimality when resource contention is high. Additionally, we discover new lower bounds for 16 RCPSP/max instances (closing six of them) and four RCPSP instances (closing one), as well as new upper bounds for two RCPSP/max instances and four RCPSP instances. Furthermore, we empirically analyse our proposed approach to determine which features are beneficial for performance, showing that *finding* cliques is one of the main bottlenecks and that detecting disjointness during search can lead to improved bounds on certain instances, but it generally negatively impacts learning. This work paves the way for reasoning over the disjointness of tasks inferred from a variety of standard constraints to discover novel information sourced from multiple constraints during search.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Constraint Programming, Lazy Clause Generation, Propagation, Scheduling, Cumulative, Disjunctive

Digital Object Identifier 10.4230/LIPIcs.CP.2025.35

Supplementary Material
Software (Pumpkin and processing scripts): `https://doi.org/10.5281/zenodo.15624366`
Dataset (Instances and solver logs): `https://doi.org/10.5281/zenodo.15624416`

Funding *Konstantin Sidorov*: supported by the TU Delft AI Labs program as part of the XAIT lab. *Imko Marijnissen*: supported by the NWO/OCW, as part of the Quantum Software Consortium programme (project number 024.003.037 / 3368).

[1] Both authors contributed equally to this research.

© Konstantin Sidorov, Imko Marijnissen, and Emir Demirović;
licensed under Creative Commons License CC-BY 4.0

Acknowledgements The authors would like to thank Maarten Flippo for the productive discussions on scheduling applications for constraint programming during the early phases of this project.

1 Introduction

Scheduling is one of the important classes of problems in the field of optimisation that can be loosely described as the problem of scheduling a set of tasks to satisfy certain constraints while optimising *some* objective. Constraint programming (CP) has been successful in solving scheduling problems. The key advantage of CP is its native support for constraints that capture task resource usage and efficient propagation algorithms to prune the search space, such as time-tabling [29] or energetic reasoning [2]. Another promising technique is exploiting the disjointness of tasks, as identified by Gay et al. [16] to infer bounds on tasks.

A common trait of the previously mentioned works is that they reason over a *single* resource, since reasoning jointly over multiple resources significantly increases the computational cost. However, the benefits of stronger inference obtained by aggregating and exchanging information across *multiple* resource constraints may outweigh the drawbacks, an opportunity underexplored in constraint programming for scheduling.

To illustrate the issues with single-resource reasoning, consider the following example:

▶ **Example 1.** We have two tasks on different resources and aim to minimise the latest finish time (makespan). Individually, neither resource constraint allows us to make any non-trivial inferences about the makespan. However, if we combine this with the information that the two tasks cannot be executed at the same time, then we can see that they need to be executed sequentially, which allows us to derive a tighter bound on the makespan of these two tasks.

To address this, we propose a principled solution for aggregating and exchanging information across constraints by reasoning over the disjointness of task pairs, based on both static incompatibilities (i.e., present in the original model) and dynamic incompatibilities (i.e., discovered during search). Our approach consists of three core components:

Variable creation We create Boolean variables representing whether two tasks are disjoint.

Disjointness mining We extend the existing propagators for constraints defined in the problem to infer whether two tasks should be disjoint and allow propagators to use the Boolean variables to perform additional propagation.

Conflict detection We introduce the `SelectiveDisjunctive` propagator that aggregates this disjointness information by deriving lower bounds on the earliest completion time of disjoint cliques of tasks to detect conflicts.

By doing so, we can detect inconsistent states for disjoint tasks by (a) aggregating information from the *whole* problem in contrast to the more orthodox focus on single constraints, (b) allowing propagators to use this shared information, and (c) computing conflicting task groups *during search*, thus using information obtained later in the search. As finding maximum cliques is an NP-hard problem [23], we propose a heuristic to find disjoint cliques of tasks based on minimising the growth of the interval spanned by the clique when adding a task.

To support the usefulness of our contributions, we identify an instance structure for which state-of-the-art solvers are guaranteed to incur an exponential number of conflicts unless they reformulate the problem. While state-of-the-art LCG solvers also exhibit exponential runtime in practice, our approach can solve the same instances in polynomial time.

To evaluate our approach, we implemented it in Pumpkin [12] and ran experiments on well-known scheduling benchmarks (RCPSP and RCPSP/max). Comparing our approach to baseline Pumpkin and Google OR-Tools CP-SAT [31] (a state-of-the-art CP solver that has

won many editions of the MiniZinc Challenge since 2017) shows that our approach works well when resource contention is high, with some instances exhibiting improvements of *three orders of magnitude*. For RCPSP/max, our approach discovers 16 new lower bounds (and closes six instances) and two new upper bounds; whereas for RCPSP, our approach yields four new upper bounds, four new lower bounds, and one closed instance. Our ablation study shows that the main bottleneck is finding the cliques *but* that it can be crucial to find the *right* cliques, whereas disjointness mining hampers performance due to poor learning on most instances.

To summarise, we address the insufficiency of *local* information by reasoning over the disjointness of tasks to infer *global* information. We do this by (1) introducing Boolean variables representing the disjointness between two tasks and extending existing propagators to infer the values of these variables *and* use them for additional propagation, and (2) introducing the `SelectiveDisjunctive` propagator that heuristically finds disjoint cliques to detect conflicts unattainable by current approaches due to their focus on individual constraints. We identify an instance structure for which current techniques are guaranteed to incur an exponential number of conflicts, while our approach only requires a polynomial number of them. We experimentally evaluate our approach and observe that it is most beneficial for instances with high resource contention, leading to *orders-of-magnitude* improvements for specific benchmarks. Through an ablation study, we find that dynamically determining which disjoint cliques to focus on can be beneficial and that looking for disjointness during search inhibits the learning capabilities of the solver. This work paves the way for aggregating information across constraints to derive *global* information about the problem.

The rest of the paper is structured as follows. Section 2 introduces the concepts of constraint programming relevant to our work. Section 3 reviews the previous research on disjunctive reasoning. We present and motivate our strategy for discovering and reasoning about disjunctive tasks in Section 4. We evaluate our approach on a range of scheduling problems in Section 5. Finally, we conclude and review further research directions in Section 6.

2 Preliminaries

Constraint programming framework. A Constraint Satisfaction Problem (CSP) consists of a tuple $(\mathcal{X}, \mathcal{C}, \mathcal{D})$ where $\mathcal{X}$ is the set of *variables*, $\mathcal{C}$ is the set of *constraints* which specify the relations between variables, and $\mathcal{D}$ is the *domain* which specifies for each variable which values it can take; we refer to the lowest and highest domain values as *lower* and *upper* bounds and denote them $\text{LB}(x)$ and $\text{UB}(x)$, respectively [32]. A *solution* $\mathcal{I}$ is a mapping that maps each variable in $\mathcal{X}$ to a single value in the domain of that variable in $\mathcal{D}$ which satisfies all of the constraints in $\mathcal{C}$. An *atomic constraint* (i.e. *atomic predicate*) is a predicate over a single integer variable $x \in \mathcal{X}$ and value v signified by $[\![x \otimes v]\!]$ where $\otimes \in \{=, \leq, \geq, \neq\}$.

Constraint programming (CP) is a paradigm for solving CSPs; CP solvers enforce constraints through *propagators*, each represented with a function $f : \mathcal{D} \mapsto \mathcal{D}'$ (where $\mathcal{D}' \subseteq \mathcal{D}$) which removes values from $\mathcal{D}$ infeasible under the constraints in $\mathcal{C}$. After applying the propagators, the solver makes a *decision* which creates several subproblems by splitting the domain of a variable into two or more parts. This process of applying propagators and making decisions is performed until either a solution $\mathcal{I}$ is found, the problem is found to be unsatisfiable, or a termination criterion is met.

During the search process, CP solvers can use a technique called *nogood learning* [10]. A nogood is a partial assignment that cannot be extended to a full solution, typically rendered with an implication of the form $N = p_1 \wedge \cdots \wedge p_k \implies \bot$ with p_j being the atomic constraints

encoding the partial assignment; a nogood is *unsatisfied* if no p_j is falsified. The purpose of adding nogoods is to prevent the search process from (re-)exploring parts of the search space. In this work, we use the lazy clause generation framework [14] to derive these nogoods. In this framework, it is required that the propagators explain their inferences in terms of atomic constraints, that is, they produce *explanations* of the form $e_1 \wedge \cdots \wedge e_m \implies q$.

`Cumulative` constraint. `Cumulative` is a constraint useful for many scheduling problems to model limited renewable resources, such as available work-hours. However, determining the satisfiability of a single `Cumulative` constraint is NP-complete [2], meaning that even reasoning efficiently over a single `Cumulative` requires reasoning over relaxations of the problem.

We define the `Cumulative` in Definition 2, the variable s_i encodes the start time of task i, and $(s_i + d_i)$ the finish time of i. Thus, in this constraint, we implicitly associate each task $i \in T$ with an interval $[s_i, s_i + d_i)$. We also introduce notation for the bounds of tasks and sets of tasks.

▶ **Definition 2.** *Let T be a set of **tasks**, and let a task $i \in T$ be defined by its start variable s_i, resource usage $r_i \in \mathbb{Z}_{\geq 0}$, and duration $d_i \in \mathbb{Z}_{\geq 0}$. Finally, let $C \in \mathbb{Z}_{\geq 0}$ be the **capacity** of the resource. Then the `Cumulative` constraint is the condition that at any time point τ the cumulative resource usage of intervals $[s_i, s_i + d_i)$ covering τ does not exceed the capacity:*

$$\forall \tau \in \mathbb{Z} \ : \sum_{i \in T \ : \ s_i \leq \tau < s_i + d_i} r_i \leq C. \tag{1}$$

*Given a task $i \in T$, we denote its: **earliest start time** $\mathrm{EST}_i = \mathrm{LB}(s_i)$, **earliest completion time** $\mathrm{ECT}_i = \mathrm{LB}(s_i) + d_i$, **latest start time** $\mathrm{LST}_i = \mathrm{UB}(s_i)$, **latest completion time** $\mathrm{LCT}_i = \mathrm{UB}(s_i) + d_i$, and **energy** $e_i = r_i \times d_i$.*

*Given a **set** of tasks $\Omega \subseteq T$, we denote its: **total duration** $d_\Omega = \sum_{i \in \Omega} d_i$, **earliest start time** $\mathrm{EST}_\Omega = \min_{i \in \Omega} \mathrm{LB}(s_i)$, **latest completion time** $\mathrm{LCT}_\Omega = \max_{i \in \Omega} \mathrm{UB}(s_i) + d_i$, **earliest completion time** $\mathrm{ECT}_\Omega = \max_{\Omega' \subseteq \Omega} \mathrm{EST}_{\Omega'} + d_{\Omega'}$, and **total energy** $e_\Omega = \sum_{i \in \Omega} e_i$*

An important concept for the inference over `Cumulative` constraints is the *mandatory part* (Figure 1a), which is a time interval that is covered by a task regardless of its placement:

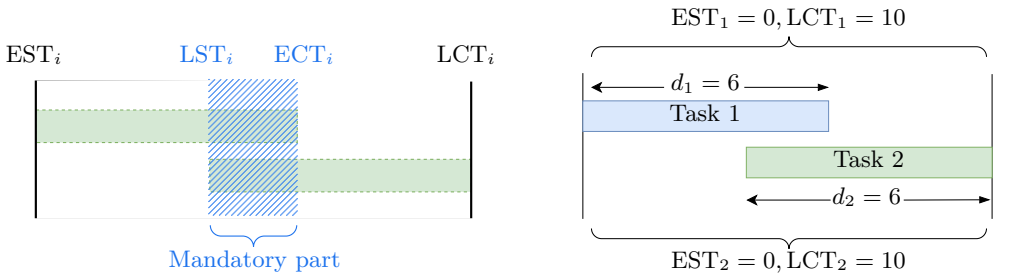
▶ **Definition 3.** *Given a `Cumulative` constraint over tasks T with capacity C, we denote the **mandatory part** MP_i of a task $i \in T$ as the interval $\mathrm{MP}_i := [\mathrm{LST}_i, \mathrm{ECT}_i)$. We define a **height** at a time point τ as total consumption of mandatory parts covering τ, that is, $\mathrm{Height}(\tau) := \sum_{i \in T \ : \ \tau \in \mathrm{MP}_i} r_i$. We also define a **reduced height** at a time point τ without $\Omega \subseteq T$ as $\mathrm{Height}^-(\tau, \ \Omega) := \sum_{i \in T \setminus \Omega \ : \ \tau \in \mathrm{MP}_i} r_i$. Finally, we define a **profile** Pr_x as a rectangle (a_x, b_x, h_x) where $\forall \tau \in [a_x, b_x] \ : \ \mathrm{Height}(\tau) = h_x$ and we assume that all profiles are maximal (i.e. there are no adjacent profiles with the same height).*

`Disjunctive` (`No-Overlap`) constraint. In this paper, we focus on a special case of `Cumulative` where the resource has unit capacity $C = 1$ and each task has unit resource usage $r_i = 1$, known as `Disjunctive`. Many inference procedures that are intractable for `Cumulative` constraints can be efficiently executed for `Disjunctive` constraints. The core inference that we use is infeasibility checking (Theorem 4), as illustrated by Figure 1b.

▶ **Theorem 4** (Overload checking). *Let $\Omega \subseteq T$ be a set of variables bound by a `Disjunctive` constraint such that $d_\Omega > \mathrm{LCT}_\Omega - \mathrm{EST}_\Omega$. Then the `Disjunctive` constraint has no feasible solutions [37], and any feasible solution of the CSP satisfies*

$$\left(\bigwedge_{i \in \Omega} [\![s_i \geq \mathrm{EST}_\Omega - \delta^-]\!] \right) \wedge \left(\bigwedge_{i \in \Omega} [\![s_i \leq \mathrm{LCT}_\Omega - d_\omega + \delta^+]\!] \right) \implies \bot \tag{2}$$

for arbitrary $\delta^-, \delta^+ \in \mathbb{Z}_{\geq 0}$ *satisfying* $\mathrm{LCT}_\Omega - \mathrm{EST}_\Omega + \delta^- + \delta^+ < d_\Omega$ *[38].*



(a) Mandatory part illustration for a *single* task i.

(b) Overload checking leads to a conflict since $d_1 + d_2 > \mathrm{LCT}_{\{1,2\}} - \mathrm{EST}_{\{1,2\}}$.

Figure 1 Key concepts related to `Cumulative` and `Disjunctive` constraints.

3 Related Work

Much work focused on inferring information based on the `Cumulative` efficiently [29, 34]. One of the approaches of primary interest in this work is edge-finding [29] as it is similar in its conflict detection procedure (also known as *input/output consistency tests* [13, Chapter 4]). A common trait of these approaches [22, 18] is that the earliest completion time of $\Omega \subseteq T$ is used to make inferences. For example, overload checking enforces the condition $\mathrm{ECT}_\Omega > \mathrm{LCT}_\Omega \implies \bot$, while edge finding extends this rule to $\mathrm{ECT}_{\Omega\cup\{i\}} > \mathrm{LCT}_\Omega \implies \Omega \ll i$, updating the bounds of t such that it does not overflow the interval bounded by the tasks in Ω. While edge-finding takes into account the earliest start times and latest finish times of the tasks (and horizontally elastic edge-finding [18] takes into account the maximum resource usage of the tasks), this type of reasoning does not take into account the disjointness of tasks and its influence on the earliest completion time of a set of tasks. Furthermore, edge-finding reasons over a single `Cumulative`, whereas we propose gathering information across several `Cumulative` constraints to infer the earliest completion time.

Detecting disjoint pairs of tasks within a `Cumulative` constraints is also a viable reasoning strategy [16], known as time-table disjunctive reasoning. This strategy can detect disjoint task pairs dynamically by taking into account the time points at which other tasks are guaranteed to consume *some* amount of resource. When it detects such disjointness using time-tabling [29], it determines when the bounds of one of the tasks can be updated. The main difference with our approach is that time-table disjunctive reasoning only reasons about a single `Cumulative` constraint while we reason jointly over multiple constraints, and our approach infers disjointness from other constraints besides the `Cumulative`.

To the best of our knowledge, the only work that addresses the reasoning over multiple `Cumulative` constraints is the work by Beldiceanu and Carlsson [5]. This work introduces a multi-resource `Cumulatives` constraint as a generalisation of the `Cumulative`, admitting negative resource consumption and lower/upper bounds on the cumulative resource usage. However, their reasoning makes limited use of the additional information provided by reasoning over multiple `Cumulative` constraints, nor does it take into account disjointness.

Besides reasoning about `Cumulative` constraints, we consider other works that reason about disjunctive tasks. One area is branch-and-bound algorithms for scheduling problems. A common example is the *resource-constrained scheduling problem* (RCPSP), in which one of the lower bounds (LB4) on makespan is the sum of durations in a group of pairwise

disjunctive tasks; with similar bounds being proposed for activity groups with at most two or at most k of them running in parallel [24]. Thus, our approach generalises that strategy while also working in the constraint programming context, where the tasks can be (a) arbitrarily bounded by the solver and (b) involved in other constraints. This type of reasoning can also be embedded into the search tree node representation, for example, as a scheduling scheme [7], which accounts for the pairs of tasks that are either disjunctive or running in parallel.

We thus note that there are several methods that attempt to find relations between variables and to exploit these relations to infer information about the bounds. However, the relations discovered by the majority of state-of-the-art techniques are local to individual resource constraints or do not reason over the disjointness of tasks; contrary to that, our work aims to discover this disjointness *dynamically during search* from multiple constraints and use information *across* propagators to infer the earliest completion time of a set of tasks.

4 Our Contribution: Finding Disjunctive Cliques

This section will introduce our approach for aggregating information across constraints. We propose the following workflow:

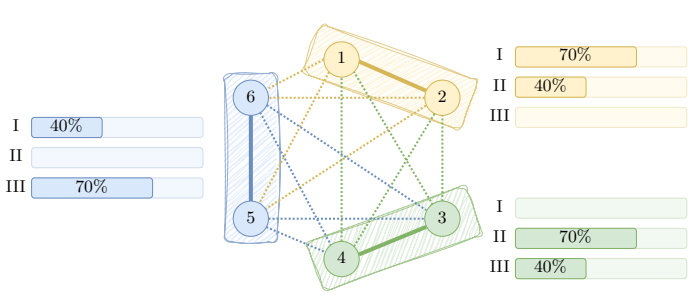
1. Given a set of tasks T (bound by `Cumulative` constraints), introduce new variables δ_{ij} for all $i, j \in T$ and reify them as $\delta_{ij} \rightarrow [s_i, s_i + d_i) \cap [s_j, s_j + d_j) = \emptyset$. (Subsection 4.2)
2. Introduce the reification variables δ into other constraints (such as the `Cumulative` and difference logic) and extend their propagation logic to derive implications of the form $[\![\text{variable bounds}]\!] \implies \delta_{ij}$, and propagate variable bounds based on δ. (Subsection 4.4)
3. Introduce our novel `SelectiveDisjunctive` propagator which derives lower bounds on the earliest competion time of sets of pairwise disjoint tasks given (a) pairs of tasks $\{i, j\}$ such that $\delta_{ij} = 1$, and (b) the bounds on start times to discover conflicts. (Subsection 4.3)

Our approach allows us to gain a *global* view across multiple constraints rather than relying on single constraints to independently make local inferences which causes limited information exchange between propagators. Thus, our approach allows them to communicate more elaborate facts between propagators and a CP solver, such as the disjointness of a pair of tasks, as opposed to existing methods focusing on a *local* view.

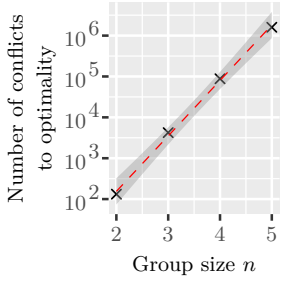
4.1 Motivating Example: $3n$ Problem

Let us first look at a case where reasoning over a single resource is insufficient: Suppose that we are scheduling six tasks, each with unit duration. At every time unit, we are given a fixed amount of resources of types I, II, and III. The amount of resources consumed by each task is illustrated in Figure 2a. *How much time has to pass before all six tasks are completed?*

This is possible to do in six units of time by sequentially scheduling the tasks in any order. What cannot be inferred by reasoning over single constraints (such as time-tabling or energetic reasoning) is that this schedule is the fastest possible (in terms of the latest completion time). To see why, observe that the two yellow tasks cannot be run in parallel due to the shortage of resource I, and the same is true for both the two green tasks and the two blue tasks by similar reasoning. On the other hand, scheduling a yellow and a green task in parallel is not possible, as it would overflow the resource II; the same holds for a green and a blue task, as well as for a blue and a yellow task. By this point, we have considered *all* pairs of tasks and concluded for each of them that they cannot be run in parallel, making it impossible to improve upon the sequential schedule.



(a) Conflict graph of the motivating example. Edge colour indicates resource, solid edges connect tasks jointly consuming 140%, and dotted edges connect tasks jointly consuming 110%. All tasks of the same colour have the same resource usage.



(b) Number of conflicts generated by CP-SAT solving $3n$ problem instances with $d = 1$.

Figure 2 Motivating example: scheduling problem with tasks that are pairwise disjoint due to *various* resource constraints and how CP-SAT performs on these instances.

The previous reasoning relies only on the partition of tasks into three groups such that choosing one task from every two groups implies disjointness; this property holds if we consider groups of size n. We generalise the reasoning above not only in terms of group size n but also by duration d and capacity parameters p, q, M:

▶ **Definition 5.** *Given a group size $n \in \mathbb{N}_{\geq 1}$, duration $d \in \mathbb{N}_{\geq 1}$, and parameters $p, q, M \in \mathbb{N}_{\geq 1}$ with $2p < M < p + q < 2q$, the* ***3n problem instance*** *is the constraint satisfaction problem defined over variables X, Y, Z with $|X| = |Y| = |Z| = n$ and domains $\mathcal{D}(\cdot) \equiv [1, (3n-1)d]$:*

$$\texttt{cumulative}(X +\!\!+ Y; [d]^{2n}; [p]^n +\!\!+ [q]^n; M) \quad \texttt{cumulative}(Y +\!\!+ Z; [d]^{2n}; [p]^n +\!\!+ [q]^n; M)$$
$$\texttt{cumulative}(Z +\!\!+ X; [d]^{2n}; [p]^n +\!\!+ [q]^n; M)$$

As we established, any $3n$ instance is unsatisfiable, but to establish this without search, a solver needs information about all resources. However, each resource individually only implies that any solution occupies some time segment of duration $2n \times d$, and the remaining $(n-1)d$ time units have to be resolved by search. This suggests that any valid sequence of inferences that operates on `Cumulative` constraints separately has exponential length in terms of n.

We prove this, and the $3n$ problem thus serves a role similar to pigeonhole principle formulas [19], but for the `Cumulative` models instead of propositional formulas. We show that any unsatisfiability proof of a $3n$ problem in the C-RES proof system [21] has length exponential in n, which shows that any LCG solver requires an exponential number of steps to prove unsatisfiability, as long as it introduces neither new constraints nor new variables:

▶ **Theorem 6** ($3n$ problem intractability)**.** *Let $\mathcal{P}$ be a C-RES proof of a CSP instance with* ***all*** *clauses valid for any single constraint in Definition 5 for $d = 1$. Then $|\mathcal{P}| = \Omega(1.05^n)$.*

Proof Sketch. First, we show that $\mathcal{P}$ can be encoded, with extra assumptions, as a pigeonhole problem proof over $3n$ variables (Appendix A). Next, we adapt the argument by Beame and Pitassi [4] to verify the exponential lower bound (Appendix B). ◀

We evaluate this insight empirically by running $3n$ instances for $n \leq 10$ and $d \leq 5$ with five solvers: two LCG solvers (Google OR-Tools CP-SAT [31] and Chuffed [9]), a non-learning CP solver (Gecode [17]), and two branch-and-cut solvers (CBC [15] and HiGHS [20]). We

ran each solver with a 12-hour time limit, 1 CPU, and 4000 MB of RAM. Both the LCG and non-learning solvers exhibit exponential behaviour, as shown on Figure 2b for CP-SAT, and only one of the runs $n \geq 6$ proved unsatisfiability. These results suggest that LCG solvers indeed resort to exhaustive enumeration on the $3n$ problem. Branch-and-cut solvers have exhibited better performance in solving those instances, demonstrating another similarity with pigeonhole principle formulas. HiGHS solved all instances with $d \leq 3$ in seven minutes. CBC has performed worse, as it has failed to solve instances with $n \geq 7$, however, it has successfully solved $d = 1$ for $n \leq 6$ in less than two seconds.

4.2 `SelectiveDisjunctive` Constraint

Now that we have motivated the necessity for using *global* information, we describe how we use information concerning the disjointness of tasks across *multiple* constraints by introducing the `SelectiveDisjunctive` constraint. The `SelectiveDisjunctive` constraint (i) generalises the disjunctive constraint in a way that allows other propagators to indicate disjoint task pairs, and (ii) restricts the variables indicating this disjointness to be true when a pair of intervals in the assignment is disjoint. We formalise this intuition in Definition 7.

▶ **Definition 7.** *Let T be a set of tasks, and Δ be the $n \times n$ matrix of Boolean variables indexed by T. Then a `SelectiveDisjunctive`(T, Δ) constraint is a condition that for any two tasks $i, j \in T, i \neq j$ either $\delta_{ij} = 0$ or $[s_i, s_i + d_i) \cap [s_j, s_j + d_j) = \emptyset$.*

Introducing these variables and `SelectiveDisjunctive` does not prune any valid solutions, but it allows performing two new operations: (a) fixing the variables from Δ and (b) using these Δ variables in other propagations. This allows us to encode facts – such as 'a pair of tasks cannot overlap given a partial assignment of other tasks' – that are not possible to encode concisely using atomic constraints over the original model variables. More specifically, we can now observe that Definition 7 embeds the standard `Disjunctive` constraint (or even exponentially many of them); to formalise this, we first need the following notion:

▶ **Definition 8.** *Given a search state with domains $\mathcal{D}'$ and a `SelectiveDisjunctive` constraint, a **conflict graph** $\mathcal{G}|_{\mathcal{D}'}$ is a graph with a vertex for each task $i \in T$ and an edge for tasks $\{i, j\}$ such that $\mathcal{D}'(\delta_{ij}) = \{1\}$ (ommitting $\mathcal{D}$ when it is clear from context).*

We can make two key observations based on Definition 8: (1) a conflict graph can be seen as an *exponentially large* collection of `Disjunctive` constraints, and (2) this is also a *dynamically updating* collection of `Disjunctive` constraints, provided that the other parts of a constraint solving engine can propagate the domains of reification variables. On top of this, it suggests that any reasoning for the conventional `Disjunctive` constraint can be deployed for any clique of the conflict graph. In Subsection 4.3, we discuss how to find these cliques in the conflict graph and how to apply the overload checking rule for the `Disjunctive` in this context.

4.3 Overload checking for `SelectiveDisjunctive`

To clarify the intuition behind our inference strategy, consider the following example:

▶ **Example 9.** Let i, j, k be tasks with possible start times $s_i \in [0, 2], s_j \in [0, 2], s_k \in [2, 3]$ and durations $d_i = d_j = 2, d_k = 3$ bound by a `SelectiveDisjunctive` constraint such that $\delta_{ij} = \delta_{jk} = \delta_{ik} = 1$ in the current search state. We can observe that any feasible assignment also satisfies `Disjunctive`$([s_i, s_j, s_k], [d_i, d_j, d_k])$. However, this constraint is infeasible by overload checking for $\{s_i, s_j, s_k\}$: these three tasks jointly cover seven time units, but their EST is 0 and LCT is 6, leaving only six time units among these three tasks.

▶ Note 10. The example above purposefully does not specify *why* the Δ-variables are true: the same reasoning can be carried out as soon as those intervals are established to be disjoint. We discuss the specific strategies for deriving those facts in Subsection 4.4.

We formalise this intuition in Proposition 11 by reformulating Theorem 4 in the context of the `SelectiveDisjunctive` constraint.

▶ **Proposition 11** (Selective overload checking). *Let T be the set of tasks bound by a* `SelectiveDisjunctive` *constraint, and suppose that $\Omega \subseteq T$ is a clique in the conflict graph $\mathcal{G}$ induced by the current variable domains. Then this constraint is infeasible if $d_\Omega > \mathrm{LCT}_\Omega - \mathrm{EST}_\Omega$, and any feasible solution of the CSP satisfies*

$$\left(\bigwedge_{i,j\in\Omega,\ i\neq j} \delta_{ij}\right) \wedge \left(\bigwedge_{\omega\in\Omega} [\![s_\omega \geq \mathrm{EST}_\Omega - \delta^-]\!]\right) \wedge \left(\bigwedge_{\omega\in\Omega} [\![s_\omega \leq \mathrm{LCT}_\Omega - d_\omega + \delta^+]\!]\right) \implies \bot \quad (3)$$

for arbitrary $\delta^-, \delta^+ \in \mathbb{Z}_{\geq 0}$ satisfying $\mathrm{LCT}_\Omega - \mathrm{EST}_\Omega + \delta^- + \delta^+ < d_\Omega$.

Unlike the `Disjunctive` overload checking, propagation of the `SelectiveDisjunctive` requires an additional step before overload checking can be performed: finding cliques. Finding *a* clique leading to a conflict, even if it exists, is NP-complete[2] and we thus propose a heuristic in Algorithm 1 that dynamically explores some of the cliques present in the conflict graph to balance the trade-off between finding " good" cliques and the time spent finding them. The heuristic algorithm considers each node as a root for a clique $\mathcal{C}$ and then adds tasks $j \in T \setminus \mathcal{C}$ which minimise $\mathrm{LCT}_{\mathcal{C}\cup\{j\}} - \mathrm{EST}_{\mathcal{C}\cup\{j\}}$ (breaking ties in favour of tasks with a longer duration) while retaining the clique property. The time complexity of Algorithm 1 is $\mathcal{O}(|T|^4)$.

4.4 Disjointness mining

We described how to deploy the `SelectiveDisjunctive` propagator when some of the reification variables have been assigned, but we have not yet described how to infer the domain of these reification variables. In this section, we describe strategies for *disjointness mining*, that is, discovering, possibly during the search, pairs of tasks $i, j \in T$ that are inferred to be disjoint.

Domain disjointness. One direct source of disjointness is the current bounds of variables. The rule can be formally stated as $\forall i, j \in T : [\mathrm{EST}_i, \mathrm{LCT}_i) \cap [\mathrm{EST}_j, \mathrm{LCT}_j) = \emptyset \implies [\![\delta_{ij} = 1]\!]$. We perform this check during the selective overload checking (Subsection 4.3). Given a propagation of δ_{ij} via this detection, we explain the propagation according to Equation 4.

$$\begin{cases} [\![s_i \leq \mathrm{EST}_j - d_i]\!] \wedge [\![s_j \geq \mathrm{EST}_j]\!] & \text{if } \mathrm{LCT}_i \leq \mathrm{EST}_j \\ [\![s_j \leq \mathrm{EST}_i - d_j]\!] \wedge [\![s_i \geq \mathrm{EST}_i]\!] & \text{otherwise} \end{cases} \implies [\![\delta_{ij} = 1]\!] \quad (4)$$

Difference logic. Scheduling problems often include tasks dependent on the execution of other tasks. These dependencies can be provided in the form of precedence constraints, which provide static information about the disjointness. For example, to encode that j starts δ time units *after* i, we can use the difference logic constraint $s_i + \delta \leq s_j$.

[2] This can be shown by reduction from the maximum clique problem

Algorithm 1 Heuristic conflict discovery procedure.

Data: A set of tasks T with intervals $[\mathrm{EST}_i, \mathrm{LCT}_i)$ and durations d_i for all $i \in T$.
Data: A conflict graph $\mathcal{G} = (T, E)$.
Result: A set $\Omega \subseteq T$ satisfying $d_\Omega > \mathrm{LCT}_\Omega - \mathrm{EST}_\Omega$ or $\bot$ if none was found.
if $\exists e \in E : d_e > \mathrm{LCT}_e - \mathrm{EST}_e$ **then**
  return e ; /* Check for binary conflicts */
for $t_0 \in T$ **do**
  $\Omega \leftarrow \{t_0\}$; /* Ω is a clique containing t_0 */
  $S, T \leftarrow \mathrm{EST}_{t_0}, \mathrm{LCT}_{t_0}$; /* $[S, T)$ is an interval covered by Ω */
  while $d_\Omega \leq \mathrm{LCT}_\Omega - \mathrm{EST}_\Omega$ **do**
    // Collect tasks that are disjoint with Ω
    $\Omega^+ \leftarrow \{j \in T \setminus \Omega : [S, T) \cap [\mathrm{EST}_j, \mathrm{LCT}_j) \neq \emptyset \wedge \forall i \in \Omega, \{i, j\} \in E\}$;
    if $\Omega^+ = \emptyset$ **then**
      break ; /* Cannot extend Ω into a clique, try another root */
    $t^+ \leftarrow$ an element of $\arg\min_{j \in \Omega^+} \Big(\max(T, \mathrm{LCT}_j) - \min(S, \mathrm{EST}_j), -d_j\Big)$;
    $\Omega \leftarrow \Omega \cup \{t^+\},\ S \leftarrow \min(S, \mathrm{EST}^+_t),\ T \leftarrow \max(T, \mathrm{LCT}^+_t)$; /* Extend Ω */
  if $d_\Omega > \mathrm{LCT}_\Omega - \mathrm{EST}_\Omega$ **then**
    return Ω;
return $\bot$;

These constraints encode *stronger* conditions than disjointness; for example, precedence can be seen as disjointness with ordering. Thus, we can deduce the disjointness variables from the difference constraint for appropriate parameter values (this occurs at the root level and requires no explanation). We formalise this in the following proposition:

▶ **Proposition 12.** *A pair of intervals $[s_i, s_i + d_i), [s_j, s_j + d_j)$ is disjoint (i.e., $\delta_{ij} = 1$ in any feasible solution) if any of the following constraints are implied by the CSP: (a) $s_j - s_i \geq \delta$ for some constant $\delta \geq d_i$, or (b) $s_i - s_j \geq \delta$ for some constant $\delta \geq d_j$.*

`Cumulative`. There are several `Cumulative` propagation techniques that reason about when the combination of scheduling a task i and a set of tasks $\Omega \subseteq T \setminus \{i\}$ leads to resource overflows to infer relations between tasks. In this work, we use resource profiles to determine when a pair of tasks is disjoint. An example of this reasoning can be seen in Example 13.

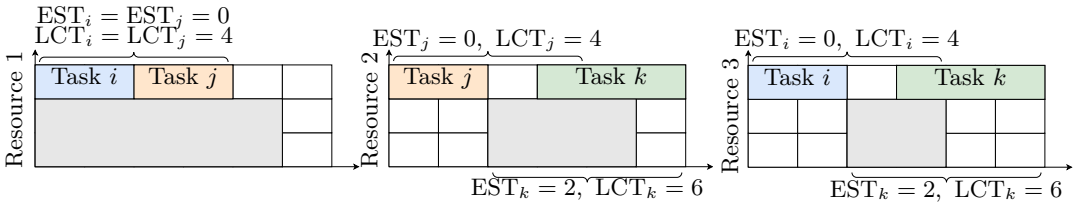


Figure 3 Example where profiles (marked in grey) cause disjointness between all tasks i, j, k.

▶ **Example 13.** In addition to the assumptions of Example 9, suppose that the problem has three `Cumulative` constraints as visualised in Figure 3. We can derive from Resource 1 that i and j are disjoint ($\delta_{ij} = 1$); if they are not, then they overlap at some point $\tau^* \in [\max(\mathrm{EST}_i, \mathrm{EST}_j) = 0, \min(\mathrm{LCT}_i, \mathrm{LCT}_j) = 4]$. However, this would cause an overflow of the capacity, implying disjointness. Similar reasoning derives that $\delta_{jk} = 1$ and $\delta_{ik} = 1$.

In this example, just as in the motivating example (Subsection 4.1), neither of the three constraints is infeasible; in fact, each can only imply one disjointness relation. However, the *intersection* of those constraints is correctly declared infeasible by (i) `Cumulative` mining and (ii) `SelectiveDisjunctive` overload checking. Thus, combining the mining strategies with `SelectiveDisjunctive` inferences can generate conflicts that are out of reach for the conventional techniques for the `Cumulative` constraint. ⌟

We adapt the time-table disjunctive reasoning [16] by going over all pairs of tasks $i, j \in T$, and calculating their overlap o_{ij}. Then we check $\forall \tau \in o_{ij}$ if there exists a profile such that scheduling the tasks together with that profile would lead to an overflow. A simple implementation of this approach has time complexity $\mathcal{O}(|T|^2|TT|)$, where TT is the set of existing profiles (i.e, the time-table). This reasoning is formally stated in the following proposition:

▶ **Proposition 14.** *Consider a* `Cumulative` *constraint on tasks T with capacity C. Let the overlap between two tasks $i, j \in T$ be given by $o_{ij} = [\max(\text{EST}_i, \text{EST}_j), \min(\text{LCT}_i, \text{LCT}_j))$. Then δ_{ij} is true for any feasible solution as long as there is no point $\tau \in o_{ij}$ such that $\text{Height}^-(\tau, \{i, j\})$ plus the resource consumption of i and j fits within the capacity:*

$$\forall \tau \in o_{ij} \ : \ r_i + r_j + \text{Height}^-(\tau, \{i, j\}) > C \implies [\![\delta_{ij} = 1]\!]$$

Moreover, given a propagation of δ_{ij} by a set of profiles $\mathcal{P} = \{(a_0, b_0, h_0), ..., (a_m, b_m, h_m)\}$, $a_0 < \cdots < a_m$, we explain the propagation according to Equation 5, assuming without loss of generality that $\text{EST}_i \leq \text{EST}_j$. For $\mathcal{E}_{pr}$, we use the big-step explanation [33].

$$\begin{cases} [\![s_j \geq a_0]\!] \wedge [\![s_j \leq b_m - d_j + 1]\!] & \text{if } \text{LCT}_i \geq \text{LCT}_j \\ [\![s_i \leq b_m - d_i + 1]\!] \wedge [\![s_j \geq a_0]\!] & \text{otherwise} \end{cases} \wedge \left(\bigwedge_{Pr \in \mathcal{P}} \mathcal{E}_{pr}(Pr) \right) \implies [\![\delta_{ij} = 1]\!] \quad (5)$$

The first part of the explanation in Equation 5 is based on the observation that either one task subsumes the entire interval of the other (in which case the explanation depends only on the bounds of s_j), or there is a partial overlap where one task starts at the same time or before the other and ends at the same time or before the other (in which case the overlapping part is only defined by the upper bound on s_i and the lower bound on s_j).

Additionally, we adapt the propagation by Gay et al. [16] and propose a rule using δ in Equation 6 (the rule for updating s_i is symmetric). The rule in Equation 6 and the rule proposed by Gay et al. [16] do not subsume one another since Equation 6 can propagate based on information gained from other constraints, while the rule by Gay et al. can propagate when two tasks are not *fully* disjoint.

$$[\![\delta_{ij} = 1]\!] \wedge \text{ECT}_i > \text{EST}_j \wedge \text{LST}_i < \text{ECT}_j \implies [\![s_j \geq \text{ECT}_i]\!] \quad (6)$$

Nogood. Last, we describe how to derive disjointness from nogoods: given an unsatisfied nogood N, tasks $i, j \in T$ are disjoint if falsifying each of the unassigned atomic constraints implies δ_{ij}. An example can be seen in Example 15 and it is formalised in Proposition 16.

▶ **Example 15.** Given two tasks $i, j \in T$ such that $s_i \in [0, 9], s_j \in [7, 20]$ with durations $d_i = 2, d_j = 5$, consider a nogood $N = \{p_i = [\![s_i \geq 5]\!], p_j = [\![s_j \leq 11]\!], \ ...\}$. Suppose that in the current search state, only the first two atomic constraints are unassigned, while the rest are satisfied. Thus, in any feasible solution either $\neg p_i = [\![s_i \leq 4]\!]$ or $\neg p_j = [\![s_j \geq 12]\!]$ holds. But in either case, we can derive δ_{ij} via domain disjointness:

- If $\neg p_i$ is true, then $\text{LCT}_i = 6 \leq \text{EST}_j = 7$, implying δ_{ij}.
- If $\neg p_j$ is true, then $\text{LCT}_j = 11 \leq \text{EST}_i = 12$, also implying δ_{ij}.

▶ **Proposition 16.** *Let $\mathcal{D}'$ be a set of domains, and suppose that a nogood N has no falsified atomic constraints in this domain. Then a pair of tasks i, j is disjoint if for every atomic constraint $p \in N$ **not** satisfied with respect to $\mathcal{D}'$, δ_{ij} is implied by domain disjointness.*

Our procedure is a simplified version of Proposition 16 based on the two-watcher scheme [28, 27]. Whenever a watcher is updated, we perform a scan to determine whether the conditions of Proposition 16 hold while reasoning over domain disjointness; this misses out on some propagations, but the time required is significantly reduced. The time complexity of this approach is $\mathcal{O}(|N|)$ for each nogood N. We explain this propagation due to an unsatisfied nogood N as $\mathcal{E}_{sn}(s_i, s_j, p_i, p_j) \wedge \left(\bigwedge_{p^+ \in N^+} p^+\right) \implies [\![\delta_{ij} = 1]\!]$, where $N^+ \subseteq N$ is the set of satisfied atomic constraints, and $p_x = [\![s_x \otimes v]\!]$ is one of the two undecided atomic constraints. See Appendix C for the definition of $\mathcal{E}_{sn}$.

5 Experiments

We implemented our approach as part of Pumpkin [12]. First, we present an ablation study in Subsection 5.1, which shows that (a) dynamic disjointness mining hinders the learning of the solver but that it can be beneficial on specific instances, (b) an alternative efficient sorting heuristic has a significant impact on the search but also leading to massive slowdowns on some instances, and (c) enabling the propagation in Equation 6 does not consistently change the performance. Next, we run a comparison on RCPSP and RCPSP/max benchmarks in Subsection 5.2 with baseline Pumpkin and CP-SAT, which shows that our approach can accelerate search by orders of magnitude on certain instances. Additionally, we compare our results with bounds reported in the literature and show that our approach discovers new lower bounds on 16 RCPSP/max instances and four RCPSP instances, as well as new upper bounds on two RCPSP/max instances and four RCPSP instances.

Experiment Setup. The implementation of our approach with the infrastructure for running the experiments is available in the supplementary material. We ran our experiments on *DelftBlue* [11], with each run of an instance being allocated a single core of an Intel Xeon E5-6248R 24C 3.0GHz processor and 4000 MB of RAM with a time limit of one hour.

For our evaluation, we use two models that minimise the latest completion time of all tasks (makespan). One is **RCPSP/max**, a collection of `Cumulative` constraints and precedence relations encoded by the constraints $s_i + \gamma_{ij} \leq s_j$ with *arbitrary* constants $\gamma_{ij} \in \mathbb{Z}$; we use the test suites distributed by PSPLIB [25], namely, C, D, UBO, and SM suites. We also use **RCPSP**, a special case of RCPSP/max where all precedence relations have the form $s_i + d_i \leq s_j$; we use the data files provided in the MiniZinc benchmarks [36], which correspond to the AT [30], BL [3], Pack [8], Pack-d and KSD15-D [26], and PSPLIB [25] suites.

Given a minimisation objective $\mathcal{O}$, each of the solvers is run with one of the two search directions. In the **primal search**, the solver generates a series of problems with an extra assumption $[\![\mathcal{O} \leq o_n]\!]$ for decreasing values of o_n. In the **dual search**, the solver generates a series of problems with an extra assumption $[\![\mathcal{O} \leq o_n]\!]$ for increasing values of o_n.

We discard all the instances solved by the Pumpkin baseline within five seconds in either search direction, leaving us with 736 RCPSP instances and 349 RCPSP/max instances. The models we used and the data files are available in the supplementary material. We evaluate the rate of progress of a solver towards the best-known bound with the following metrics. First, if $M(t)$ is the lowest makespan discovered at time $t \in [0, T]$, and M^* is the lowest discovered makespan for this problem, then the **primal integral** is $\int_0^T \frac{M(t) - M^*}{M(t)}$ and

measures how fast the solver progresses towards good solutions [6]. Conversely, if $B(t)$ is the highest lower bound discovered at time $t \in [0, T]$, and B^* is the highest discovered lower bound for this problem, the **dual integral** is defined as $\int_0^T \frac{B^* - B(t)}{B^*}$.

5.1 Ablation Study

To evaluate the influence of individual components on the performance of the whole approach, we ran the disjunctive approach, both in primal and dual search directions, with *all* sixteen combinations of enabling and disabling the following features:

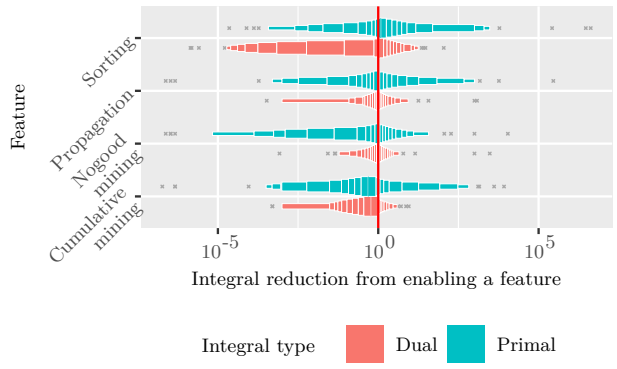
Cumulative mining Enable the mining for `Cumulative` as described in Subsection 4.4.

Nogood mining Enable the mining for nogoods as described in Subsection 4.4.

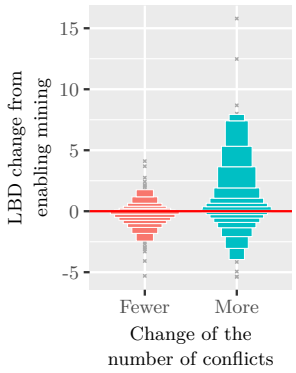
Sorting Restrict Ω^+ in Algorithm 1 to tasks j with $\mathrm{LCT}_j \leq \min_{i \in \Omega} \mathrm{LCT}_i$.

Propagation Execute the propagation shown in Equation 6 on discovered disjunctive pairs.

Figure 4a reports the relative gains from enabling each feature. For each feature, we consider *all* combinations of the remaining three features and evaluate the ratio between the aforementioned integrals when adding the feature to this configuration and *without* the feature.



(a) Distribution of ratio of integrals of enabling each evaluated feature and *not* enabling the feature. The vertical line in the middle corresponds to no observed difference, a ratio larger than 1 corresponds to an improvement by using the feature, and values smaller than 1 correspond to performance degradation.



(b) Change in LBD split by change in # conflicts with cumulative mining compared to baseline; a negative value indicates a decrease in the LBD when using mining.

Figure 4 Impact of enabling the listed features on the solver performance.

One of the major takeaways is that cumulative mining has a predominantly negative impact on search performance. As we understand, this happens because nogood learning derives statements overly specific to the current search state and thus leads to less general nogoods. To support this conclusion, we compare the average literal block distance (LBD) [1] of nogoods produced[3] with and without cumulative mining in Figure 4b, as this metric indicates learned nogood quality. In 47% of all pairs of runs, mining leads to fewer conflicts, while having a relatively small impact on LBD; however, in the other half of run pairs, mining not only increases the number of conflicts but also exhibits a much larger increase in the LBD.

[3] If two runs reached different solutions, we compare the number of conflicts to the best common bound.

On the positive side, we have discovered that cumulative mining *can* be a beneficial strategy for instances where many *triples* of tasks – but few *pairs* of them – are mutually disjoint. In that case, fixing the position of a few tasks can uncover a variety of disjoint pairs, aiding the conflict discovery. In particular, this is the case for several instances from the Pack-d collection; we ran the mining configuration of our approach on those instances and observed that the non-mining implementation discovers few conflicts (and thus makes little progress), whereas the mining implementation discovers several bounds tighter than reported by all the other approaches (see Appendix D).

Another takeaway is that the sorting heuristic has a significant influence on the search. Enabling it reduces integrals for most instances, suggesting that the bottleneck of our approach is the clique search, and most of the iterations of Algorithm 1 do not yield conflicts. However, there is a group of instances for which the *dual* integral degrades by three orders of magnitude or more; for this reason, we only use the sorting heuristic when the disjunctive approach runs in the primal search direction.

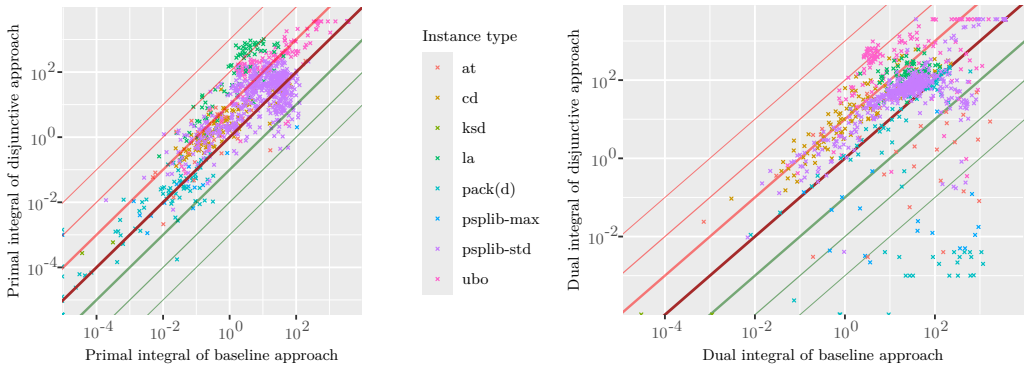
Last, enabling propagation does not exhibit a consistent change in any direction; the primal search can substantially benefit from propagation, but fails to do so consistently, and the dual search is left virtually uninfluenced by the propagation. A similar conclusion holds for using nogood mining in dual search, albeit the performance of primal search becomes consistently worse with nogood mining enabled.

As a result, we use the following configurations: enable sorting and disable propagation and mining for *primal search*, and disable all additional features for *dual search*.

5.2 Scheduling Problem Evaluation

In order to establish whether our approach is beneficial, we compare the disjunctive approach with baseline Pumpkin and CP-SAT v9.11 using the aforementioned integrals.

We start by evaluating the influence of adding disjointness reasoning to the baseline Pumpkin. As seen from Figure 5: (1) the disjunctive approach proves much more successful when using dual search as opposed to primal search, and (2) the disjunctive approach commonly translates to *much* larger improvements in cases where it does not hinder the search, while it can also degrade the performance on a number of instances.



(a) Primal integrals. **(b)** Dual integrals.

Figure 5 Comparison of integrals of baseline and disjunctive approaches. Both axes are logarithmic; lines are evenly spaced and correspond to a tenfold relative change between the integrals.

Figure 5b suggests that the disjunctive approach can accelerate the search by at least *three orders of magnitude*, which indicates that our approach can improve state-of-the-art bounds. Indeed, our approach discovers a variety of lower bounds that are both better than the bounds previously reported and the bounds discovered by all other evaluated approaches. Appendix D introduces a precise criterion for reporting the bounds and the complete list of the novel bounds.

Remarkably, our approach is able to close five of the previously open RCPSP/max J30 instances in *under a second*. For each of those instances, there is a disjoint clique Ω at the root level with d_Ω equal to the optimal bound. In all five cases, these cliques correspond to various resource constraints, making them similar to the aforementioned $3n$ instances; fittingly, CP-SAT has been able to certify only one of them after ten minutes of search, whereas the other instances time out with all other approaches. To clarify this point, we have reproduced and justified such a clique for one of those instances in Appendix E, thereby closing one of the previously open instances without any search.

Both plots in Figure 5 suggest that instance features have a major influence on the change in performance. To establish which instance properties aid our approach, we introduce a measure of similarity of a `Cumulative` constraint to a `Disjunctive` constraint.

▶ **Definition 17.** *For a* `Cumulative` *constraint over a resource with capacity C, and tasks T, we call the quantity* $\mathrm{RC} = \frac{1}{|T|}\sum_{i\in T}\frac{r_i}{C}$ *the* ***constrainedness*** *of that* `Cumulative`*. Given a CSP, we say that its* ***resource constrainedness*** RC *is the maximum constrainedness among its cumulative constraints. A larger value indicates more resource contention.*

Figure 6 demonstrates the impact of the resource constrainedness: there is exactly *one* instance with an RC under 15% which benefits from using the disjunctive approach, whereas instances with a " scarce" resource (i.e., RC $\geq$ 40%) account for the largest improvements and have the most variability. The performance on the instances between those two values mostly degrades, with a small number of instances seeing minor improvements.

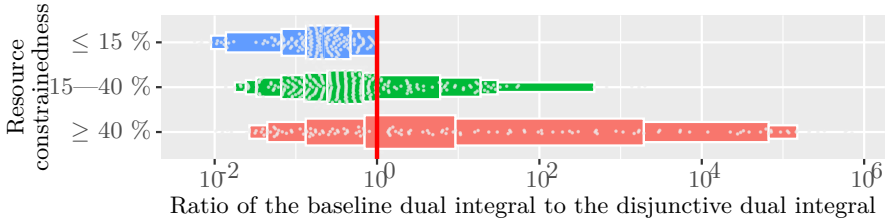
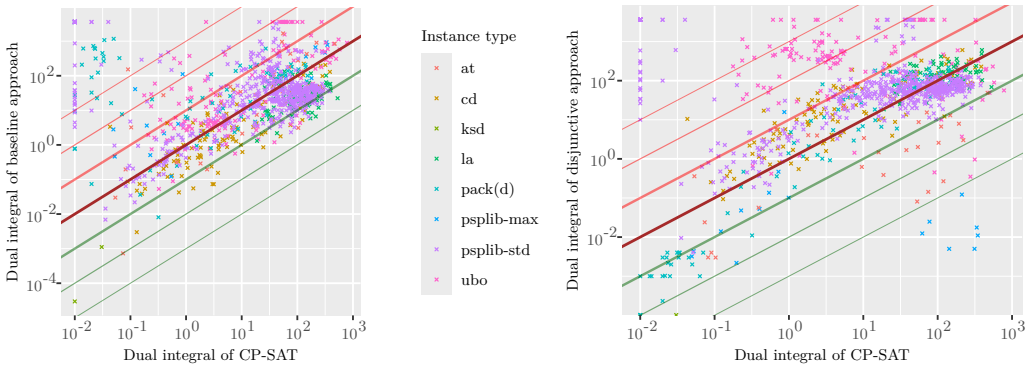


Figure 6 Distribution of dual integral ratio of baseline and disjunctive approach for various values of resource constrainedness. Points to the right of the vertical line correspond to improvements by the disjunctive approach, and points to the left correspond to deterioration.

Last, we evaluate our approach against CP-SAT. Figure 7a shows that the baseline consistently performs better on PSPLIB, with worse performance on pack(d) instances. On the other hand, Figure 7b indicates that the disjunctive approach has a much smaller edge against CP-SAT, yet it can improve certain instances by many orders of magnitude.

6 Conclusions

We have presented a novel way to aggregate information about the disjointness of tasks in scheduling problems across multiple constraints. We do this by creating new variables representing the disjointness between pairs of tasks, mining for this disjointness during search,



(a) Baseline. **(b)** Disjunctive.

Figure 7 Comparison of Pumpkin and CP-SAT dual integrals. All axes are logarithmic; lines on the plots are evenly spaced and correspond to a tenfold relative change.

and heuristically finding disjoint cliques that lead to conflicts. We show that the "standard" approaches require exponentially many conflicts to prove optimality on crafted instances, while our approach can show their infeasibility instantaneously. We also show that our approach provides improvements in the order of magnitudes for benchmark instances with high resource contention, indicating that reasoning across multiple constraints *during search* is of vital importance for inferring the *global* structure of instances.

One direction for future work would be determining what is a *good* clique. Another direction would be to look into incorporating more `Disjunctive` reasoning over the cliques. Additionally, future work could focus on mining *jointness* rather than *dis*jointness.

References

1 Gilles Audemard and Laurent Simon. Refining restarts strategies for SAT and UNSAT. In *Proceedings of the 18th international conference on principles and practice of constraint programming*, pages 118–126, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-33558-7_11.

2 Philippe Baptiste, Claude Le Pape, and Wim Nuijten. Satisfiability tests and time-bound adjustments for cumulative scheduling problems. *Annals of operations research*, 92:305–333, 1999. doi:10.1023/a:1018995000688.

3 Philippe Baptiste and Claude Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints: an international journal*, 5(1/2):119–139, January 2000. doi:10.1023/a:1009822502231.

4 Paul Beame and Toniann Pitassi. Simplified and improved resolution lower bounds. In *Proceedings of the 37th annual symposium on foundations of computer science*, FOCS '96, page 274, USA, 1996. IEEE Computer Society.

5 Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In *Proceedings of the 8th international conference on principles and practice of constraint programming*, CP '02, pages 63–79, Berlin, Heidelberg, 2002. Springer-Verlag. doi:10.1007/3-540-46135-3_5.

6 Timo Berthold. Measuring the impact of primal heuristics. *Operations research letters*, 41(6):611–614, 1 November 2013. doi:10.1016/j.orl.2013.08.007.

7 Peter Brucker, Sigrid Knust, Arno Schoo, and Olaf Thiele. A branch and bound algorithm for the resource-constrained project scheduling problem. *European journal of operational research*, 107(2):272–288, 1 June 1998. doi:10.1016/s0377-2217(97)00335-4.

8 Jacques Carlier and Emmanuel Néron. On linear lower bounds for the resource constrained project scheduling problem. *European journal of operational research*, 149(2):314–324, 1 September 2003. doi:10.1016/s0377-2217(02)00763-4.

9 Geoffrey Chu, Peter J Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed: The chuffed CP solver. https://github.com/chuffed/chuffed.

10 Rina Dechter. Learning while searching in constraint-satisfaction-problems. In *Proceedings of the fifth AAAI national conference on artificial intelligence*, AAAI'86, pages 178–183, Philadelphia, Pennsylvania, 1986. AAAI Press.

11 Delft High Performance Computing Centre. DelftBlue, 2024.

12 Emir Demirović, Maarten Flippo, Imko Marijnissen, Konstantin Sidorov, and Jeff Smits. Pumpkin: a lazy clause generation constraint solver in Rust. https://github.com/ConSol-Lab/Pumpkin, 2024.

13 Ulrich Dorndorf. Consistency tests. In *Project scheduling with time windows*, chapter 4, pages 31–65. Physica-Verlag HD, Heidelberg, 2002. doi:10.1007/978-3-642-57506-8_4.

14 Thibaut Feydy and Peter J Stuckey. Lazy clause generation reengineered. In *Proceedings of the 15th international conference on principles and practice of constraint programming*, CP'09, pages 352–366, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-04244-7_29.

15 John Forrest, Ted Ralphs, Stefan Vigerske, Haroldo Gambini Santos, John Forrest, Lou Hafer, Bjarni Kristjansson, jpfasano, EdwinStraver, Jan-Willem, Miles Lubin, rlougee, a-andre, jpgoncal, Samuel Brito, h-i-gassmann, Cristina, Matthew Saltzman, tosttost, Bruno Pitrus, Fumiaki Matsushima, Patrick Vossler, Ron @ Swgy, and to-st. coin-or/Cbc: Release releases/2.10.12, 2024. doi:10.5281/ZENODO.13347261.

16 Steven Gay, Renaud Hartert, and Pierre Schaus. Time-table disjunctive reasoning for the cumulative constraint. In *Integration of AI and OR techniques in constraint programming*, Lecture notes in computer science, pages 157–172, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-18008-3_11.

17 Gecode Team. Gecode: Generic constraint development environment. http://www.gecode.org, 2006.

18 Vincent Gingras and Claude-Guy Quimper. Generalizing the edge-finder rule for the cumulative constraint. In *Proceedings of the twenty-fifth international joint conference on artificial intelligence*, IJCAI'16, pages 3103–3109. AAAI Press, 9 July 2016. doi:10.5555/3061053.3061056.

19 Armin Haken. The intractability of resolution. *Theoretical computer science*, 39:297–308, 1985. doi:10.1016/0304-3975(85)90144-6.

20 Q Huangfu and J A J Hall. Parallelizing the dual revised simplex method. *Mathematical programming computation*, 10(1):119–142, March 2018. doi:10.1007/s12532-017-0130-5.

21 Joey Hwang and David G Mitchell. 2-way vs. d-way branching for CSP. In *Principles and practice of constraint programming - CP 2005*, Lecture notes in computer science, pages 343–357, Berlin, Heidelberg, 1 October 2005. Springer Berlin Heidelberg. doi:10.1007/11564751_27.

22 Roger Kameugne, Laure Pauline Fotso, Joseph Scott, and Youcheu Ngo-Kateu. A quadratic edge-finding filtering algorithm for cumulative resource constraints. *Constraints: an international journal*, 19(3):243–269, July 2014. doi:10.1007/s10601-013-9157-z.

23 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103, Boston, MA, 1972. Springer US. doi:10.1007/978-1-4684-2001-2_9.

24 Robert Klein and Armin Scholl. Computing lower bounds by destructive improvement: An application to resource-constrained project scheduling. *European journal of operational research*, 112(2):322–346, January 1999. doi:10.1016/s0377-2217(97)00442-6.

25 Rainer Kolisch and Arno Sprecher. PSPLIB - a project scheduling problem library. *European journal of operational research*, 96(1):205–216, January 1997. doi:10.1016/s0377-2217(96)00170-1.

26 Oumar Koné, Christian Artigues, Pierre Lopez, and Marcel Mongeau. Event-based MILP models for resource-constrained project scheduling problems. *Computers & operations research*, 38(1):3–13, 1 January 2011. doi:10.1016/j.cor.2009.12.011.

27 Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of satisfiability*, Frontiers in artificial intelligence and applications, chapter 4. IOS Press, 2 February 2021. doi:10.3233/faia200987.

28 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*. ACM, 2002. doi:10.1109/dac.2001.935565.

29 Wim Nuijten. *Time and resource constrained scheduling: a constraint satisfaction approach*. PhD thesis, Technische Universiteit Eindhoven, Eindhoven, 1994. doi:10.6100/IR431902.

30 Ramón Alvarez-Valdés Olaguíbel and José Manuel Tamarit Goerlich. Heuristic algorithms for resource-constrained project scheduling: A review and an empirical analysis. *Advances in project scheduling*, pages 113–134, 1989.

31 Laurent Perron and Frédéric Didier. CP-SAT, 7 May 2024.

32 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of constraint programming*. Foundations of artificial intelligence. Elsevier Science, London, England, 18 August 2006. doi:10.1016/s1574-6526(06)x8001-x.

33 Andreas Schutt, Thibaut Feydy, Peter J Stuckey, and Mark G Wallace. Explaining the cumulative propagator. *Constraints: an international journal*, 16(3):250–282, July 2011. doi:10.1007/s10601-010-9103-2.

34 Andreas Schutt, Thibaut Feydy, Peter J Stuckey, and Mark G Wallace. Solving RCPSP/max by lazy clause generation. *Journal of scheduling*, 16(3):273–289, June 2013. doi:10.1007/s10951-012-0285-x.

35 Peter J Stuckey. RCPSP. https://people.eng.unimelb.edu.au/pstuckey/rcpsp/. Accessed: 2025-3-29.

36 Peter J Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The MiniZinc Challenge 2008–2013. *AI magazine*, 35(2):55–60, 1 June 2014. doi:10.1609/aimag.v35i2.2539.

37 Petr Vilím. $o(n \log n)$ filtering algorithms for unary resource constraint. In Jean-Charles Régin and Michel Rueher, editors, *Proceedings of CP-AI-OR 2004*, volume 3011 of *Lecture Notes in Computer Science*, pages 335–347, Nice, France, April 2004. Springer-Verlag. doi:10.1007/978-3-540-24664-0_23.

38 Petr Vilím. Computing explanations for the unary resource constraint. In Roman Barták and Michela Milano, editors, *Integration of AI and OR techniques in constraint programming for combinatorial optimization problems, second international conference, CPAIOR 2005, Prague, Czech Republic, May 30 - June 1, 2005*, volume 3524 of *Lecture Notes in Computer Science*, pages 396–409. Springer, 2005. doi:10.1007/11493853_29.

39 Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In *Proceedings of the 8th international conference on integration of Ai and OR techniques in constraint programming for combinatorial optimization problems*, CPAIOR'11, pages 230–245, Berlin, Heidelberg, 2011. Springer-Verlag. doi:10.1007/978-3-642-21311-3_22.

40 Petr Vilím, Philippe Laborie, and Paul Shaw. Failure-directed search for constraint-based scheduling. In *CPAIOR '15: proceedings of the 12th international conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems*, Barcelon, Spain, 2015. Springer-Verlag.

A Conversion of $3n$ problem proofs into pigeonhole principle pseudo-proofs

For convenience, we reproduce the definition of the C-RES proof system subject to the notation differences between our paper and the work of Hwang and Mitchell [21].

▶ **Definition 18.** *Given a set of variables $\mathcal{X}$ with domains $\mathcal{D}$, a* ***clause*** *is a constraint of the form $\left(\bigvee_{j=1}^{N} [\![x_j^+ = v_j^+]\!]\right) \vee \left(\bigvee_{j=1}^{M} \neg[\![x_j^- = v_j^-]\!]\right)$ with $x_j^\pm \in \mathcal{X}, v_j^\pm \in \mathcal{D}(x)$.*

▶ **Definition 19.** *Given a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, a sequence of clauses $\mathcal{P}$ is an (unsatisfiability)* ***proof*** *of this CSP if the last clause is empty, and every clause $\omega \in \mathcal{P}$ is produced by one of the following rules:*

Resolution *ω can be represented as $\omega' \vee \omega''$ for some clauses ω', ω'' such that $\omega' \vee [\![x = v]\!]$ and $\omega'' \vee \neg[\![x = v]\!]$ are clauses encountered earlier in the proof. We denote the result of this operation as $\omega' \diamond \omega''$.*

Domain clause *$\omega = \bigvee_{v \in \mathcal{D}(x)} [\![x = v]\!]$ for some variable x.*

Unique value clause *$\omega = \neg[\![x = u]\!] \vee \neg[\![x = v]\!]$ for some variable x and two distinct values $u, v \in \mathcal{D}(x)$.*

Constraint clause *ω is* ***implied*** *by some constraint $c \in \mathcal{C}$, that is, any solution satisfying c also satisfies ω.*

Each step of the proof is either a " consistency" clause encoding that any variable is assigned to exactly one value, a constraint clause (which in our case corresponds to a propagation), or a resolution of two clauses (which typically corresponds to branching or clause learning). The original C-RES definition by Hwang and Mitchell also restricts $\mathcal{C}$ to only contain nogoods, and thus the constraint clauses are the negations of nogoods in $\mathcal{C}$. Again, that difference does not impact the proof definition, because any constraint can be replaced by a set of all nogoods implied by it, which impacts the size of $\mathcal{C}$ *but not the proof length.*

In this appendix, we show that the proofs of $3n$ instances can be encoded as the proofs of the *pigeonhole formula* [19] with $3n$ variables, with clauses encoding the pigeonhole subformulas for up to $2n$ variables. We use the following definition of a pigeonhole problem in this text:

▶ **Definition 20.** *A* ***pigeonhole formula*** PHP_m *is the unsatisfiable propositional formula on $(m-1) \times m$ variables $p_{i,j}, 1 \leq i \leq m-1, 1 \leq j \leq m$ having the following clauses:*

Pigeon clauses *Any pigeon is placed in a hole: $p_{1,i} \vee p_{2,i} \vee \cdots \vee p_{m-1,i}$ $\forall 1 \leq i \leq m-1$.*

Hole clauses *No hole contains two pigeons: $\bar{p}_{i,j} \vee \bar{p}_{i,k}$ $\forall 1 \leq i \leq m-1, 1 \leq j < k \leq m$.*

To simplify the description of the encoding, we switch to a more lenient notion of a proof proposed by Beame and Pitassi [4]:

▶ **Definition 21.** *Given a formula PHP_m, an assignment α is called* ***critical*** *if some $(m-1)$ pigeons are assigned to $(m-1)$ distinct holes, that is, $p_{1,\pi_1}, \ldots, p_{m-1,\pi_{m-1}}$ are satisfied by α for some pairwise different $\pi_j \in [1, m], 1 \leq j \leq m-1$; if additionally the only assigned pigeon has index k, then α is called* ***k-critical****.*

A pair of clauses ω', ω'' is ***congruent*** *if ω' and ω'' are equal for any critical assignment. If any critical assignment satisfying clauses ω' and ω'' also satisfies another clause ρ, then this clause is a* ***critical implication*** *by ω' and ω''.*

One of the immediate simplifications resulting from the notion of congruency is that we can assume without loss of generality that all clauses defined on pigeonhole variables are defined without negations:

▶ **Proposition 22** (Beame and Pitassi [4], Section 3)**.** *Given a formula PHP_m, any clause ω is congruent to a* ***positive clause*** *ω^+ obtained by replacing all literals $\bar{p}_{i,j} \in \omega$ with a negative polarity by the conjunction $\bar{p}_{i,j} \mapsto \bigvee_{k \in [1,m], k \neq i} p_{i,k}$ of variables over all pigeons except the i-th one.*

The core result of this appendix is the reduction from $3n$ proofs to pigeonhole proofs:

▶ **Lemma 23.** *Any proof $\mathcal{P}$ of a $3n$ problem instance can be encoded into a sequence of clauses $\mathcal{P}'$ with $|\mathcal{P}'| \leq |\mathcal{P}|$ such that the final clause is empty and any clause is either a pigeon clause*[4] *of* PHP_{3n}*, a critical implication of previous clauses, or a* ***subproblem*** *clause of the form*

$$\sigma(K,V) := \bigvee_{k\in[1,3n-1]\setminus K, v\in V} p_{k,v}, |V| > |K| \tag{7}$$

with $1 < |V| \leq 2n$.

We start by replacing the original $3n$ problem with a simpler formulation that does not invalidate unsatisfiability proofs of $3n$ problems.

▶ **Definition 24.** *Given a set of variables X, the* $\texttt{Alldifferent}(X)$ *constraint is true for the solutions in which any two distinct variables $x, y \in X$ are assigned to different values.*

▶ **Lemma 25.** *Any proof of a $3n$ problem instance is also a proof of a CSP on the same variables and constraints* $\texttt{Alldifferent}(G)$ *with $G \in \{X \cup Y, Y \cup Z, Z \cup X\}$.*

Proof. Let $\mathcal{P}$ be a proof of the $3n$ problem instance, and ω is a constraint clause in $\mathcal{P}$ with respect to the original $3n$ instance. We show that this is also a constraint clause with respect to the CSP in the lemma statement. This is sufficient to show that $\mathcal{P}$ is also a valid proof of the new problem, because the resolution steps only depend on the previous clauses. As for the domain and unique value clauses, they only depend on the underlying variable definitions, which are the same between two problem instances, and are also valid for the new problem instance.

Without loss of generality, suppose that a clause ω is implied by the first constraint in Definition 5. Consider an arbitrary solution $\mathcal{I}$ that falsifies ω and thus also falsifies the `Cumulative` constraint in question. Any overflow of such a constraint can be described as either scheduling two tasks from X in parallel, scheduling one task from X and one task from Y in parallel, or scheduling three tasks from Y in parallel. In each case, this assignment violates the $\texttt{Alldifferent}(X \cup Y)$. Thus, $\texttt{Alldifferent}(X \cup Y)$ is false whenever ω is false, or, in other words, ω is true whenever $\texttt{Alldifferent}(X \cup Y)$ is true. ◀

We now complete the reduction to pigeonhole proofs as follows:

Proof of Lemma 23. Introduce a one-to-one mapping between atomic predicates in $\mathcal{P}$ and pigeonhole variables with $[\![x_j = v]\!] \leftrightarrow p_{v,j}$ for some ordering of variables $\mathcal{X} = \{x_1, \ldots, x_{3n}\}$; we further refer to this as the canonical mapping. We show that the proof $\mathcal{P}'$ produced by (i) replacing atomic constraints in C-RES clauses with canonical mapping and (ii) discarding tautologies satisfies the lemma conditions. More specifically, we show that each clause $\omega_P \in \mathcal{P}'$ produced from a C-RES clause $\omega_C \in \mathcal{P}$ either falls into one of the three stated categories or is tautologically true. Additionally, we enforce the following invariant: if a non-tautological clause ω_C is falsified by an assignment $\mathcal{I}$ of $(3n-1)$ variables to $(3n-1)$ different values, then the critical assignment produced from it by canonical mapping falsifies ω_P.

We start with the " consistency" clauses. If $\omega_C = [\![x_k = 1]\!] \vee \cdots \vee [\![x_k = 3n-1]\!]$ is a *domain* clause, then $\omega_P = p_{1,k} \vee \cdots \vee p_{3n-1,k}$ is a valid clause in $\mathcal{P}'$ as a pigeon clause of PHP_{3n}. Additionally, if $\mathcal{I}$ falsifies ω_P, that means that x_k is unassigned; the canonical mapping leaves

[4] Hole clauses are tautological with respect to critical assignments.

k-th pigeon unassigned, which falsifies ω_P. On the other hand, if $\omega_C = \neg[\![x_k = u]\!] \vee \neg[\![x_k = v]\!]$ is a *unique value* clause, then $\omega_P = \bar{p}_{u,k} \vee \bar{p}_{v,k}$ is true for any *critical* assignment and is thus a tautology.

Next, consider the case when ω_C is a *constraint* clause. By Lemma 25, we can assume without loss of generality that $\omega_C \in \mathcal{P}$ is a constraint clause with respect to $\texttt{Alldifferent}(X \cup Y)$. Thus, if ω_C is false, then any assignment of $X \cup Y$ assigns the same values to some two variables in this set. By Hall's theorem, that means that there is a set of variables $Q \subseteq X \cup Y$ with indices V and a set of values $K \subset [1, 3n-1]$ such that $|K| < |V| \leq 2n$ and assigning all variables in Q to values in K falsifies ω_C. Conversely, that means that assigning *some* variable in Q to a value outside of K, equivalently, in $\bar{K} = [1, 3n-1] \setminus K$ satisfies ω_C. We have encoded a condition $\bigvee_{v \in V}(x_v \notin K) = \bigvee_{v \in V}(x_v \in \bar{K})$ implying ω_C, which after applying the canonical mapping becomes $\omega_P = \bigvee_{k \in \bar{K}, v \in V} p_{k,v}$ and can be directly added to the proof, since it coincides with $\sigma(K, V)$ from Equation 7.

Last, suppose that $\omega_C = \omega'_C \diamond \omega''_C$ is a *resolution* of some previously introduced propositional clauses ω'_C, ω''_C. Let ω'_P and ω''_P be the corresponding clauses in $\mathcal{P}'$; suppose first that neither of those clauses was skipped. In that case, we observe the encoding ω_P of ω_C is critically implied by ω'_P and ω''_P: otherwise, there would be a critical assignment α satisfying both ω'_P and ω''_P but falsifying ω_P, and applying the canonical mapping to it yields an assignment $\mathcal{I}$ satisfying ω'_C and ω'_C but falsifying ω_C, contradicting the soundness of the resolution step. In case when one (or both) of the original clauses is a tautology, the same reasoning holds, because resolving with a tautological clause does not increase the set of falsifying assignments. ◀

B Exponential lower bound on pigeonhole proofs

We start with an auxiliary fact that establishes a lower bound on the width of a valid proof:

▶ **Lemma 26.** *Let $n \in \mathbb{N}_{\geq 1}, 0 < \gamma \leq 1$, and consider a sequence $\mathcal{P}$ such that any clause in it is either a pigeon clause of $\mathrm{PHP}_{(2+\gamma)n}$, a critical implication of the previous clauses, or a subproblem clause $\sigma(K, V)$, $1 \leq |K| < |V| \leq 2n$. Then $\mathcal{P}$ has a clause with at least $2\gamma n^2$ literals.*

Proof. Given a clause ω, let $\mathrm{Complexity}(\omega)$ be the smallest number of pigeon clauses that imply ω on all critical assignments; in particular, $\mathrm{Complexity}(\omega) = 1$ for all pigeon clauses $\omega \in \mathrm{PHP}_{(2+\gamma)n}$, $\mathrm{Complexity}(\bot) = (2+\gamma)n$, and $\mathrm{Complexity}(\sigma(K, V)) = |V|$.

Observe that a proof contains a clause $\hat{\omega}$ of complexity greater than γn and not greater than $2n$. To verify this, let $V^* \leq 2n$ be the largest cardinality of a set V used in a subproblem clause. If $V^* > \gamma n$, then the subproblem clause that achieves this bound also satisfies the complexity bound. Otherwise, assume that this is not the case and that all proof clauses have complexity that is either at most γn or more than $2n$. Observe that if ω is critically implied by ω' and ω'', then $\mathrm{Complexity}(\omega) \leq \mathrm{Complexity}(\omega') + \mathrm{Complexity}(\omega'')$, since the implication relation is transitive. As the formula clauses have complexity 1 and subproblem clauses have complexity at most γn, then *all* their resolvents ω have to have the complexity of at most γn: otherwise, $\mathrm{Complexity}(\omega) \leq \gamma n + \gamma n \leq 2n$ violates our assumption. But then the same bound has to hold for the resolvents of those resolvents, and so on until the empty clause, which has complexity $(2+\gamma)n > 2n$, which is a contradiction.

We complete the proof by demonstrating that this clause satisfies the lemma conditions. Let $\hat{\Omega}$ be the set of formula clauses that implies $\hat{\omega}$ such that $\gamma n < |\hat{\Omega}| \leq 2n$; observe that $\hat{\omega}$ has at least $|\hat{\Omega}|\big((2+\gamma)n - |\hat{\Omega}|\big)$ literals [4, Lemma 1], and this expression achieves the smallest value $2\gamma n^2$ at the endpoints of the interval on $|\hat{\Omega}|$. ◀

We now derive an exponential bound on the proof length of PHP_{3n} with extra clauses:

▶ **Lemma 27.** *Let $n \in \mathbb{N}_{\geq 1}$, and consider a sequence of clauses $\mathcal{P}$ such that any clause is either a pigeon clause of PHP_{3n}, a resolution of the previous clauses, or a subproblem clause $\sigma(K, V)$ with $1 \leq |K| < |V| \leq 2n$. Then $\mathcal{P}$ has at least $2^{0.08n}$ clauses for sufficiently large n.*

Proof. Suppose this is not the case, and we discovered a proof of length $L < 2^{0.08n}$. For the rest of the proof, we say that the clause is *long* if it has at least n^2 literals. By pigeonhole principle, there is a variable $x_{i,j}$ that is contained in at least $\frac{Ln^2}{3n(3n-1)} \geq \frac{1}{9}L$ long clauses. Observe that setting $x_{i,j} = 1$ with $x_{k,j} = x_{i,\ell} = 0$ for all $k \neq i, \ell \neq j$ and discarding all satisfied clauses from both the proof and the formula yields a proof of a PHP_{3n-1} with the same assumptions on subproblem clauses but having at most $\frac{8}{9}L$ long clauses.

Repeating this variable elimination procedure for $T = \lceil \log_{9/8} L \rceil$ steps yields a proof of PHP_{3n-T} with at most $\left(\frac{8}{9}\right)^T L < 1$ long clause, or, in other words, *no* long clauses. On the other hand, we have a pigeonhole formula with $3n - T = 3n - \lceil 0.08n \times \log_{9/8} 2 \rceil \geq \left(2 + \frac{1}{2}\right) n$ variables and a proof with no clauses of length at least n^2. However, we can now rewrite $3n - T = \hat{\gamma} n$ with $\hat{\gamma} \geq 1/2$ and apply Lemma 26, which implies that the proof in question has to have a clause with at least $2\hat{\gamma}n^2 \geq n^2$ literals. That is a contradiction, therefore, $\mathcal{P}$ has to have length of at least $2^{0.08n}$. ◀

The derivation of the intractability theorem bound is a compilation of the results above:

Proof of Theorem 6. Let $\mathcal{P}'$ be the pigeonhole formula proof produced from $\mathcal{P}$ from Lemma 23 with $|\mathcal{P}| \geq |\mathcal{P}'|$. By Lemma 27, we can rewrite the bound further as $|\mathcal{P}| = \Omega(2^{0.08n})$, and the theorem bound follows from the $1.05 < 2^{0.08}$ inequality. ◀

C Explanations

Assume without loss of generality that $\mathrm{EST}_i \leq \mathrm{EST}_j$, as otherwise the tasks i and j can be swapped. Then we generate the explanations $\mathcal{E}_{sn}(s_i, s_j, p_i, p_j)$ as prescribed by Table 1.

Table 1 List of explanations $\mathcal{E}_{sn}(s_i, s_j, p_i, p_j)$ produced for each pair of atomic constraints p_i, p_j.

Atomic predicate p_i	Atomic predicate p_j	Explanation $\mathcal{E}_{sn}(s_i, s_j, p_i, p_j)$
$[\![s_i \leq v_i]\!]$	$[\![s_j \geq v_j]\!]$	$[\![s_j \geq v_i + d_i]\!] \wedge [\![s_i \leq v_j - d_i]\!]$
$[\![s_i \leq v_i]\!]$	$[\![s_j \neq v_j]\!]$	$[\![s_j \geq v_i + d_i]\!] \wedge [\![s_i \leq v_j - d_i + 1]\!]$
$[\![s_i \leq v_i]\!]$	$[\![s_j = v_j]\!]$	$[\![s_j \geq v_i + d_i]\!] \wedge [\![s_i \leq v_j - d_i]\!]$
$[\![s_i \neq v_i]\!]$	$[\![s_j \geq v_j]\!]$	$[\![s_j \geq v_i + d_i - 1]\!] \wedge [\![s_i \leq v_j - d_i]\!]$
$[\![s_i \neq v_i]\!]$	$[\![s_j \neq v_j]\!]$	$[\![s_j \geq v_i + d_i - 1]\!] \wedge [\![s_i \leq v_j - d_i + 1]\!]$
$[\![s_i \neq v_i]\!]$	$[\![s_j = v_j]\!]$	$[\![s_j \geq v_i + d_i - 1]\!] \wedge [\![s_i \leq v_j - d_i]\!]$
$[\![s_i = v_i]\!]$	$[\![s_j \geq v_j]\!]$	$[\![s_j \geq v_i + d_i]\!] \wedge [\![s_i \leq v_j - d_i]\!]$
$[\![s_i = v_i]\!]$	$[\![s_j \neq v_j]\!]$	$[\![s_j \geq v_i + d_i]\!] \wedge [\![s_i \leq v_j - d_i + 1]\!]$
$[\![s_i = v_i]\!]$	$[\![s_j = v_j]\!]$	$[\![s_j \geq v_i + d_i]\!] \wedge [\![s_i \leq v_j - d_i]\!]$

The common idea behind the listed explanations is to lift the bounds in a way that ensures the absence of overlap when one of the two predicates is true. For example, if $p_i = [\![s_i \leq v_i]\!]$ and $p_j = [\![s_j \neq v_j]\!]$, then the first predicate $[\![s_j \geq v_i + d_i]\!]$ in the explanation states the following fact: 'if p_i is true then s_j starts only after task i has ended'. Similarly, the second predicate $[\![s_i \leq v_j - d_i + 1]\!]$ in the explanation states that task i should end before task j starts; we add an extra unit of time since we know that $[\![s_j \neq v_j]\!] \implies [\![s_j \geq v_j + 1]\!]$.

D Novel bounds

We report the bounds discovered by our approach if they are *both* better than the previous reported bounds and are not directly reproducible without using our approach. More precisely, we report bounds that are simultaneously (a) tighter than the bounds reported in the previous sources known to us that used the same benchmarks [25, 40, 39, 35], and (b) either tighter than any bound derived by non-disjunctive approaches (CP-SAT and baseline Pumpkin) or matches it but was derived at least ten times faster than with any other approach.

Novel upper bounds (makespans) are reported in Table 2, and novel lower bounds are reported in Table 3; the same data is available as supplementary materials. All durations reported in both tables are in `MM:SS` format; Table 3 additionally reports closed instances and bounds derived with mining. To indicate the remaining optimality gap, we also state the best known lower bound in Table 2 and the best known makespan in Table 3; in either case, that bound is the tightest among the previously reported values and the values discovered by non-disjunctive approaches.

Table 2 Novel upper bounds derived with our approach.

Problem	Collection	#	Ref. objective	Our objective	Time	Best bound
RCPSP/max	C	61	378	374	07:03	345
RCPSP/max	C	69	380	371	08:47	356
RCPSP	J90	5-4	103	102	01:02	101
RCPSP	J120	27-7	125	124	08:21	122
RCPSP	J120	46-10	188	187	57:34	184
RCPSP	J120	9-4	87	86	06:40	85

E Optimality proof for the RCPSP/max instance #64 of collection J30

In this appendix, we support the optimality claim on instance # 64 from the J30 collection by showing that the tasks with indices $\Omega = \{1, \ldots, 9, 11, \ldots, 30\}$ are pairwise disjoint, which implies optimality since $\sum_{t\in\Omega} d_t = 169$. This instance has five resources, each with the capacity of five units.

To decrease the number of considered pairs, we partition the claimed clique into six parts and record for each of them the lowest amount of each resource consumed by a task:

1. $\Omega_1 = \{6, 11, 21, 24\}$; lowest resource consumption is $(2, 4, 1, 2, 2)$.
2. $\Omega_2 = \{18, 23, 28\}$; lowest resource consumption is $(2, 1, 2, 2, 4)$.
3. $\Omega_3 = \{30\}$; lowest resource consumption is $(4, 1, 3, 2, 1)$.
4. $\Omega_4 = \{1, 2, 3, 4, 5, 8, 9, 12, 13, 14, 15, 16, 17, 19, 20, 25, 29\}$; lowest resource consumption is $(1, 1, 1, 4, 1)$.
5. $\Omega_5 = \{7\}$; lowest resource consumption is $(3, 5, 1, 1, 2)$.
6. $\Omega_6 = \{22, 26, 27\}$; lowest resource consumption is $(1, 1, 5, 1, 2)$.

The last two groups consume 100% of either resource #2 or resource #3; since all other tasks consume some amount of all resources, any pair of tasks involving a task in $\Omega_5 \cup \Omega_6$ is disjoint. The same holds for any pair of tasks involving Ω_4 and any task in $\Omega_1 \cup \cdots \cup \Omega_4$ due to the overflow of resource #4. Thus, it remains to show that any pair of tasks in $\Omega_1 \cup \Omega_2 \cup \Omega_3$ is disjoint. Table 4 handles the six cases for all remaining pairs of task groups.

Table 3 Novel lower bounds derived with our approach. Rows with an asterisk highlight closed instances; rows with a dagger highlight objective values derived with enabled mining.

Problem	Collection	#	Ref. bound	Our bound	Time	Best objective
RCPSP/max	J30	64	141	169*	< 1s	169
RCPSP/max	J30	65	144	162*	< 1s	162
RCPSP/max	J30	151	142	157*	< 1s	157
RCPSP/max	J30	153	163	176*	< 1s	176
RCPSP/max	J30	155	125	154*	< 1s	154
RCPSP/max	UBO50	10	154	186*	00:05	186
RCPSP/max	UBO100	4	303	365	44:07	376
RCPSP/max	UBO100	7	281	373	51:47	395
RCPSP/max	UBO100	8	364	376	29:06	385
RCPSP/max	UBO100	32	353	414	53:52	434
RCPSP/max	UBO100	33	328	397	47:04	406
RCPSP/max	UBO200	35	610	747	57:27	823
RCPSP/max	UBO200	62	621	702	56:46	796
RCPSP/max	UBO500	36	1 285	1 423	26:37	1 908
RCPSP/max	UBO500	61	1 296	1 533	59:50	1 944
RCPSP/max	UBO500	64	1 329	1 489	55:08	1 932
RCPSP	J120	46-9	157	159	18:58	166
RCPSP	Pack-d	2	745	746†	25:39	747
RCPSP	Pack-d	3	624	625†*	13:54	625
RCPSP	Pack-d	47	2 740	2 742†	52:35	2 745

Table 4 Disjointness justifications for every pair of tasks in $\Omega_1 \cup \Omega_2 \cup \Omega_3$.

	Ω_1	Ω_2	Ω_3
Ω_1	#2: 4 + 4	#5: 2 + 4	#1: 2 + 4
Ω_2	—	#5: 4 + 4	#5: 2 + 4
Ω_3	—	—	#1: 4 + 4

Balancing Latin Rectangles with LLM-Generated Streamliners

Florentina Voboril
Algorithms and Complexity Group, TU Wien, Austria

Vaidyanathan Peruvemba Ramaswamy
Algorithms and Complexity Group, TU Wien, Austria

Stefan Szeider
Algorithms and Complexity Group, TU Wien, Austria

Abstract

We present an integration of Large Language Models (LLMs) with streamlining techniques to find well-balanced Latin rectangles. Our approach combines LLM-generated streamlining constraints that effectively partition the search space, directing constraint solvers toward structured subspaces containing high-quality solutions. Our methodology extends LLM-generated streamliners, as Voboril et al. (2024) introduced for decision problems, to the optimization context through techniques that incrementally refine the objective function value.

We propose two complementary strategies to orchestrate sets of streamliners: an incremental mechanism that utilizes improving solutions to initialize subsequent search processes, and an evolutionary framework that maintains and refines effective streamliner populations. Our experiments demonstrate that our approach successfully reduces established minimum imbalance values for partially spatially balanced Latin rectangles across multiple problem dimensions. The results validate the efficacy of combining LLMs with constraint programming methodologies for tackling problems characterized by complex global constraints.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Mathematics of computing → Combinatorial optimization; Mathematics of computing → Combinatorial algorithms; Information systems → Language models

Keywords and phrases Balanced Latin Rectangles, Streamliners, Large Language Models, Warm-starts, Evolutionary Search

Digital Object Identifier 10.4230/LIPIcs.CP.2025.36

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.15616074` [19]

Funding Austrian Science Fund (FWF) projects 10.55776/COE12 and 10.55776/P36420.

1 Introduction

Latin rectangles are combinatorial structures consisting of k rows and n columns filled with n symbols, where each symbol appears exactly once in each row and column. When these structures are balanced – meaning the distance between any pair of symbols over all rows is minimized – they become particularly valuable for experimental design, especially in agricultural field trials. *Spatially balanced Latin rectangles (BLRs)* help minimize bias due to spatial correlation in experimental plots, leading to more accurate statistical analyzes and reliable results across fields such as agriculture, drug testing, and psychology [2, 17, 6, 16].

Finding optimally balanced Latin rectangles presents a significant computational challenge. The imbalance of a Latin rectangle is measured as the sum of absolute differences between actual and ideal distances between all pairs of symbols. For many combinations of number of rows and columns, determining whether a given imbalance value is optimal remains an open question. Previous work by Díaz et al. [2] established upper bounds for BLRs of various

© Florentina Voboril, Vaidyanathan Peruvemba Ramaswamy, and Stefan Szeider;
licensed under Creative Commons License CC-BY 4.0

sizes, with provably optimal solutions known only for specific dimensions. Despite advances using constraint programming, mixed integer programming, and local search methods, the computational complexity has limited progress beyond rectangles of moderate size [9, 10].

The technique of *streamlining* – adding constraints that focus the search on promising regions of the solution space – was initially introduced by Gomes and Sellmann [3] for related combinatorial design problems, including spatially balanced Latin squares. Streamlining constraints partition the search space, guiding the solver toward structured subspaces likely to contain high-quality solutions. While effective, the manual design of streamliners requires domain expertise and experimentation [7].

In this paper, we push the boundaries of balanced Latin rectangles by combining established streamlining techniques with the novel approach of generating streamliners using *Large Language Models (LLMs)*. We build upon the recent work by Voboril et al. [18], who introduced StreamLLM, a method for using LLMs for generating streamliners for decision problems, and adapt this approach to optimization problems. Our method improves most of the best-known bounds for BLRs in the range of $k \in [3, 12]$ and $n \in [9, 13]$, demonstrating the effectiveness of this hybrid approach.

Our contribution is twofold. First, we extend StreamLLM to address *optimization problems* instead of decision problems, introducing techniques to guide the search toward solutions with better objective values. Second, we present two complementary strategies for orchestrating the generation of a set of streamliners that together help to obtain Latin rectangles with lower imbalance than any known before of the same dimension.

The first strategy is the *incremental warmstart* approach that uses improved solutions as starting points for subsequent searches. The second strategy is an *evolutionary* approach that maintains a population of effective streamliners, combining them to generate increasingly powerful constraints.

Our experimental results show that our method outperforms previous approaches, improving upper bounds on imbalance for 32 out of 44 instances. Table 1 summarizes our results compared to previously known bounds, highlighting the cases where we establish new record values. Our implementation draws on five different LLM models and employs various prompting strategies to generate diverse and effective streamliners, enabling a thorough exploration of the solution space.

Table 1 Final results. The gray cells are already proven optimal and hence excluded from our experiments. The purple cells are where our approach finds a BLR with strictly better imbalance. Of the remaining white cells, the cells with bold text are where our approach matches the previously known imbalance bound.

$n\backslash k$	2	3	4	5	6	7	8	9	10	11	12
8	40.0	36.0	32.0	30.0	**24.0**	**28.0**	0.0				
9	65.3	**56.0**	**56.0**	52.6	**48.0**	51.3	53.3	0.0			
10	92.0	80.0	78.6	66.6	70.0	72.0	82.6	70.0	40.0		
11	124.0	114.0	112.0	116.0	108.0	118.0	114.0	120.0	110.0	0.0	
12	168.0	154.0	154.6	145.3	**120.0**	164.0	148.0	154.0	**174.6**	147.3	0.0
13	218.6	198.0	211.3	210.6	204.0	203.3	220.6	214.0	240.6	224.0	274.0

2 Preliminaries

2.1 Balanced Latin Rectangles

In this section, we define the general notation and background related to balanced Latin rectangles. A *Latin rectangle* is an $k \times n$ grid where $k \leq n$ and each cell contains a symbol from 1 to n, such that no symbol appears more than once within a row or within a column. *Latin squares* (or Magic squares) are Latin rectangles where $n = k$. Latin squares are widely known and have been studied for centuries.

Díaz et al. [2] introduced the notion of balance in Latin rectangles, minimizing which helps preclude spatial-correlation artifacts and ensures statistical fairness while devising experiments. A *spatially balanced Latin rectangle* (or simply *balanced Latin rectangle*) is a Latin rectangle where the total distance between any pair of symbols is the same. Formally, $\text{dist}_i(r, s)$ denotes the distance between symbols r and s in the ith row. Note that $0 < \text{dist}_i(r, s) < n$. The distance between two symbols is defined as $\text{dist}(r, s) = \sum_{1 \leq i \leq k} \text{dist}_i(r, s)$. A Latin rectangle is *balanced* iff $\text{dist}(r, s)$ is the same for every pair $1 \leq r, s \leq n$. Letting $\text{Z}_{n,k}$ be $k(n+1)/3$, it is easy to see that in a balanced Latin rectangle, $\text{dist}(r, s) = \text{Z}_{n,k}$ for every pair.

However, balanced Latin rectangles only exist when $n = k$ and $n \not\equiv 1 \pmod 3$ [2], which rules out several combinations of n and k. Consequently, Díaz et al. [2] defined the imbalance of a Latin rectangle as a measure of how far it is from being balanced. The imbalance between a pair of symbols r, s is defined as $\mathbb{I}(r, s) = |\text{dist}(r, s) - \text{Z}_{n,k}|$ and the imbalance of a Latin rectangle L is defined as $\mathbb{I}(L) = \sum_{1 \leq r < s \leq n} \mathbb{I}(r, s)$. In the BLR problem, we are given two integers $k \leq n$, and the goal is to find a $k \times n$ Latin rectangle L such that $\mathbb{I}(L)$ is minimized. Note that $\text{Z}_{n,k}$ is always a rational number of the form $p/3$ and if either k or $n + 1$ are divisible by 3, then $\text{Z}_{n,k}$ is an integer. The same holds for $\mathbb{I}(L)$; hence, we always denote the imbalance values as x.0, x.3 or x.6, respectively, to represent x, $\text{x} + \frac{1}{3}$ or $\text{x} + \frac{2}{3}$ where $\text{x} \in \mathbb{Z}$.

1	2	3	4
2	4	1	3
3	1	4	2

Figure 1 Example of a 3×4 optimally balanced Latin rectangle L with imbalance 4. For the pair $(2, 3)$, the distance $\text{dist}(2, 3)$ is the sum of distances over each row, i.e., $1 + 3 + 3 = 7$, and the imbalance $\mathbb{I}(2, 3) = |\text{dist}(2, 3) - k(n + 1)/3| = 2$. By the same method, $\mathbb{I}(1, 3) = \mathbb{I}(2, 4) = 1$ and $\mathbb{I}(1, 2) = \mathbb{I}(1, 4) = \mathbb{I}(2, 4) = 0$. Finally, the imbalance of the Latin rectangle $\mathbb{I}(L)$ is the sum of all the pair-wise imbalances, which is 4.

2.2 LLM-generated Streamliners

Streamlining constraints or *streamliners* are constraints added to a constraint programming model in order to speed up the solving process by pruning the search space and guiding the solver towards more promising subspaces of the solution space. Streamliners have provided significant speedups across numerous problem domains [1, 2, 3, 4, 5, 7, 8, 11, 14].

By definition, they are not required to be *sound*, i.e., they need not preserve the set of feasible solutions and are allowed to remove some or even all of the solutions. Streamliners are a generalization of the following well-known constraints:

- *implied* or *redundant* constraints which do not alter the solution space,

- *symmetry-breaking* constraints which eliminate all but one solution from symmetric equivalence classes
- *dominance-breaking* constraints which eliminate potentially suboptimal solutions such that at least one of the optimal solution remains.

In our work, we use the term "streamliner" in a broader sense to refer to any constraint that can speed up the solving of a constraint programming model. Consequently, the streamliners that we present include but are not limited to the three types listed above.

Formerly, streamliners were generated manually for each problem which was labor-intensive and hard to scale, which motivated research into automating streamliner generation [12, 13, 15, 20]. Typically, these methods involved crafting combinations of atomic constraints which restrict the variable domains and testing them on a large pool of benchmark instances. This process was extremely resource-intensive, taking several days to come up with promising streamliners for a given problem.

Looking for a more scalable and efficient way to come up with high-quality streamliners, we turn our attention to *Large Language Models* or *LLMs*. LLMs are transformer architectures with billions of parameters that have been trained on large datasets and can produce human-like text and code. In recent times, LLMs have seen significant growth in innovation and use. This widespread use is a testament to the versatility and broad applicability of LLMs, such as, chatbots, translators, and coding assistants. Recent advancements have endowed LLMs with the power of reasoning and problem-solving which further enables them to manipulate mathematical expressions and assist in devising proofs. However, LLMs are not infallible and can produce incorrect yet believable output. As a result, independent verification is crucial to harness the power of LLMs.

In this work, we extend the StreamLLM approach developed by Voboril et al. [18]. In contrast to their work, we target optimization problems instead of decision problems, and we use more sophisticated procedures to traverse the space of solutions using the objective values as guidance. Please refer to Listing 1 for the MiniZinc model of the BLR problem that we use to test the performance of the different approaches and also provide to the LLM as context for generating streamlining constraints. Note that in the MiniZinc model: `pos[row,symbol]` denotes the position of a symbol in a row, `rect[row,col]` denotes the symbol at a specific cell of the Latin rectangle, `dist[r,s]` denotes $\mathrm{dist}(r, s)$, and `imbalances[r,s]` denotes $\mathbb{I}(r, s)$.

3 Our Approach

3.1 Framework

We introduce a novel approach to improve the performance of optimization problems in constraint programming. Our approach is based on the automatic generation of streamliners by using LLMs. We use BLR as the target problem to test and demonstrate the performance of our approach. Our completely autonomous procedure prompts the LLM to suggest streamlining constraints for the supplied (unstreamlined) MiniZinc model of the BLR problem. The generated streamliners are tested on the input instance for a short time to evaluate their performance relatively quickly.

In each iteration, one of several LLMs and one of three possible prompts is randomly chosen. This capitalizes on the strengths of different LLMs and different prompts. The first prompt `prompt_basic` (shown in Figure 2) asks the LLM to analyze the MiniZinc code and generate five new, creative, and syntactically correct streamliners. The formulation of the prompt is very similar to the prompt used by Voboril et al. [18]. The second

Listing 1 MiniZinc Model.

```
include "alldifferent.mzn";

% Input
int: k;
int: n;

% Alternate representation
array[1..k, 1..n] of var 1..n: pos;
constraint forall(row in 1..k) (
    alldifferent([pos[row, symbol] | symbol in 1..n])
);
constraint forall(symbol in 1..n) (
    alldifferent([pos[row, symbol] | row in 1..k])
);

% Solution representation for convenience
array[1..k, 1..n] of var 1..n: rect;
constraint forall(row in 1..k, col in 1..n, symbol in 1..n) (
    pos[row, symbol] = col <-> rect[row, col] = symbol
);

% Imbalance calculation
array[1..n, 1..n] of var 0..k*(n-1): dist;
array[1..n, 1..n] of var 0..max(k*(n+1)-3, 2*k*(n-2)): imbalances;
constraint forall(r in 1..n, s in 1..n) (
  if r < s then
    dist[r, s] = sum(row in 1..k) (abs(pos[row, r]-pos[row, s]))
  else
    dist[r, s] = 0
  endif
);
constraint forall(r in 1..n, s in 1..n) (
  if r < s then
    imbalances[r, s] = abs(3*dist[r, s]-k*(n+1))
  else
    imbalances[r, s] = 0
  endif
);

% Objective function
solve minimize sum(r in 1..n, s in 1..n where r < s) (
  imbalances[r, s]
);
```

Objective: Analyze the given MiniZinc code and suggest five additional constraints to enhance the problem-solving process. These constraints can include streamlining, implied, symmetry-breaking, or dominance-breaking constraints.

Steps:

1. Analyze Content: Read the provided MiniZinc code. Understand the problem being addressed, including its variables, constraints, and optimization goals.
2. Generate additional Constraints: Based on your analysis, create five unique constraints. These should offer targeted modifications or restrictions designed to reduce the search space effectively.
3. Always return your constraints as a JSON object, adhering to the structure: {"streamliner_1": "constraint <MiniZinc constraint>", ..., "streamliner_5": "constraint <MiniZinc constraint>"}.
   Your final output should exclusively be the JSON object containing the five constraints.

Compliance Rules:

1. Code Quality: All MiniZinc code provided for the constraints must be syntactically correct and functional. For some functions, you may need to include an additional library.
2. Creativity: You're encouraged to be innovative in proposing constraints, keeping in mind their purpose: to narrow down the search space efficiently without oversimplifying the problem.

Figure 2 `prompt_basic`.

prompt `prompt_combinations` asks for five combinations of streamliners (each consisting of multiple constraints) instead of five individual streamliners. The third prompt `prompt_description` is an extension of the first prompt with a detailed description of the BLR problem. In the following sections, we present two different approaches to use the produced streamliners. In both approaches, syntactically incorrect streamliners are simply discarded. All hyperparameter values used in these approaches were fixed on the basis of some preliminary experiments.

3.2 Incremental Warmstart Approach

Our *incremental warmstart* approach utilizes the warmstart annotation in MiniZinc. Warmstarting is a feature of certain solvers for optimization problems, where we can seed the solver's search with an initial feasible solution. This can be useful to guide the solver to find better solutions quicker. In the context of the BLR problem, we warmstart the solver with the best solution obtained so far.

Our approach, as shown in Figure 3, starts with an initial training time $t = 1$ minute, running the original encoding for that time and storing the solution s. Then, one of the LLMs is randomly chosen and asked to produce five new streamliners for the given problem. These streamliners are evaluated in parallel for t minutes. For the evaluation, the currently best solution s is used as warmstart. If a new solution s' is better than s, it replaces s as the current best solution. If there is no new improved solution after 5 iterations, the training time t is increased by 2 minutes. With a longer training time, there is a higher potential for

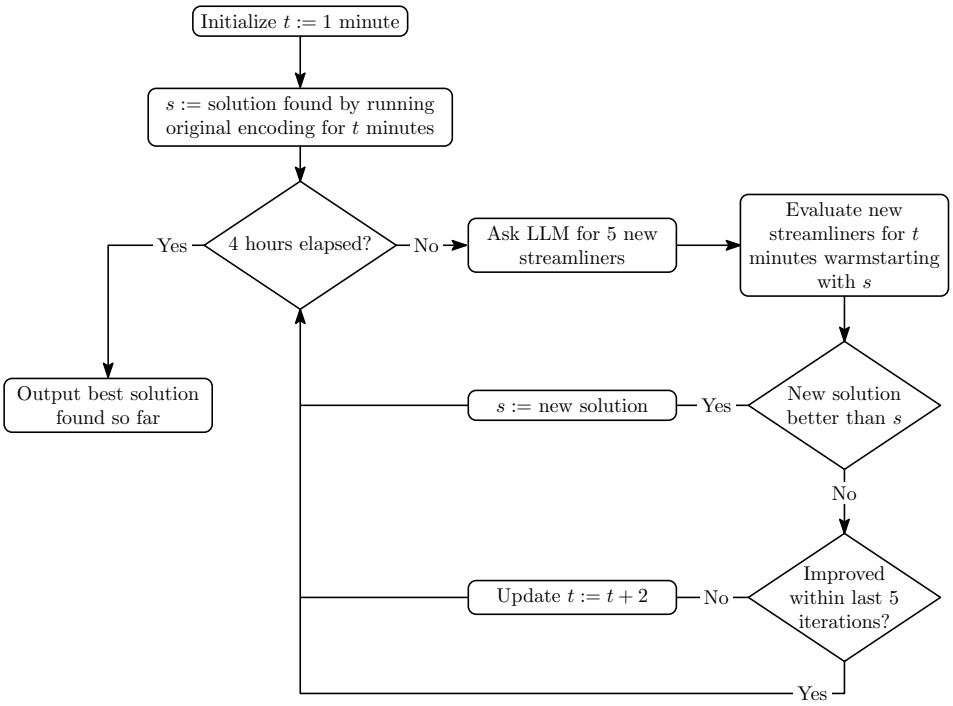


Figure 3 Diagram visualizing the incremental warmstart approach.

future streamliners to find better solutions. The process ends after a total of 4 hours have passed. At the end of the process, we can directly read off the final solution s. One variation of our incremental warmstart approach is to give it an already known solution (potentially from literature) right at the start. This makes the solving process even more efficient and enhances the chances to improve upon previously known bounds.

3.3 Evolutionary Approach

Evolutionary algorithms are optimization techniques inspired by the biological process of evolution. A set of possible solutions is stored as a population. In every generation, good solutions are selected, combined, and mutated to create potentially better solutions to add to the population.

Our evolutionary approach is shown in Figure 4. At the beginning, we run the original, unstreamlined encoding for three minutes to figure out its performance. Further, we create the set S for potential streamliners. At the beginning, S is empty. So we ask the randomly picked LLM to produce five new streamliners for the given problem. They are evaluated in parallel for 3 minutes. All streamliners that perform better than the original model after three minutes are added to the population, along with their corresponding imbalance values $\mathbb{I}$. This phase is called *exploration*. The first 20 streamliners added to the population are called the *original population*. Then, the *evolutionary* phase starts. In every iteration step, we sample 10 streamliners from our population according to probabilities proportional to $\mathbb{I}^{-4}$, favoring those with lower imbalance values. Then, these 10 streamliners are given to the LLM as reference to derive five new streamliners. This would be the combination and mutation part in terms of evolution. Again, all streamliners that produce better results than the original model are added to the population. We run this process for 5 hours. At the end of

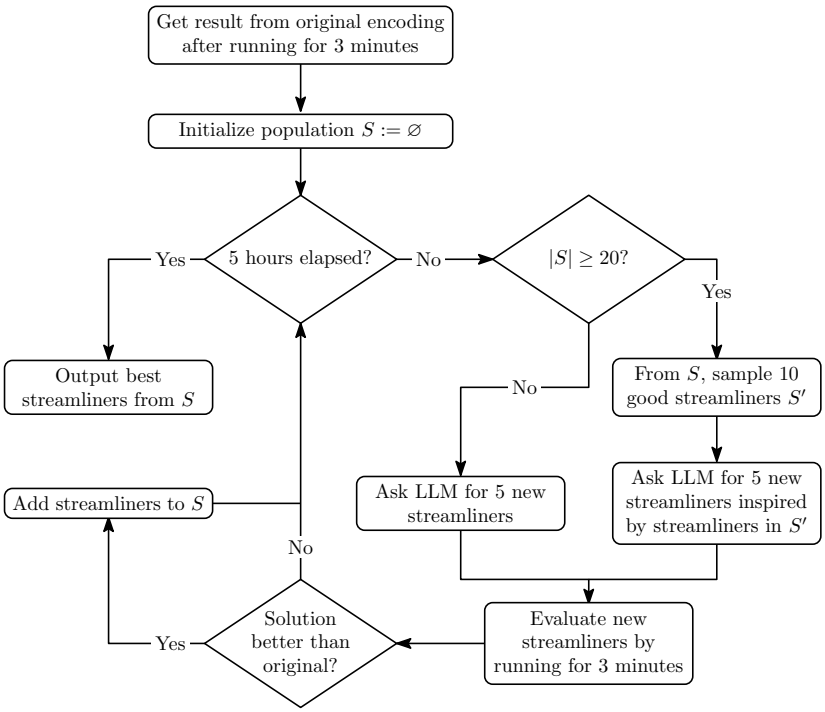


Figure 4 Diagram visualizing the evolutionary approach.

the process, we read off the three best-performing streamliners from the final population. These streamliners can potentially also be used for other instances of the same problem. Finally, we run these three streamliners in parallel for a further 4 hours and then report the best result. In our experiments, we also run a variant of this process called *exploration-only*, where we omit the evolutionary phase and only run the exploration phase for 5 hours. This is again followed by running the three best streamliners in parallel for a further 4 hours. The goal of the exploration-only variant is to assess whether the evolution indeed works well or whether the improvements are only due to the relatively long running time.

4 Experiments

The MiniZinc model, the Python implementation of the incremental warmstart and evolutionary approaches, and the minimum-imbalance Latin rectangle for each instance found by our approaches are available on Zenodo [19].

4.1 Setup and Hardware

We run all our experiments on compute nodes with 2.40GHz, 10-core 2×Intel Xeon E5-2640 v4 processors. We use MiniZinc version 2.9.2 for the incremental warmstart approach[1] and MiniZinc version 2.8.3 for the other experiments. We use Gurobi version 11.0.2 as the backend solver for MiniZinc, since previous work mainly used Gurobi and our preliminary experiments

[1] MiniZinc version 2.9.2 offers more robust support for the warmstart feature

showed it to outperform Chuffed. We use five LLMs, namely GPT-4o (openai/gpt-4o-2024-11-20), GPT-o3 (openai/o3-mini-high), Claude 3.7 Sonnet (anthropic/claude-3.7-sonnet), Deepseek R1 (deepseek/deepseek-r1), and Gemini 2.0 Flash (google/gemini-2.0-flash-001). We access all these LLMs via the unified OpenRouter API in Python 3.11.5.

4.2 Baseline

As a baseline for our experiments, we use the currently best-known results for BLRs from literature [2, 9, 10]. Since several best-known results were computed more than a decade ago, we augment the baseline with results obtained by running Gurobi for 4 hours. We run Gurobi on two variants of the model, one with the symmetry-breaking constraints used by Díaz et al. [2] and one without. These symmetry-breaking constraints enforce: $\text{pos}(s,1) = s$ for $1 \leq s \leq n$ and $\text{rect}(i,1) < \text{rect}(i+1,1)$ for $1 \leq i < k$. Listing 2 shows the MiniZinc code for these constraints. We use the results from Gurobi for instances where it can improve the previous best result or where no results are found in the literature (this includes most of the instances with $n = 13$). The baseline imbalance values are shown in Table 2. In our experiments, we only consider the instances that are yet to be solved optimally, which includes most with $k > 2$ and $n > 7$.

Listing 2 Known Symmetry-Breaking Constraints.

```
constraint forall(col in 1..n) (invrect[1,col]=col);
constraint forall(row in 1..k where row > 1) (
  rectangle[row-1,1]< rectangle[row,1]
);
```

Table 2 Baseline results for our experiments. The gray cells are already known to be optimal. The blue cells indicate instances for which either Gurobi outperforms the known results from literature or no previous results exist. The light blue cells indicate instances where the improvement came from Gurobi without symmetry-breaking, while dark blue cells indicate instances where the improvement comes from Gurobi with symmetry-breaking.

$n\backslash k$	2	3	4	5	6	7	8	9	10	11	12
8	40.0	36.0	32.0	30.0	24.0	28.0	0.0				
9	65.3	56.0	56.0	54.0	48.0	64.6	58.6	0.0			
10	92.0	82.0	90.6	66.6	82.0	76.0	95.3	80.0	40.0		
11	124.0	118.0	118.0	118.0	124.0	120.0	128.0	128.0	110.0	0.0	
12	168.0	158.0	162.6	170.6	120.0	165.3	184.6	178.0	174.6	147.3	0.0
13	218.6	210.0	228.6	224.0	234.0	222.6	273.3	248.0	288.6	278.6	352.0

4.3 Incremental Warmstart Approach

We run the incremental warmstart approach described in Section 3.2 twice to see how much the randomness of LLMs influences the experiment's outcome. Overall, the results are fairly similar between the two runs. For about three-quarters of the instances, the results differ by less than 10%. Three outliers differ by more than 20%, with the highest difference being 34%. Of all the LLM responses, around 14% had syntax errors in the produced json or minizinc code and were simply discarded. For each instance, the result of the better run is shown in Table 3. Overall, 24 out of 44 instances show an improvement compared to the baseline.

Table 3 Results of the incremental warmstart approach. The top entry within each cell shows the baseline result (from Table 2), while the bottom entry shows the result from the incremental warmstart approach for each instance. The highlighted cells indicate that the incremental warmstart approach outperforms the baseline.

$n\backslash k$	3	4	5	6	7	8	9	10	11	12
8			30.0	24.0	28.0					
			32.0	24.0	34.0					
9	56.0	56.0	54.0	48.0	64.6	58.6				
	56.0	56.0	52.6	52.0	57.3	60.0				
10	82.0	90.6	66.6	82.0	76.0	95.3	80.0	40.0		
	82.0	84.0	82.6	80.0	82.0	88.0	70.0	76.0		
11	118.0	118.0	118.0	124.0	120.0	128.0	128.0	110.0		
	114.0	120.0	116.0	116.0	122.0	118.0	124.0	118.0		
12	158.0	162.6	170.6	120.0	165.3	184.6	178.0	174.6	147.3	
	158.0	165.3	168.6	152.0	164.0	148.0	170.0	174.6	220.6	
13	210.0	228.6	224.0	234.0	222.6	273.3	248.0	288.6	278.6	352.0
	198.0	214.0	210.6	214.0	212.0	220.6	252.0	242.0	252.0	294.0

Analyzing the different prompts shows that `prompt_basic`, `prompt_description`, and `prompt_combinations` are respectively responsible for generating 37.8%, 36.9%, and 25.3% of all streamliners that lead to an improvement. This shows that providing a detailed problem description does not influence the outcome of the LLM a lot. Further, it shows that asking for combinations of streamliners also does not lead to significantly better results. This might be because there is a higher likelihood that streamliners hinder each other instead of combining their strengths, and the LLM is not fully capable of finding the mutually compatible combinations. When comparing the five different LLMs, the fraction of improvements they contribute are as follows: GPT-o3: 22.6%, GPT-4o: 22.3%, Gemini: 20.7%, Claude: 20.2%, Deepseek R1: 14.2%.

Figure 5 shows the change of imbalance over time for seven curated, representative instances. The initial result that is found after 1 minute is about 25% to 60% worse than the baseline. In the beginning, new improvements are found quite quickly; later, however, it flattens out. After about 2 hours of incremental warmstart, many of the instances can already outperform the baseline. Interestingly, if we ignore the previously known results for the BLR problem and only compare against the results from the 4-hour Gurobi runs, our incremental warmstart approach performs better on 40 of 44 instances.

4.4 Evolutionary Approach

We run the evolutionary approach and the exploration-only approach for 5 hours each. At the end, we evaluate the three best-performing streamliners from each by running them for 4 hours and then report the best result in Table 4. The evolutionary approach could improve 24 out of 44 instances with respect to the baseline, while the exploration-only approach improved 21 instances. For 6 instances, the evolutionary approach failed to assemble the original population of 20 streamliners and thus terminated before starting the evolution phase. In those cases, the result is the same as the result for the exploration-only approach. Overall, the evolutionary approach performs slightly better than the exploration-only approach.

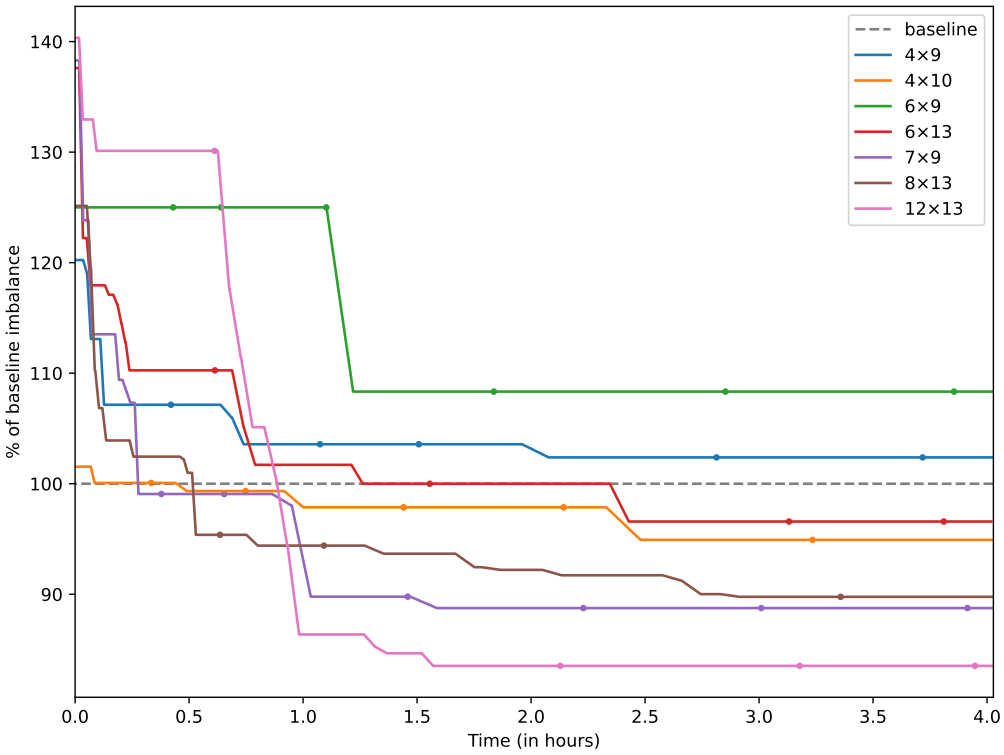


Figure 5 Change of imbalance with time for the incremental warmstart approach shown as percentage of baseline imbalance. The dashed line at 100% denotes the baseline imbalance value, and the dots on each line indicate when the training time is increased.

Another striking observation is that some of the streamliners found by our approach perform so well that running the streamlined model for 3 minutes yielded better results than running the unstreamlined model for 4 hours. Running the streamliners found in the evolutionary process for 4 further hours improved the imbalance by 10% on average compared to the result from initial 3-minute run.

Comparing the five different LLM models shows that GPT-o3, GPT-4o, Deepseek R1, Claude, and Gemini respectively found 26.0%, 25.2%, 17.9%, 17.5%, and 13.4% of the three best streamliners across all instances and variants. It is also worth noting that there are only two streamliners that appear in the list of top three streamliners for more than two instances. This demonstrates the great diversity of LLM-generated streamliners.

We showcase some interesting streamliners that were generated by our evolutionary approach in Listing 3 and provide a brief explanation of each of them below:

A This streamliner is generated for the instances 8×10, 8×11, and 3×9. It enforces each row to be strictly lexicographically smaller than the next row. However, since every entry within a row must be different, this constraint amounts to enforcing lexicographically ascending entries in the first cell of each row. Thus, the LLM rediscovered the already known symmetry-breaking constraint from literature. Interestingly, the formulation found by the LLM performed a bit better on the corresponding instances than the original formulation by Díaz et al. [2].

B This symmetry-breaking constraint works similarly to the one above. It fixes the order of the rows according to the position of the symbol 1 in each row. Instead of the values of the variable `rect`, the values of the variable `pos` are increasing with every row. This

Listing 3 Selected Streamliners.

```
A: constraint forall(r in 1..k-1) (
     lex_less(
       [rect[r,c] | c in 1..n],
       [rect[r+1,c] | c in 1..n]
     )
   );

B: constraint forall(row in 1..k-1) (
     pos[row,1] <= pos[row+1,1]
   );

C: constraint forall(row in 1..k, col in 1..(n div 2)) (
     rect[row, col] + rect[row, n-col+1] = n+1
   );

D: constraint forall(i in 1..n, j in 1..n where i < j) (
     dist[i,j] != 0
   );

E: include "global_cardinality.mzn";
   array[1..n] of var 0..k: val_counts;
   constraint global_cardinality(
     [pos[i,j] | i in 1..k, j in 1..n],
     [j | j in 1..n],
     val_counts
   );
   constraint forall(i in 1..n) (val_counts[i] >= k-1);
   constraint forall(i in 1..n, j in 1..n where i<j) (
     imbalances[i,j] <= max(k*(n+1)-3,2*k*(n-2)) div 2
   );
```

Table 4 Results of the exploration-only and evolutionary approaches. Within each cell, the top entry within each cell shows the baseline result, the middle entry shows the result from the exploration-only approach, and the bottom entry shows the result from the evolutionary approach. Green-colored cells indicate that the corresponding variant outperformed the baseline. For those instances where the baseline could be improved upon, bold text indicates which variant performed better

$n\backslash k$	3	4	5	6	7	8	9	10	11	12
8			30.0	24.0	28.0					
			32.0	28.0	28.0					
			32.0	28.0	28.0					
9	56.0	56.0	54.0	48.0	64.6	58.6				
	56.0	56.0	54.6	52.0	**55.3**	60.0				
	56.0	56.0	54.6	52.0	56.0	60.0				
10	82.0	90.6	66.6	82.0	76.0	95.3	80.0	40.0		
	80.0	**84.6**	82.0	**76.0**	78.6	**87.3**	92.0	77.3		
	80.0	**84.6**	82.0	78.0	76.0	89.3	94.0	88.0		
11	118.0	118.0	118.0	124.0	120.0	128.0	128.0	110.0		
	116.0	**112.0**	118.0	118.0	122.0	**124.0**	136.0	138.0		
	116.0	**112.0**	**116.0**	**112.0**	130.0	126.0	132.0	144.0		
12	158.0	162.6	170.6	120.0	165.3	184.6	178.0	174.6	147.3	
	160.0	**154.6**	169.3	160.0	174.0	182.0	**160.0**	186.0	204.0	
	156.0	**154.6**	**145.3**	120.0	174.0	**177.3**	176.0	199.3	219.3	
13	210.0	228.6	224.0	234.0	222.6	273.3	248.0	288.6	278.6	352.0
	204.0	220.0	219.3	**204.0**	**218.6**	**227.3**	270.0	**252.6**	283.3	332.0
	204.0	**213.3**	**216.6**	220.0	227.3	**227.3**	**228.0**	**252.6**	**224.0**	**314.0**

constraint is generated for the instances 5×8, 8×10. Further, this constraint was also part of a combination of constraints for three instances, namely, 5×11, 5×8, and 6×11.

C This streamliner is decisive to achieve the improved imbalance result for the 4×12 instance. It enforces that for every row, cells that are horizontally mirrored about the center column, add up to $n + 1$. The resulting Latin rectangle can be found in Figure 6c.

D This constraint enforces that for every pair of symbols, the distance must not be 0. Since all elements in a row are anyway defined to be different, this is an implied constraint. Nonetheless, it helped the solver find a 4×11 Latin rectangle with better imbalance.

E This combination of constraints ensures that each value $i \in [1, n]$ appears at least $k - 1$ times in each column of `pos` and that each pairwise imbalance is less than or equal to half of the maximum possible imbalance. It showcases one of the most complex combinations of constraints we obtained from the LLMs. It is not only able to combine multiple constraints but also add an `include` statement and introduce a new array of decision variables. This code is generated for the 6×8 instance. It outperforms the results from the 4-hour Gurobi runs but does not manage to beat the previously known best result.

4.5 Combinations of Approaches

Our incremental warmstart approach can not only be started from scratch but also from an already known solution. Hence, with an aim of finding the best Latin rectangles without concern for fair comparison, we run the incremental warmstart approach using the results from the evolutionary approach and with some previously published results from literature.

Table 1 summarizes the final best results we found from all approaches considered. Overall, we can improve 32 out of 44 instances. The list below shows which improving result is found by which approach. If two approaches arrive at the same result for a particular instance, then it is included in both approaches.

- **incremental warmstart**: 5×9, 9×10, 3×11, 5×11, 7×12, 8×12, 3×13, 5×13, 8×13
- **exploration-only**: 3×10, 4×11, 4×12, 6×13
- **evolution**: 3×10, 4×11, 5×11, 4×12, 5×12, 11×13
- **combination of approaches**: 7×9, 8×9, 4×10, 6×10, 7×10, 8×10, 6×11, 7×11, 8×11, 9×11, 3×12, 9×12, 4×13, 7×13, 9×13, 10×13, 12×13

In Figure 6 we present some of the Latin rectangles generated by our approach which significantly improved upon previously known results (by at least 14%) as well the 4×12 Latin rectangle for illustration purposes.

5 Discussion and Conclusion

In this paper, we present two strategies for using LLM-generated streamlining constraints for the optimization problem BLR, namely the incremental warmstart approach and the evolutionary approach. While the incremental warmstart approach uses streamliners directly on the last-found best solution, the evolutionary approach discovers a high variety of new promising streamliners. Both approaches show strong potential. An interesting finding is that, when running an optimization problem, the biggest improvements are often found in the beginning. Our approach exploits this by running many streamlined versions of the original encoding for only a few minutes, and thus can select good streamliners quickly and efficiently. Thereby, we successfully improve the upper bounds for many instances of the BLR problem, outperforming state-of-the-art methods in 32 out of 44 instances. For the other instances, it is important to consider that some of the previously known results might already be optimal. This demonstrates the potential of using LLMs to generate structural constraints that significantly enhance the solver performance.

It would be interesting to see whether it works just as effectively for other problems, particularly novel problems that are unlikely to be in the training corpus of LLMs. Concerning this, our approach has the advantage that it is very flexible. One can easily adapt it to other optimization problems and other solvers or use different LLMs. The only limitation is that it must be possible to find the first feasible solution rather quickly. However, it is also crucial to note that our method cannot prove optimality. Although we may not always find the optimal solution, our approach often finds better solutions in shorter time frames. This trade-off between theoretical guarantees and practical performance is acceptable in many real-world applications, particularly when improved solutions are more valuable than guarantees.

Looking ahead, we see several potential avenues. One particularly promising vision is that future constraint solvers might integrate our incremental warmstart approach with LLM-generated streamliners in their solving procedure. Performance improvements could be substantial. In summary, LLM-generated streamlining constraints offer a practical and powerful way to enhance the solving performance of optimization problems. Our method not only contributes new best results for the BLR problem, but also opens the door for more intelligent and adaptive solving frameworks in the future.

1	2	4	5	3	6	7	9	8	10
3	4	9	7	2	8	1	10	6	5
8	7	2	6	4	10	9	1	5	3
7	1	10	3	8	5	4	2	9	6
9	10	5	2	7	3	8	6	1	4
4	5	7	8	9	1	6	3	10	2

(a) 6×10 instance with imbalance 70.0.

8	9	5	4	6	1	2	3	7
7	4	2	6	9	3	5	1	8
5	2	4	1	7	8	9	6	3
3	5	8	7	4	6	1	2	9
4	1	3	2	8	9	7	5	6
1	6	7	8	5	2	3	9	4
9	7	1	5	3	4	6	8	2

(b) 7×9 instance with imbalance 51.3.

5	7	2	3	1	4	9	12	10	11	6	8
9	1	11	7	8	3	10	5	6	2	12	4
6	2	1	9	3	8	5	10	4	12	11	7
3	11	5	6	4	1	12	9	7	8	2	10

(c) 4×12 instance with imbalance 154.6.

5	11	3	6	4	1	12	9	7	10	2	8
7	3	1	8	11	4	9	2	5	12	10	6
1	4	10	5	7	2	11	6	8	3	9	12
9	10	11	1	6	8	5	7	12	2	3	4
2	1	6	3	9	5	8	4	10	7	12	11

(d) 5×12 instance with imbalance 145.3.

1	8	9	4	11	10	6	5	12	2	3	7
12	1	6	2	9	3	11	10	7	4	8	5
3	10	4	1	2	5	8	7	6	9	12	11
11	7	1	3	5	9	4	12	10	6	2	8
5	12	8	6	3	11	10	1	2	7	4	9
7	6	10	12	1	4	5	9	3	8	11	2
4	2	11	5	6	7	1	8	9	12	10	3
6	4	3	11	12	8	7	2	1	5	9	10

(e) 8×12 instance with imbalance 148.0.

1	11	4	8	2	7	12	13	5	6	10	9	3
8	2	10	11	6	13	7	3	1	9	5	12	4
10	12	13	2	1	3	9	8	6	4	11	5	7
6	1	7	10	9	4	2	5	8	12	13	3	11
9	13	8	4	11	5	10	6	12	2	3	7	1
7	8	12	5	10	9	1	11	4	3	6	13	2
2	9	3	7	4	10	11	12	13	1	8	6	5
12	6	11	9	7	2	8	1	3	5	4	10	13

(f) 8×13 instance with imbalance 220.6.

11	12	2	3	7	10	5	6	1	8	4	13	9
6	7	13	5	8	9	2	4	12	10	11	1	3
13	10	7	4	12	2	8	3	6	5	9	11	1
4	5	11	13	3	8	7	12	9	1	2	6	10
8	3	6	7	4	1	11	9	2	13	10	12	5
10	9	3	12	5	6	4	1	8	7	13	2	11
12	6	4	1	2	5	13	11	10	3	8	9	7
2	4	9	8	6	11	10	5	7	12	1	3	13
5	8	1	10	11	13	6	2	3	9	7	4	12
7	1	5	2	9	3	12	13	4	11	6	10	8
9	11	12	6	13	7	1	10	5	4	3	8	2

(g) 11×13 instance with imbalance 224.0.

1	2	11	10	12	6	5	13	9	4	3	7	8
13	11	3	6	8	5	12	9	10	2	1	4	7
9	10	4	11	3	8	13	1	12	6	7	2	5
3	6	1	12	9	10	7	4	13	8	5	11	2
11	12	8	7	1	3	10	2	5	9	13	6	4
4	8	6	2	10	13	1	5	3	7	12	9	11
6	7	5	9	11	2	8	3	4	12	10	1	13
7	13	10	8	6	9	2	11	1	5	4	3	12
10	5	7	4	13	12	11	6	2	3	9	8	1
5	9	13	1	4	11	3	8	7	10	2	12	6
8	4	12	5	2	1	9	7	6	13	11	10	3
2	3	9	13	7	4	6	12	11	1	8	5	10

(h) 12×13 instance with imbalance 274.0.

Figure 6 Examples of Latin rectangles generated by our approach that improve upon the previous best imbalance values.

References

1 Md. Masbaul Alam, M. A. Hakim Newton, and Abdul Sattar. Constraint-based search for optimal Golomb rulers. *J. Heuristics*, 23(6):501–532, 2017. doi:10.1007/s10732-017-9353-x.

2 Mateo Díaz, Ronan Le Bras, and Carla P. Gomes. In search of balance: The challenge of generating balanced Latin rectangles. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, volume 10335 of *Lecture Notes in Computer Science*, pages 68–76. Springer, 2017. doi:10.1007/978-3-319-59776-8_6.

3 Carla P. Gomes and Meinolf Sellmann. Streamlined constraint reasoning. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2004. doi:10.1007/978-3-540-30201-8_22.

4 Aditya Grover, Tudor Achim, and Stefano Ermon. Streamlining variational inference for constraint satisfaction problems. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 10579–10589, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/02ed812220b0705fabb868ddbf17ea20-Abstract.html.

5 Marijn J. H. Heule, Manuel Kauers, and Martina Seidl. Local search for fast matrix multiplication. 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 155–163. Springer, 2019. doi:10.1007/978-3-030-24258-9_10.

6 Marcus Jones, Richard Woodward, and Jerry Stoller. Increasing precision in agronomic field trials using Latin square designs. *Agronomy Journal*, 107(1):20–24, 2015. doi:10.2134/agronj14.0232.

7 Ronan LeBras, Carla P. Gomes, and Bart Selman. From streamlined combinatorial search to efficient constructive procedures. 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 499–506. AAAI Press, 2012. doi:10.1609/aaai.v26i1.8147.

8 Zhenjun Liu, Leroy Chew, and Marijn J. H. Heule. Avoiding monochromatic rectangles using shift patterns. In Hang Ma and Ivan Serina, editors, *Proceedings of the Fourteenth International Symposium on Combinatorial Search, SOCS 2021, Virtual Conference [Jinan, China], July 26-30, 2021*, pages 225–227. AAAI Press, 2021. doi:10.1609/socs.v12i1.18591.

9 Renee Mirka, Laura Greenstreet, Marc Grimson, and Carla P. Gomes. A new approach to finding 2 x n partially spatially balanced Latin rectangles (short paper). 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 47:1–47:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CP.2023.47.

10 Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. Proven optimally-balanced Latin rectangles with SAT (short paper). 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 48:1–48:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CP.2023.48.

11 Casey Smith, Carla P. Gomes, and Cèsar Fernández. Streamlining local search for spatially balanced Latin squares. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1539–1540. Professional Book Center, 2005. URL: http://ijcai.org/Proceedings/05/Papers/post-0460.pdf.

12 Patrick Spracklen, Özgür Akgün, and Ian Miguel. Automatic generation and selection of streamlined constraint models via Monte Carlo search on a model lattice. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 362–372. Springer, 2018. doi:10.1007/978-3-319-98334-9_24.

13 Patrick Spracklen, Nguyen Dang, Özgür Akgün, and Ian Miguel. Automatic streamlining for constrained optimisation. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 of *Lecture Notes in Computer Science*, pages 366–383. Springer, 2019. doi:10.1007/978-3-030-30048-7_22.

14 Patrick Spracklen, Nguyen Dang, Özgür Akgün, and Ian Miguel. Towards portfolios of streamlined constraint models: A case study with the Balanced Academic Curriculum Problem. *CoRR*, abs/2009.10152, 2020. doi:10.48550/arXiv.2009.10152.

15 Patrick Spracklen, Nguyen Dang, Özgür Akgün, and Ian Miguel. Automated streamliner portfolios for constraint satisfaction problems. *Artificial Intelligence*, 319:103915, 2023. doi:10.1016/J.ARTINT.2023.103915.

16 Nseobong Peter Uto and RA Bailey. Balanced semi-Latin rectangles: properties, existence and constructions for block size two. *Journal of Statistical Theory and Practice*, 14(3):51, 2020.

17 H.M. van Es and C.L. van Es. Spatial nature of randomization and its effect on the outcome of field experiments. *Agronomy Journal*, 85(2):420–428, 1993.

18 Florentina Voboril, Vaidyanathan Peruvemba Ramaswamy, and Stefan Szeider. Stream-LLM: 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, May 2025. IEEE Computer Soc. URL: https://doi.ieeecomputersociety.org/10.1109/NSE66660.2025.00010, doi:10.1109/NSE66660.2025.00010.

19 Florentina Voboril, Vaidyanathan Peruvemba Ramaswamy, and Stefan Szeider. Supplementary material for paper - Balancing Latin Rectangles with LLM-generated Streamliners, June 2025. doi:10.5281/zenodo.15616074.

20 James Wetter, Özgür Akgün, and Ian Miguel. Automatically generating streamlined constraint models with Essence and Conjure. 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 480–496. Springer, 2015. doi:10.1007/978-3-319-23219-5_34.

Multi-League Sports Scheduling with Team Interdependencies: An Optimization Model

Nils Weidmann
Westdeutscher Tischtennis Verband e. V., Duisburg, Germany

Abstract

Every year, a large number of matches must be scheduled for professional and amateur sports teams. Several constraints have to be considered, including the overall capacity of venues and interdependencies between teams of the same club. As interdependent teams of a club play in different leagues, finding an optimal solution is very challenging for practitioners. While the problem of respecting capacity restrictions is well-addressed in prior work, interdependencies between teams are widely neglected, despite being a problem of major importance in practice. This paper enhances the formal definition of the multi-league-sports scheduling problem to take team interdependencies into account. We create an optimization problem to be solved by means of integer linear programming, and prove the corresponding decision problem to be $\mathcal{NP}$-complete by a polynomial reduction from 3-SAT. An implementation which was used to schedule German table tennis leagues of a certain district demonstrates the practical applicability of the approach.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Mathematics of computing → Combinatorial optimization

Keywords and phrases sports scheduling, linear optimization, constraint programming

Digital Object Identifier 10.4230/LIPIcs.CP.2025.37

Supplementary Material

Software (source code): `https://github.com/NilsWeidmann/KeyGenerator` [35]
archived at `swh:1:dir:c668e66011bff623318334bb60799385ccb4357e`

Dataset: `https://github.com/NilsWeidmann/Multi-League-Sports-Scheduling` [36]
archived at `swh:1:dir:6b32a179b07ef602689a3c799174f938fe11b379`

1 Introduction and Motivation

In several disciplines, sports teams play each other twice a season – once at home and once away – which is called a double round robin tournament (DRR). Usually, schedules for DRRs are created prior to the season, and define in which rounds (i.e., in which weeks) teams play each other. To assure fairness among teams, these schedules should be compact, such that teams play each week [4]. According to their skill, age and gender, teams are grouped into different leagues. Especially in amateur and youth sports, teams of a league are further subdivided into divisions according to regional aspects [28, 33].

Scheduling such divisions gets complicated when additional constraints have to be respected. When two teams share a venue, e.g., their home matches cannot take place in the same week. Prominent examples include AC and Inter Milan (football), New York Giants and Jets (American football) and Los Angeles Lakers and Sparks (basketball).

While creating schedules for a few professional leagues might be already challenging [1, 12], the problem gets more severe when scheduling hundreds of amateur divisions, where it is common practice that multiple teams of the same club share a common infrastructure that underlies capacity constraints. This scenario is denoted as the multi-league sports scheduling problem (MLSSP) in the following. As the number of matches which can be hosted by a club at the same time is limited, and teams that share a venue usually play in different divisions, these divisions cannot be scheduled independently.

© Nils Weidmann;
licensed under Creative Commons License CC-BY 4.0

Apart from overall capacity constraints, there are also other interdependencies between teams of a club that ensure that they play at home in complementary weeks, or in the same weeks on different days, to ensure that players can attend matches of both teams [30]. Furthermore, schedule creation is usually a distributed task: Schedulers are responsible for all divisions of a specific region allotted to them [7], which makes it necessary to respect scheduling decisions that were made on super-ordinate (e.g., national) level.

1.1 Related Work

Davari et al. propose an algorithm that minimizes the number of capacity violations of a schedule, which does not respect team interdependencies, though [7]. For the special case of all teams of a club playing at home in the same weeks without allowing capacity constraint violations, they prove this version of the problem to be $\mathcal{NP}$-complete via a polynomial reduction from vertex coloring. Li et al. extend this work by a computational study [22], a construction and improvement heuristic that minimizes constraint violations, and a multi-objective approach minimizing both traveling distances and constraint violations [23].

Other approaches rather aim at creating an optimal schedule for just a few divisions. Schönberger solves the scheduling problem on the level of single matches by means of mixed-integer programming (MIP), focusing on the optimization of player substitution opportunities [30, 31]. A computational study shows that solving the MIP involves large efforts even for small instances. Creating optimal schedules for single (professional) leagues is a well-addressed topic in prior work [2, 9, 11, 19, 24, 34]. Further research concentrates on scheduling sports competitions based on other objectives, such as balancing and minimizing breaks [3,8,13,14,18,25,26], traveling distances [10,15,16,29], required venues [32], scheduling matches in specific rounds [5], or introducing additional matches [6, 20].

To the best of our knowledge, none of the existing approaches aims at scheduling a large number of interdependent divisions, respecting both capacity constraints, team interdependencies and constraints from super-ordinate schedules.

1.2 Contribution and Paper Organization

In this paper, we address the identified research gap by extending the definition of the MLSSP to take team interdependency constraints (which also have implications on the venue utilization) and constraints from super-ordinate schedules into account. The optimization goal is to minimize the overall number of deviations in actual and required schedules, whereby those are guaranteed to be at least similar for all teams. Thereby, the problem definition comes closer to the requirements of sports scheduling in practice. We prove the extended MLSSP to be $\mathcal{NP}$-complete by a polynomial reduction from 3-SAT, and underpin the practical applicability of our approach by scheduling German table tennis leagues of a certain district. Further experiments on generated data demonstrates the scalability of our approach.

The remainder of this paper is organized as follows: An overview of the problem domain and the proposed solution approach is provided in Sect. 2. The MLSSP is formally defined in Sect. 3, before an integer linear program (ILP) is constructed based on this definition in Sect. 4. The decision problem corresponding to MLSSP is proven to be $\mathcal{NP}$-complete in Sect. 5, and a practical implementation is sketched in Sect. 6. We apply our implementation to a real-world case study and evaluate its runtime performance in Sect. 7, before Sect. 8 summarizes the main results and sketches promising directions for future research.

2 Background and Solution Approach

Every year, schedules for thousands of (amateur) sports leagues have to be created, whereby matches between two **teams** take place at either of the teams' venues. Typically, teams of the same **club** share a common infrastructure (venues, material, etc.). The number of simultaneous matches is thus restricted by the availability of the shared infrastructure. Oftentimes, teams playing home on different days must be considered separately (e.g., those playing home on Saturdays and Sundays), turning capacity constraints into **team interdependencies**.

It is desirable to find a schedule that respects those dependency constraints to the largest possible extent. To simplify the scheduling process, matches are not scheduled individually, but on the basis of **grids** that define the order of matches throughout the season. Matches are expressed in terms of **key** pairs, whereby the keys serve as placeholders for the teams of a **division**. A concrete example illustrating the basic terminology is presented in Sect. 3.1.

The use of uniform grids for each division makes it possible to express dependency constraints involving teams playing in different divisions, as well as synchronizing schedules for leagues of different levels (e.g., national, federal state and regional level). For instance, assigning the same key to teams in different divisions ensures that they play home in **parallel**, whereas other key combinations guarantee that two teams play home in **opposite** weeks.

The overall process is depicted in Fig. 1 in form of a unified modeling language (UML) activity diagram, and comprises the following four steps:

(1) Before the schedule creation process starts, teams of a league must be grouped into divisions. **Club administrators** define interdependencies between teams of their clubs based on the availability of venues and/or tactical reasons via an internet portal. Regular match days and starting times for home matches are also defined in this step, which will come into play in step (4) when generating the match schedules.

(2) A **scheduler** assigns keys to teams in a way that dependency constraints are respected to the largest possible extent. Note that keys for clubs of which at least one team plays on a super-ordinate level are provided as additional input for the key determination process, causing a loop in the data flow.

(3) Conflict situations cannot be avoided completely in practice, as teams of neighboring clubs play against each other in multiple divisions. Thus, the scheduler resolves conflicts by assigning different, but **similar** keys to some teams, balancing interests among clubs.

(4) Finally, the scheduler enters the generated keys into the internet portal, such that match schedules can be generated based on predefined uniform grids and the regular match days and starting times as defined in step (1).

Step (2), i.e., the determination of keys for schedule creation, is a hard task to solve: The keys to be assigned to teams of the same club must be aligned, while those teams are spread over multiple divisions, and each key can only be assigned once per division. In practice, due to the lack of proper conceptual and technical solutions, this process is done manually. Schedulers try to overcome the inherent complexity by applying a divide-and-conquer approach, leading to low-quality solutions [22]. This is problematic for several reasons: First, schedulers put large efforts in finding a barely acceptable solution, whereby efforts increase disproportionately with the number of divisions to schedule [17]. Second, clubs thereupon have to eliminate the effects of bad schedules, e.g., by postponing matches or renting further venues, resulting in additional efforts on their side.

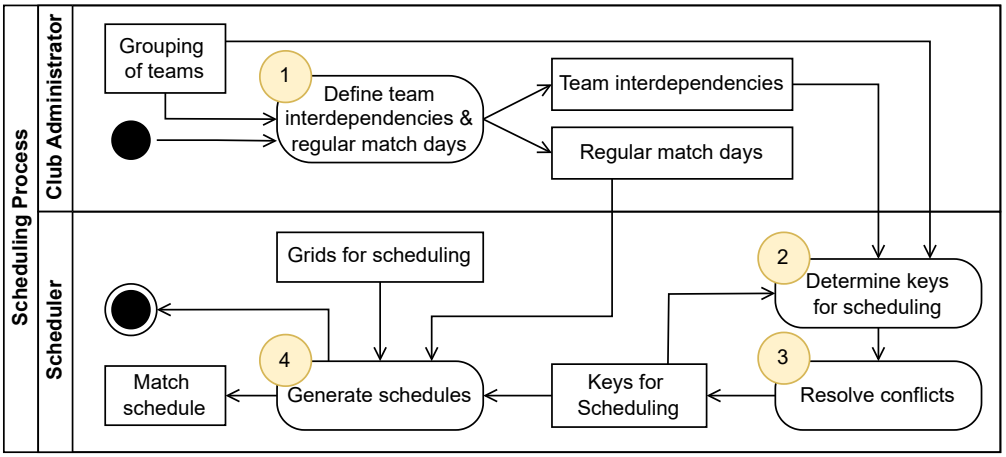


Figure 1 Process of match scheduling in double-round-robin tournaments

To overcome these hurdles, we propose to formalize the task of sports schedule creation as a combinatorial optimization problem. Instead of manually determining suitable keys, the scheduler transfers the dependency constraints of all clubs allotted to their region to a **key generator**. This software tool shall pursue the optimization goal of minimizing the number of conflict situations, i.e., situations in which the key assigned to a team differs from the required key. The advantage of modeling the problem at hand in terms of combinatorial optimization is that established SAT or ILP solvers can be integrated into the key generator, promising high performance and reliability. The final decision of how to resolve conflicts should be left to the scheduler, though, to balance the interests of the involved stakeholders. As a result, high-quality schedules can be created with substantially reduced efforts for both schedulers and club administrators.

3 Problem Definition

This section introduces the general setting and terminology formally, before team interdependencies are specified. Finally, scheduling constraints for interdependent divisions are defined, resulting in a linear optimization problem.

3.1 General Setting and Terminology

The basic concepts of the problem definition comprise clubs, teams, and divisions. Each team belongs to exactly one club and one division.

▶ **Definition 1** (Club, Team, Division). *Let $\mathcal{C}$ be a set of clubs and $\mathcal{D}$ be a set of divisions. Let $\mathcal{T}_C, C \in \mathcal{C}$ denote the set of teams of each club, and let $\mathcal{T}_D, D \in \mathcal{D}$ denote the set of teams of each division. Let $\mathcal{T} = \bigcup_{C \in \mathcal{C}} \mathcal{T}_C = \bigcup_{D \in \mathcal{D}} \mathcal{T}_D$ denote the set of all teams. Let $c : \mathcal{T} \to \mathcal{C}$, $d : \mathcal{T} \to \mathcal{D}$ be functions that map teams to their respective club and division.*

An example instance with 18 teams allocated to three divisions is shown in Tab. 1 (ignoring the columns "Key" and "WS" for the moment). Each team is represented by the name of its club, while most clubs have teams in multiple divisions. It is also possible that multiple teams of a club play in one division, such as two teams of "Austin" in Men's division.

DRRs comprise multiple rounds, which are usually scheduled in consecutive weeks. To create schedules, **grids** are used that determine which matches are scheduled in which week.

Table 1 Example: Three divisions to be scheduled

Women's division			Men's division			Youth division		
Key	Team	WS	Key	Team	WS	Key	Team	WS
1	Austin	$\check{W}_1$	4	Austin	$\hat{W}_1$	1	Austin	$\check{W}_1$
2	Berkeley	$\check{W}_2$	2	Berkeley	$\check{W}_2$	3	Chicago	$\hat{W}_1$
6	Chicago	$\check{W}_1$	1	Detroit	W_0	2	Detroit	$\hat{W}_1$
5	Detroit	$\check{W}_1$	6	Green Bay	$\hat{W}_2$	5	El Paso	$\check{W}_1$
4	El Paso	$\check{W}_2$	5	Houston	W_0	6	Green Bay	$\hat{W}_2$
3	Fort Worth	W_0	3	Austin II	$\hat{W}_1$	4	Houston	W_0

▶ **Definition 2** (Grid). *Let $\mathcal{G}$ be a set of grids, with $|G| \in \{2n : n \in \mathbb{N}\}$ being the size of a grid $G \in \mathcal{G}$. Let $g : \mathcal{D} \to \mathcal{G}$ be a function that maps divisions to their grids. It holds that $\forall\ D \in \mathcal{D}\ (|\mathcal{T}_D| \leq |g(D)|)$, i.e., the number of teams in a division must be less or equal to the grid size of the division.*

An example grid for a division of up to six teams is depicted in Tab. 2. It can be read as follows: *1 - 6* in the first row of week W1 means that team 1 plays team 6 at home. The matches in W6 - W10 correspond to those in W1 - W5, swapping the home and away teams.

Table 2 Grid for divisions of a DRR with six teams

W1	W2	W3	W4	W5	W6	W7	W8	W9	W10
1 - 6	6 - 4	2 - 6	6 - 5	3 - 6	6 - 1	4 - 6	6 - 2	5 - 6	6 - 3
2 - 5	5 - 3	3 - 1	1 - 4	4 - 2	5 - 2	3 - 5	1 - 3	4 - 1	2 - 4
3 - 4	1 - 2	4 - 5	2 - 3	5 - 1	4 - 3	2 - 1	5 - 4	3 - 2	1 - 5

The numbers in a grid serve as placeholders for the teams of a division. In the following, these numbers are denoted as **keys** (an example allocation of teams to keys can be found in the respective columns of Tab. 1):

▶ **Definition 3** (Key). *Let $\mathcal{K}_G = \{1, \ldots, |G|\}$ be a set of keys for a grid $G \in \mathcal{G}$.*

3.2 Team Interdependencies

To ensure that venues can be shared between two teams, it is important that they do not play at home in the same week. Grids have **opposite** pairs of **keys**, e.g., 1/4, 2/5 and 3/6 in the example of Tab. 2. By choosing the keys for interdependent teams accordingly, it can be assured that at most one team plays at home each week.

▶ **Definition 4** (Opposite Keys). *Let $o_G : \mathcal{K}_G \to \mathcal{K}_G, G \in \mathcal{G}$ be a total function that maps keys of a grid G to their opposite. It holds:*

$$\forall\ G \in \mathcal{G}, \forall\ K \in \mathcal{K}_G\ (o_G(o_G(K)) = K) \tag{1}$$

Using such standardized grids, team interdependencies can be defined across multiple divisions. For grids of different sizes (which are necessary for divisions with different numbers of teams), there must be a notion of which keys correspond to each other, i.e., for which keys home games are scheduled in the same weeks. Keys fulfilling this property are denoted as **parallel keys**:

▶ **Definition 5** (Parallel Keys). *Let $P_{G_1,G_2} \subseteq \mathcal{K}_{G_1} \times K_{G_2}, G_1, G_2 \in \mathcal{G}$ define a relation for parallel keys and (potentially) different grids. It holds:*

$$\begin{aligned} &\forall\ G \in \mathcal{G}, \forall\ K \in \mathcal{K}_G, ((K,K) \in P_{G,G}) \\ &\forall\ G_1, G_2 \in \mathcal{G}, \forall\ K_1 \in \mathcal{K}_{G_1}, \forall\ K_2 \in \mathcal{K}_{G_2}\ ((K_1,K_2) \in P_{G_1,G_2} \implies (K_2,K_1) \in P_{G_2,G_1}) \end{aligned} \tag{2}$$

In contrast to professional leagues, in which matches are often shown in the media, the order of matches or the allocation of matches to certain weeks is not that critical for amateur leagues. It is far more important for clubs to define which teams should have their matches at home in opposite weeks (such that they can share one venue) or in parallel weeks (such that players can attend matches of two teams, given they do not take place on the same day).

In order to express this requirement, **week schemes** are used: Teams of a club that follow the same week scheme play at home in the same weeks, whereas teams that follow an opposite week scheme play at home in opposite weeks. There is also a distinguished week scheme W_0 to express that there are no dependencies to other teams. This can happen if a club has only one team, or the capacity of its venue(s) is sufficient to host matches of all teams at the same time (without this being an explicit requirement for the club).

▶ **Definition 6** (Week Scheme). *Let $\hat{\mathcal{W}} = \bigcup_{i=1}^{n} \{\hat{W}_i\}$, and $\check{\mathcal{W}} = \bigcup_{i=1}^{n} \{\check{W}_i\}, n \in \mathbb{N}$ be two sets of week schemes. $\hat{W}_i \in \hat{\mathcal{W}}$ and $\check{W}_i \in \check{\mathcal{W}}, 1 \leq i \leq n$ are denoted as pairs of opposite week schemes. Let $\mathcal{W} = \hat{\mathcal{W}} \cup \check{W} \cup \{W_0\}$, with W_0 being a distinguished week scheme. Let $w : \mathcal{T} \to \mathcal{W}$ be a function that maps teams to week schemes. Let $\mathcal{T}_{C_W}, C \in \mathcal{C}, W \in \mathcal{W}$ be the set of teams of a club C that follow the week scheme W. It holds that*

$$\forall\ C \in \mathcal{C}(\bigcup_{W \in \mathcal{W}} \mathcal{T}_{C_W} = \mathcal{T}_C) \tag{3}$$

The "WS" column of Tab. 1 provides examples for week schemes. The women's and youth teams of Detroit follow opposite week schemes $\check{W}_1$ and $\hat{W}_1$, whereas the week scheme W_0 for the men's team indicates that there are no interdependencies to other teams. The women's and men's teams of El Paso follow independent week schemes $\check{W}_2$ and $\check{W}_1$, meaning that there are no interdependencies to each other, but potentially to other teams of the club.

Apparently, each key can only be assigned once per division (the constraint will be specified formally in Def. 12). Without weakening the requirement for opposite and parallel keys, instances of realistic size are barely feasible. A simple example can be found in the men's division of Table 1: Austin and Austin II follow the same week scheme $\hat{W}_1$, which means that they should be assigned the same key, which would directly lead to an infeasible system (which was already pointed out by Davari et al. in prior work [7]).

To overcome this non-exceptional problem, **similar keys** are defined for grids, meaning that only a few deviations from the intended sequence of home and away matches occur. In Tab. 2, one can observe that the keys 1 and 6 only differ in two weeks (W1 and W6), just as the keys 3 and 4. Therefore, keys 1/6 and 3/4, respectively, can be considered as similar keys that can be assigned in case the required key is not available.

▶ **Definition 7** (Similar Keys). *Let $S_G \subseteq {\mathcal{K}_G}^2, G \in \mathcal{G}$ define a relation for similar keys of a grid G. It holds that*

$$\begin{aligned} &\forall\ G \in \mathcal{G}, \forall\ K \in \mathcal{K}_G\ (K,K) \in S_G \\ &\forall\ G \in \mathcal{G}, \forall\ K_1, K_2 \in \mathcal{K}_G\ ((K_1,K_2) \in S_G \implies (K_2,K_1) \in S_G) \end{aligned} \tag{4}$$

3.3 Key Assignment

In this subsection, we elaborate on the assignment of keys to clubs (one key per week scheme), and based on that, the assignment of keys to teams. Before keys can be assigned to week schemes of clubs, it must be clear to which grid the keys refer. Regarding teams, the grid is predefined by the divisions the teams play in, whereas such an automatism does not exist for clubs. Therefore, a **reference grid** is defined for each pair of opposite week schemes.

▶ **Definition 8** (Reference Grid). *Let $r : \hat{\mathcal{W}} \cup \check{\mathcal{W}} \to \mathcal{G}$ be a function that assigns a **reference grid** to each pair of opposite week schemes. It holds that*

$$r(\hat{W}_i) = r(\check{W}_i), i \in \{1 \dots n\}, n \in \mathbb{N} \tag{5}$$

Based on the reference grid, keys can be assigned to week schemes of clubs according to Def. 9. Note that opposite keys must be assigned for opposite week schemes, which is formalized in Eq. 6.

▶ **Definition 9** (Key Assignment for Clubs). *Let $a_W : \mathcal{C} \to \mathcal{K}_{r(W)}, W \in \hat{\mathcal{W}} \cup \check{\mathcal{W}}$ be a total function that maps clubs to keys for a week scheme W. It holds that*

$$\forall\ C \in \mathcal{C}\ (a_{\hat{W}_i}(C) = o_{r(\hat{W}_i)}(a_{\check{W}_i}(C))), \hat{W}_i \in \hat{\mathcal{W}}_i, \check{W}_i \in \check{\mathcal{W}}_i, 1 \leq i \leq n, n \in \mathbb{N} \tag{6}$$

As mentioned in Sect. 1.1 and 1.2, the scheduling task is usually distributed between multiple schedulers for different levels (e.g., national, federal state and regional level). Despite having a potentially negative impact on the solution quality, this approach is indispensable to cope with the number of divisions that have to be scheduled, especially when schedules are created mostly manually. Usually, schedules for super-ordinate leagues are created first, and impose constraints for scheduling sub-ordinate leagues. In particular, key assignments for clubs that have teams in super-ordinate leagues are fixed in advance:

▶ **Definition 10** (Fixed Key Assignments for Clubs). *Let $\bar{\mathcal{C}} \subseteq \mathcal{C}$ be a set of clubs, for which there exists a fixed key assignment. Let $f_W : \bar{\mathcal{C}} \to \mathcal{K}_{r(W)}, W \in \hat{\mathcal{W}} \cup \check{\mathcal{W}}$ be a total function that maps these clubs to keys for a week scheme W. It holds that*

$$\forall\ C \in \bar{\mathcal{C}}, \forall\ W \in \hat{\mathcal{W}} \cup \check{\mathcal{W}}\ (a_W(C) = f_W(C)) \tag{7}$$

The assignment of keys to teams is specified in Def. 11.

▶ **Definition 11** (Key Assignment for Teams). *Let $k : \mathcal{T} \to \mathcal{K}_{g(d(T))}$ be a total function that maps teams to keys.*

In the remainder of this section, a connection will be established between the key assignments of clubs and their respective teams. As mentioned beforehand, we must also ensure that each key is unique within a division, i.e., is assigned to at most one team per division, which is specified in Def. 12.

▶ **Definition 12** (Key Uniqueness).

$$\forall\ D \in \mathcal{D}, \forall\ T_1, T_2 \in \mathcal{T}_D\ (k(T_1) = k(T_2) \implies T_1 = T_2) \tag{8}$$

A central requirement for clubs is that – while minor deviations are allowed – teams that follow the same week scheme are assigned keys in a way that they play at home in the same weeks. Therefore, the keys of clubs and their teams must be aligned with respect to the different week schemes. In particular, each team's key must be similar to its club's key of the respective week scheme, which is specified in Def. 13.

▶ **Definition 13** (Key Similarity)**.**

$$\begin{aligned}&\forall\ C \in \mathcal{C}, \forall\ W \in \hat{\mathcal{W}}\ \cup\ \check{\mathcal{W}}, \forall\ T \in \mathcal{T}_{C_W}, \exists\ K \in \mathcal{K}_{g(d(T))}\\&((k(T), K) \in S_{g(d(T))} \wedge (a_W(C), K) \in P_{r(W),g(d(T))})\end{aligned} \tag{9}$$

Using the previous sets of constraints, we define **satisfiability** for problem instances of MLSSP in Def. 14.

▶ **Definition 14** (Satisfiability)**.** *An instance m of MLSSP is satisfiable, if:*

$$\forall\ C \in \mathcal{C}, \forall\ W \in \hat{\mathcal{W}}\ \cup\ \check{\mathcal{W}}, \forall\ T \in \mathcal{T}, \exists\ a_W, \exists\ k\ ((6) \wedge (7) \wedge (8) \wedge (9)) \tag{10}$$

While it is necessary in practice to assign similar keys in case the required key is unavailable, this should be avoided whenever possible, as it can, e.g., reduce the number of capacity violations. Thus, the optimization goal is to minimize the number of teams that receive a key that is **not** parallel to the key which is assigned to their club for the week scheme the team follows. The resulting optimization problem is specified in Def. 15.

▶ **Definition 15** (MLSSP as an Optimization Problem)**.**

$$\begin{aligned}&min.\ |\{T \in \mathcal{T} \mid w(T) \neq W_0 \wedge (a_{w(T)}(c(T)), k(T)) \notin P_{r(W),g(d(T))}\}|\\&s.t.\ (10)\end{aligned} \tag{11}$$

4 Construction of the Integer Linear Program

In this section, the optimization problem defined in Sect. 3 is converted into a (binary) ILP that can be solved to optimality with a wide range of well-established solvers.

First, binary variables are associated with teams, representing the information whether a specific key is assigned to the team.

▶ **Definition 16** (Binary Variables for Teams)**.** *Let $\tau_{T,K} \in \{0,1\}, T \in \mathcal{T}, K \in \mathcal{K}_{g(d(T))}$ be binary variables. Let*

$$\tau_{T,K} = \left\{ \begin{array}{ll} 1 & \text{if } k(T) = K \\ 0 & \text{otherwise} \end{array} \right\}$$

Likewise, binary variables are associated with clubs, adding the week scheme as a further dimension.

▶ **Definition 17** (Binary Variables for Clubs)**.** *Let $\gamma_{C,W,K} \in \{0,1\}, C \in \mathcal{C}, W \in \hat{\mathcal{W}} \cup \check{\mathcal{W}}, K \in \mathcal{K}_{r(W)}$ be binary variables. Let*

$$\gamma_{C,W,K} = \left\{ \begin{array}{ll} 1 & \text{if } a_W(C) = K \\ 0 & \text{otherwise} \end{array} \right\}$$

Up to here, the information whether a team received the required or a similar key was expressed implicitly. As we want to minimize the number of teams for which this is the case, further binary variables are introduced that represent this **conflict** situation. They operate as auxiliary variables and will be essential for specifying the objective function.

▶ **Definition 18** (Binary Variables for Conflicts)**.** *Let $\xi_T \in \{0,1\}, T \in \mathcal{T}$ be binary variables.*

In the remainder of this section, the necessary constraints for a feasible solution are set up, before the objective function completes the optimization problem. We start with a set of simple but indispensable constraints that ensure that every club is assigned exactly one key per week scheme (resembling the property of a total function in Def. 9).

$$\forall\, C \in \mathcal{C}, \forall\, W \in \hat{\mathcal{W}} \cup \check{\mathcal{W}}\ (\sum_{K \in \mathcal{K}_{r(W)}} \gamma_{C,W,K} = 1) \tag{12}$$

Likewise, the same must hold for each team, as k is a total function according to Def. 11.

$$\forall\, T \in \mathcal{T}\ (\sum_{K \in \mathcal{K}_{g(d(T))}} \tau_{T,K} = 1) \tag{13}$$

Additionally, it must be ensured that each key is assigned at most once per division (cf. Def. 12 for the key uniqueness requirement).

$$\forall\, D \in \mathcal{D}, \forall\, K \in \mathcal{K}_{g(D)}\ (\sum_{T \in T_D} \tau_{T,K} \leq 1) \tag{14}$$

Another set of constraints (15) ensures that opposite keys (cf. Def. 4) are assigned to opposite week schemes (cf. Def. 6).

$$\begin{aligned} &\forall\, C \in \mathcal{C}, \forall\, K \in \mathcal{K}_{r(\hat{W}_i)}\ (\gamma_{C,\hat{W}_i,K} \leq \gamma_{C,\check{W}_i,o_{r(\hat{W}_i)}(K)}), 1 \leq i \leq n \\ &\forall\, C \in \mathcal{C}, \forall\, K \in \mathcal{K}_{r(\check{W}_i)}\ (\gamma_{C,\check{W}_i,K} \leq \gamma_{C,\hat{W}_i,o_{r(\check{W}_i)}(K)}), 1 \leq i \leq n \end{aligned} \tag{15}$$

For a subset of clubs, the key assignments are already fixed by external factors (cf. Def. 10), which is expressed by the following set of constraints:

$$\forall\, C \in \bar{\mathcal{C}}, \forall\, W \in \hat{\mathcal{W}} \cup \check{\mathcal{W}}\ (1 \leq \gamma_{C,W,f_W(C)}) \tag{16}$$

It remains to encode the key similarity requirement (Def. 13) into linear constraints. Two aspects have to be taken into consideration: First, the keys for teams depend on the grid used in the respective divisions, whereas the clubs' keys are determined by the reference grids per week scheme. Therefore, parallel keys have to be determined, as the grids in use are different in general. Second, the keys assigned to each of the teams must be similar (which includes equal) to one of the parallel keys determined in the previous step. As one valid key is sufficient for both steps and the previous constraints ensure that at most one key is assigned to each team and club/week scheme, all candidates in S and P, respectively, can be added up in (17).

$$\begin{aligned} &\forall\, C \in \mathcal{C}, \forall\, W \in \hat{\mathcal{W}} \cup \check{\mathcal{W}}, \forall\, T \in \mathcal{T}_{C_W}, \forall\, K \in \mathcal{K}_{r(W)} \\ &(\gamma_{C,W,K} \leq \textstyle\sum_{(K,K') \in P_{r(W),g(d(T))}} (\sum_{K'' \in S_{g(d(T))}(K')} \tau_{T,K''})) \end{aligned} \tag{17}$$

While the previous set of constraints (17) guarantees that the keys assigned to teams are similar to the keys assigned to their clubs, we need to express that it is indeed more desirable to assign parallel keys (i.e., equal keys in case of equal grids) than only similar keys. Assigning a similar (but not parallel) key can be considered as the resolution of a conflict situation, as the key is required by another team of the same division as well. Therefore, the (auxiliary) binary variables for conflicts (Def. 18) can be used to "count" the number of teams to which a similar (but not parallel) key is assigned.

$$\begin{aligned} &\forall\, C \in \mathcal{C}, \forall\, W \in \hat{\mathcal{W}} \cup \check{\mathcal{W}}, \forall\, T \in \mathcal{T}_{C_W}, \forall\, K \in \mathcal{K}_{g(d(T))} \\ &(\tau_{T,K} \leq \xi_T + \textstyle\sum_{(K,K') \in P_{g(d(T)),r(W)}} \gamma_{C,W,K'}) \end{aligned} \tag{18}$$

In accordance with Def. 15, the optimization goal can be formulated in terms of the sum of the binary variables for conflicts, which should be minimized. Adding all sets of constraints, the linear program for the MLSSP can be specified as shown in (19).

$$min. \sum_{T \in \mathcal{T}} \xi_T\ s.t.\ (12) \wedge (13) \wedge (14) \wedge (15) \wedge (16) \wedge (17) \wedge (18) \tag{19}$$

5 MLSSP is $\mathcal{NP}$-complete

The MLSSP is defined as an optimization problem in this paper, whereas the complexity class $\mathcal{NP}$ is only defined for decision problems. The proof for $\mathcal{NP}$-completeness will therefore refer to the decision problem, i.e., whether an instance of the MLSSP is satisfiable according to Def. 14.

▶ **Lemma 19.** *MLSSP* $\in \mathcal{NP}$.

Proof. To prove that MLSSP $\in \mathcal{NP}$ holds, we must show that a witness for MLSSP can be verified within polynomial runtime. As a verifier, the constraint sets defined in Eq. (12) - (18) can be used. As many constraints involve a summation or iteration over a set of keys, which depends on the size $|G|$ of the respective grid, the maximum size of any grid in use is denoted as $|G_{\max}|$ in the following. The runtime complexity of checking an inequality is $\mathcal{O}(1)$, we further assume that the same holds for applying the functions o (Def. 4) and f (Def. 10), and evaluating the relations P (Def. 5) and S (Def. 7) using suitable hash functions.

Table 3 provides an overview of the runtime complexity of verifying constraints of Eq. (12) - (18). Note that the number of similar and parallel keys is bounded by $|G_{\max}|$ for constraints (17) and (18).

Table 3 Runtime complexity to verify constraint satisfaction

Constraint	Runtime Complexity
(12)	$\mathcal{O}(\lvert\mathcal{C}\rvert \cdot \lvert\mathcal{W}\rvert \cdot \lvert G_{\max}\rvert)$
(13)	$\mathcal{O}(\lvert\mathcal{T}\rvert \cdot \lvert G_{\max}\rvert)$
(14)	$\mathcal{O}(\lvert\mathcal{T}\rvert \cdot \lvert G_{\max}\rvert)$
(15)	$\mathcal{O}(\lvert\mathcal{C}\rvert \cdot \lvert\mathcal{W}\rvert \cdot \lvert G_{\max}\rvert)$
(16)	$\mathcal{O}(\lvert\mathcal{C}\rvert \cdot \lvert\mathcal{W}\rvert)$
(17)	$\mathcal{O}(\lvert\mathcal{T}\rvert \cdot \lvert G_{\max}\rvert \cdot (\lvert S_{G_{\max}}\rvert + \lvert P_{G_{\max},G_{\max}}\rvert)) = \mathcal{O}(\lvert\mathcal{T}\rvert \cdot \lvert G_{\max}\rvert^2)$
(18)	$\mathcal{O}(\lvert\mathcal{T}\rvert \cdot \lvert G_{\max}\rvert \cdot (\lvert P_{G_{\max},G_{\max}}\rvert)) = \mathcal{O}(\lvert\mathcal{T}\rvert \cdot \lvert G_{\max}\rvert^2)$
Overall	$\mathcal{O}(\lvert\mathcal{C}\rvert \cdot \lvert\mathcal{W}\rvert \cdot \lvert G_{\max}\rvert + \lvert\mathcal{T}\rvert \cdot \lvert G_{\max}\rvert^2)$

Overall, a witness for MLSSP can be verified in polynomial time, proving Lemma 19. ◀

▶ **Lemma 20.** $\forall L \in \mathcal{NP} : L \leq_p$ *MLSSP.*

Proof. We prove that MLSSP is $\mathcal{NP}$-hard by a polynomial reduction from 3-SAT, i.e., 3-SAT $\leq_p$ MLSSP. Let $\phi = (x_{1,1} \vee x_{1,2} \vee x_{1,3}) \wedge \cdots \wedge (x_{n,1} \vee x_{n,2} \vee x_{n,3})$ be a formula of propositional logic in 3-CNF with n clauses $c_1, \ldots c_n$. Without loss of generality, we assume each clause to consist of exactly three literals (clauses with less than three literals can be brought into this form by doubling or tripling the literals present).

A formula ϕ' is constructed by adding a variable v – which does not occur in ϕ – to each clause of ϕ, and add a clause $(\neg v)$, such that ϕ' and ϕ are equisatisfiable:

$$\phi' = (x_{1,1} \vee x_{1,2} \vee x_{1,3} \vee v) \wedge \cdots \wedge (x_{n,1} \vee x_{n,2} \vee x_{n,3} \vee v) \wedge (\neg v) \tag{20}$$

Based on ϕ', an MLSSP instance m is constructed in the following way:

- $\mathcal{G} = \{G_2, G_4\}, |G_2| = 2, |G_4| = 4$
- $\mathcal{K}_{G_2} = \{1, 2\}, \mathcal{K}_{G_4} = \{1, 2, 3, 4\}$
- $\hat{\mathcal{W}} = \{\hat{W}_1\}, \check{\mathcal{W}} = \{\check{W}_1\}, \mathcal{W} = \{W_0, \hat{W}_1, \check{W}_1\}$, with $r(\hat{W}_1) = r(\check{W}_1) = G_2$
- $P_{G_2,G_2} = \{(1,1),(2,2)\}$, $P_{G_2,G_4} = \{(1,1),(2,3)\}$, $P_{G_4,G_2} = \{(1,1),(3,2)\}$, $P_{G_4,G_4} = \{(1,1),(2,2),(3,3),(4,4)\}$

- $o_{G_2}(1) = 2$, $o_{G_2}(2) = 1$, $o_{G_4}(1) = 3$, $o_{G_4}(2) = 4$, $o_{G_4}(3) = 1$, $o_{G_4}(4) = 2$
- $S_{G_2} = \{(1,1), (2,2)\}$, $S_{G_4} = \{(1,1), (1,2), (1,4), (2,1), (2,2), (2,3), (3,2), (3,3), (3,4), (4,1), (4,3), (4,4)\}$
- A club $C_{i,j}$ is created for each distinct $x_{i,j} \in \phi'$, and a club V is created for v
- $\vec{\mathcal{C}} = \{V\}$, $f_{\hat{W}_1}(V) = 2$
- For each clause $c_1, \dots c_n$ in ϕ' (except for the last clause $(\neg v)$), a division D_i with four teams $T_{i,1}, T_{i,2}, T_{i,3}$ and $T_{i,V}$ is created. Each team $T_{i,j}$ corresponds to a literal $x_{i,j}$ in ϕ', $1 \leq j \leq 3$. Each team $T_{i,V}$ corresponds to the literal v.
- Teams corresponding to negative literals follow the week scheme $\check{W}_1$, those corresponding to positive literals follow week scheme $\hat{W}_1$.

▶ **Proposition 21.** *$\phi' \in$ 3-SAT $\iff$ m is satisfiable*

Proof. For each club $C_{i,j}$, let $(a_{\hat{W}_1}(C_{i,j}) = 1 \wedge a_{\check{W}_1}(C_{i,j}) = 2) \iff$ *true* is assigned to $x_{i,j}$ in ϕ', and $(a_{\hat{W}_1}(C_{i,j}) = 2 \wedge a_{\check{W}_1}(C_{i,j}) = 1) \iff$ *false* is assigned to $x_{i,j}$ in ϕ'. $\phi' \in$ 3-SAT holds if at least one literal of each clause in ϕ' yields *true*. This means that in each clause, there must be at least one positive literal with variable assignment *true* or one negative literal with variable assignment *false*.

Equivalently, to satisfy m, there must be at least one team T of a club C following week scheme $\hat{W}_1$ with $a_{\hat{W}_1}(C) = 1$ and $a_{\check{W}_1}(C) = 2$, or following week scheme $\check{W}_1$ with $a_{\hat{W}_1}(C) = 2$ and $a_{\check{W}_1}(C) = 1$ in each division D_i of m, resulting in at least one occurrence of $a_W(C) = 1$. The same must hold vice versa (at least one occurrence of $a_W(C) = 2$) for each division of m, which is trivially fulfilled because $a_{\hat{W}_1}(V) = 2$ and one team $T_{i,V}$ of club V following week scheme $\hat{W}_1$ is present in each division D_i.

This observation is depicted in Tab. 4, showing a case distinction for the eight possible cases. The columns $a_W(C_{i,j})$ contain the key assignments for the clubs $C_{i,j}$, while the columns $k(T_{i,j})$ contain the sets of possible key assignments for the teams $T_{i,j}$ according to the relations P_{G_2,G_4} and S_{G_4}. A valid key assignment for the teams $T_{i,j}$ is highlighted in bold font for the first seven cases. Only in the last case, which corresponds to all variables of positive literals being assigned *false* and all variables of negative literals being assigned *true* in the respective clause of ϕ', no valid key assignment is possible for the teams of D_i, proving Proposition 21.

Table 4 Case distinction demonstrating the equivalence of clauses and divisions

$a_W(C_{i,1})$	$a_W(C_{i,2})$	$a_W(C_{i,3})$	$a_W(V)$	$k(T_{i,1})$	$k(T_{i,2})$	$k(T_{i,3})$	$k(T_{i,V})$	valid
1	1	1	2	**1**/2/4	1/**2**/4	1/2/**4**	2/**3**/4	✓
1	1	2	2	**1**/2/4	1/**2**/4	2/**3**/4	2/3/**4**	✓
1	2	1	2	**1**/2/4	**2**/3/4	1/2/**4**	2/**3**/4	✓
1	2	2	2	**1**/2/4	**2**/3/4	2/**3**/4	2/3/**4**	✓
2	1	1	2	**2**/3/4	**1**/2/4	1/2/**4**	2/**3**/4	✓
2	1	2	2	**2**/3/4	**1**/2/4	2/**3**/4	2/3/**4**	✓
2	2	1	2	**2**/3/4	2/**3**/4	**1**/2/4	2/3/**4**	✓
2	2	2	2	2/3/4	2/3/4	2/3/4	2/3/4	✗

◀

Lemma 20 follows from Proposition 21, as $\phi \in$ 3-SAT $\iff$ $\phi' \in$ 3-SAT. ◀

▶ **Theorem 22.** *MLSSP is $\mathcal{NP}$-complete.*

Proof. Theorem 22 follows directly from Lemma 19 and Lemma 20. ◀

There are some practical implications of the proof result, which will briefly discussed in the following. The key learning is that for very large instances, some multilevel divide-and-conquer approach will still be necessary to schedule leagues from national to regional level, even with tool support. For very large problem instances (e.g., an instance involving all football leagues of Spain), the scheduling problem gets intractable from a practical point of view, even if all required data was accessible to a single person. Heuristic approaches might succeed finding a sufficiently good solution, but usually need a basic feasible solution to start with, though, which itself is challenging to determine for the MLSSP.

6 Implementation

Our approach is implemented as a C# desktop application [35]. The software architecture of our implementation is sketched in Fig. 2 in form of a UML component diagram.

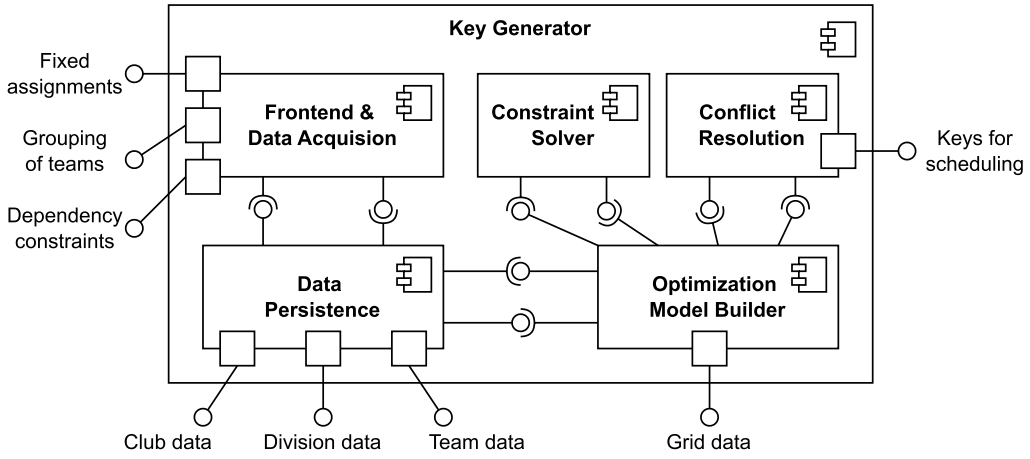


Figure 2 Component diagram for the key generator

Using the frontend & data acquisition component, the grouping of teams into divisions and the dependency constraints provided via an internet portal can be imported into the key generator via comma-separated values (CSV) files. Fixed assignments must be entered manually. The data persistence layer processes data of clubs, divisions and teams, filters out information that is irrelevant for the match scheduling task and stores the relevant information on disk. The optimization model builder constructs an ILP based on the input data and the information on parallel, opposite and similar keys for each grid in use (grids for 6 - 14 teams are supported). The ILP is subsequently solved to optimality by a SAT or ILP solver. We use the CP-SAT solver [27] in our implementation. With the conflict resolution component, the user can modify the computed solution by deciding which teams should receive similar keys in conflict situations. This makes it possible to balance interests between the clubs involved, while the solution quality remains unchanged. The data acquisition and conflict resolution components form the user interface, whereas the other components are hidden in the back end.

7 Evaluation

In this section, the practical applicability of our approach shall be investigated. Particularly, the following research questions will be answered in the course of this section:

Q1 Can the approach be used to determine solutions for scheduling instances of realistic size?

Q2 Which factors influence the scalability of the approaches, i.e., which characteristics of the input affect the runtime and solution quality of the key generation process?

Q3 How does the approach perform compared to similar existing approaches?

Outline. Q1 will be investigated by solving a real-world case study involving German table tennis leagues (Sect. 7.1). To answer Q2, the approach is applied on generated instances with different characteristic parameters (Sect. 7.2). Finally, we compare our approach to an implementation by Li et al. [22] to answer Q3 (Sect. 7.3).

Experimental setup. All experiments were run on a Windows 11 notebook with a 2.0 GHz Intel i7 processor and 32 GB main memory. The CP-SAT solver was run with 8 parallel workers (i.e., threads) and a time-out of 300 seconds. To reduce the effect of outliers, the experiments were run 10 times (Q1), or on 10 different instances that were generated with the same characteristic parameters (Q2 and Q3), respectively. From these 10 test runs, the median values of the measured runtimes and objective function values were computed.

7.1 Case Study

We applied our approach to create the schedules for a district of the West-German Table Tennis Association (WTTV) for three seasons [36]. The characteristics of the three input instances are summed up in Tab. 5. They differ with respect to the number of teams, clubs, and divisions involved. It can be further observed that for the majority of teams, dependencies were defined (i.e., week schemes other than W_0 according to Def. 6 were used). Five week schemes were available for the clubs as predefined by the WTTV. For a few clubs, key assignments were fixed by the schedulers of super-ordinate levels according to Def. 10.

Table 5 Real-world case studies from German amateur table tennis leagues

Season	# divisions	# teams (with dependencies)	# clubs (with fixed assignm.)	# week schemes	Runtime (in s)	Objective fun. value
2022/23	94	885 (638)	194 (18)	5	2.876	35
2023/24	49	516 (387)	107 (16)	5	2.145	16
2024/25	50	496 (392)	106 (11)	5	2.225	14

For all three instances, the application returns an optimal solution within a few seconds in all ten test runs. It can be concluded that the implementation is suitable for solving instances of realistic size. Limitations will be investigated in the following performance evaluation.

7.2 Performance Evaluation

As the implementation comprises an exact approach to determine the scheduling solution, the runtime is expected to increase exponentially for larger inputs. Other characteristic parameters (cf. Tab. 5) may also have an impact on the runtime and solution quality.

Base scenario construction. To investigate the approach's scalability, we derive a base scenario from the real-world case study of Sect. 7.1, which is subsequently varied with respect to its characteristic parameters. The base scenario consists of 100 *divisions* with 10 *teams per division*, which belong to 200 *clubs*. For 75% of the teams, *dependencies* are defined, and for 10% of the clubs, *key assignments* are *fixed* prior to the optimization. The number of *week schemes* is set to 5 (including W_0) in the base scenario.

Instance generation. To analyze the effect of each characteristic parameter, their values are varied separately, resulting in 30 different configurations (cf. Tab. 6). For each configuration, 10 instances were generated at random [36] and handed over to CP-SAT solver.

Table 6 Performance evaluation varying different parameters of the base scenario

Characteristic	Runtime (in s)					Objective function value				
# Divisions	20	50	100	200	500	20	50	100	200	500
	0.35	2.79	55.02	T/O	T/O	1	4	10	257	1202
# Teams per division	6	8	10	12	14	6	8	10	12	14
	0.28	1.13	2.87	2.97	3.40	2	2.5	6	8	8.5
# Clubs	20	50	100	200	500	20	50	100	200	500
	T/O	2.57	2.93	1.93	0.62	66.5	7	2.5	2.5	1.5
# Week schemes	3	5	7	9	11	3	5	7	9	11
	12.04	2.43	2.35	2.24	1.71	10	5	4	3	3
% Teams with dependencies	75	80	85	90	95	75	80	85	90	95
	2.73	2.55	2.60	2.88	3.74	5	4	4.5	5	7
% Clubs with fixed assignm.	10	20	30	40	50	10	20	30	40	50
	2.55	2.56	1.74	1.47	1.64	4.5	4	19	25.5	40

Results. The median values for runtime and solution quality are depicted in Tab. 6, while the measured values for each test run are listed in Appendices A.1 and A.2. For all conducted test runs, the CP-SAT solver returned an optimal or feasible solution, or classified the problem as infeasible. The values measured for infeasible instances were excluded from computing the median values.

The largest impact with respect to runtime and solution quality was observed when varying the number of **divisions**. For 200 and 500 divisions, the solver reached the time-out in all 10 runs, whereby the solution quality deteriorated substantially. This seems plausible, as the number of divisions corresponds to the size of the constructed ILP. Considering the number of **teams per division**, both runtime and solution quality deteriorate when increasing the division size. The effects are less pronounced than the effects of increasing the number of divisions, though, which can be explained by the increased flexibility of larger division sizes regarding key assignment.

The measured values for both runtime and solution quality show that reducing the number of **clubs** makes the problem harder. This seems valid, because an increased number of teams per club leads to more teams of a club following the same week scheme, severely restricting the possibilities of assigning suitable keys. Similarly, reducing the number of **week schemes** apparently increases the problem complexity, as more teams follow the same week scheme.

Increasing the portion of **teams with dependencies** slightly increases the runtime consumption at almost constant objective function values. The most noticeable difference occurs when increasing the portion to 95%, presumably caused by lacking flexibility regarding

key assignment. Interestingly, increasing the portion of **clubs with fixed assignments** reduces the runtime but also the solution quality. An explanation could be that fixed assignments reduce the search space, but also cause unavoidable conflict situations.

In theory, one would expect the same runtime measurements and objective function values in all cells of Tab. 6 which represent the base scenario. When conducting performance tests, we varied each characteristic parameter separately, such that the base scenario occurred multiple times. As different instances were generated, the measurement results slightly differ.

7.3 Comparison to Similar Approaches

We compared our implementation to the approach of Li et al. [22], who determine optimal schedules based on capacity constraints without defining interdependencies between teams. They implemented both a heuristic and an exact approach and applied them on instances with 3,5 and 10 divisions with different numbers of teams (cf. Tab. 7). All instances were made publicly available [21], such that we were able to implement an adapter for their file format and run experiments on the same input data.

Table 7 Performance comparison to exact and heuristic approaches of Li et al. [22]

Approach	Runtime (in s) per instance type					
	3-1	3-2	5-1	5-2	10-1	10-2
Li et al. (exact)	66.33	16.55	588.10	803.71	678.28	528.63
Li et al. (heuristic)	0.02	0.11	0.79	0.01	3.97	0.93
This approach	0.16	0.29	0.05	0.06	0.09	0.10

Optimal results could be determined in all 60 test runs. The median values of the measured runtimes for 10 instances which were constructed with equal characteristics are shown in Tab. 7. Our implementation performs relatively well compared to the heuristic approach and clearly outperforms the exact approach. Interestingly, our approach takes most time for the instances with only three divisions, which can be explained by the reduced possibilities to avoid conflict situations.

The exact approach of Li et al. and our approach share the optimization goal (minimizing constraint violations) and the general setting (teams play DRR tournaments in several groups, clubs' venues have capacity constraints). The main reason for the differences in runtime performance is that the search space constructed by Li et al. is much larger, as they use decision variables for every single match, while our approach operates on the level of assigning keys to teams. Li et al. can potentially further reduce the number of constraint violations via a sophisticated allocation of bye weeks to teams, or by stretching the schedule of smaller divisions to more weeks than necessary. However, our approach allows clubs to define that two specific teams should play in turn or in parallel, while Li et al. solely aim at minimizing the number of capacity constraint violations.

7.4 Summary and Threats to Validity

Revisiting our research questions, it can be stated that our approach is suitable to be applied on real-world instances (Q1). Its scalability is mainly restricted by the number of divisions and teams involved (Q2), but performs well in comparison to existing approaches (Q3).

The conducted experiments have a few limitations to be discussed in the following. Due to the lack of real-world examples, all instances used in the experiments of the performance evaluation (Sect. 7.2) were artificially generated. The time-out was set to a rather low value,

which was indeed sufficient to identify whether modifying a characteristic parameter increases or decreases runtime and solution quality. Using more than one solver for the experiments and varying the characteristic parameters in combination could give more reliable insights into performance bottlenecks.

Furthermore, the differences with respect to runtime and solution quality often varied noticeably between test runs of the same configuration (cf. Appendices A.1 and A.2). This implies that the grouping of teams into divisions has a major impact on the problem complexity, which was not covered by the performance analysis. Usually, teams are not distributed randomly over the groups, but according to regional aspects. This means that the generated instances are probably harder to solve than practical use cases.

Finally, the presented evaluation was purely technical, whereas the validity of the applied quality metrics should be evaluated in the course of a user study.

8 Conclusion and Future Work

This paper defines an extended version of the Multi-League Sports Scheduling Problem (MLSSP) that takes capacity constraints, team interdependencies and constraints from superordinate schedules into account. The MLSSP is specified in form of a linear optimization problem, the corresponding decision problem is proven to be $\mathcal{NP}$-complete. The approach was applied to scheduling German table tennis leagues of a certain district, underpinning its practical applicability. Experiments with generated instances of varying size highlight the factors influencing the approach's scalability.

In future research, meta-heuristics such as simulated annealing, tabu search or genetic algorithms should be implemented to handle even larger instances. Regarding the problem definition, the similarity requirement of assignments for clubs and teams could be replaced by a requirement for pairwise similarity of assignments for teams of the same club, leading to better results in practice. Further constraints, such as inter-club dependencies or requirements for teams playing home or away in certain weeks should be easy to include. Finally, the validity of quality measures should be assessed in an empirical evaluation with practitioners.

References

1 Fernando Alarcón, Guillermo Durán, Mario Guajardo, Jaime Miranda, Hugo Muñoz, Luis Ramírez, Mario Ramírez, Denis Sauré, Matías Siebert, Sebastian Souyris, Andrés Weintraub, Rodrigo Wolf-Yadlin, and Gonzalo Andres Zamorano. Operations Research Transforms the Scheduling of Chilean Soccer Leagues and South American World Cup Qualifiers. *Interfaces*, 47(1):52–69, 2017. doi:10.1287/INTE.2016.0861.

2 Thomas Bartsch, Andreas Drexl, and Stefan Kröger. Scheduling the professional soccer leagues of Austria and Germany. *Computers & Operations Research*, 33(7):1907–1937, 2006. doi:10.1016/J.COR.2004.09.037.

3 Dirk Briskorn. Alternative IP Models for Sport Leagues Scheduling. In Jörg Kalcsics and Stefan Nickel, editors, *Operations Research, Proceedings 2007, Selected Papers of the Annual International Conference of the German Operations Research Society (GOR)*. Springer, 2007. doi:10.1007/978-3-540-77903-2_62.

4 David Van Bulck and Dries R. Goossens. Handling fairness issues in time-relaxed tournaments with availability constraints. *Comput. Oper. Res.*, 115:104856, 2020. doi:10.1016/J.COR.2019.104856.

5 Wayne Burrows and Christopher P. Tuffley. Maximising common fixtures in a round robin tournament with two divisions. *Australas. J Comb.*, 63:153–169, 2015. URL: http://ajc.maths.uq.edu.au/pdf/63/ajc_v63_p153.pdf.

6 Mats Carlsson, Mikael Johansson, and Jeffrey Larson. Scheduling double round-robin tournaments with divisional play using constraint programming. *European Journal of Operational Research*, 259(3):1180–1190, 2017. doi:10.1016/J.EJOR.2016.11.033.

7 Morteza Davari, Dries R. Goossens, Jeroen Beliën, Roel Lambers, and Frits C. R. Spieksma. The multi-league sports scheduling problem, or how to schedule thousands of matches. *Oper. Res. Lett.*, 48(2):180–187, 2020. doi:10.1016/J.ORL.2020.02.004.

8 Dominique de Werra, Loïc Jacot-Descombes, and P. Masson. A constrained sports scheduling problem. *Discret. Appl. Math.*, 26(1):41–49, 1990. doi:10.1016/0166-218X(90)90019-9.

9 Federico Della Croce and Dario Oliveri. Scheduling the Italian football league: An ILP-based approach. *Computers & Operations Research*, 33(7):1963–1974, 2006. doi:10.1016/J.COR.2004.09.025.

10 Guillermo Alfredo Durán, Mario Guajardo, Agustina F. López, Javier Marenco, and Gonzalo Andres Zamorano. Scheduling Multiple Sports Leagues with Travel Distance Fairness: An Application to Argentinean Youth Football. *INFORMS J. Appl. Anal.*, 51(2):136–149, 2021. doi:10.1287/INTE.2020.1048.

11 Dries Goossens. Optimization in sports league scheduling: experiences from the Belgian Pro League soccer. In *International conference on operations research and enterprise systems*, pages 3–19. Springer, 2017.

12 Dries R. Goossens and Frits C. R. Spieksma. Scheduling the Belgian Soccer League. *Interfaces*, 39(2):109–118, 2009. doi:10.1287/INTE.1080.0402.

13 Dries R. Goossens and Frits C. R. Spieksma. Breaks, cuts, and patterns. *Oper. Res. Lett.*, 39(6):428–432, 2011. doi:10.1016/J.ORL.2011.09.001.

14 Mark R. Grabau. Softball Scheduling as Easy as 1-2-3 (Strikes You're Out). *Interfaces*, 42(3):310–319, 2012. doi:10.1287/INTE.1110.0559.

15 Xiaoyun Ji and John E Mitchell. Finding optimal realignments in sports leagues using a branch-and-cut-and-price approach. *Int. J. Oper. Res.*, 1(1-2):101–122, 2005.

16 Graham Kendall. Scheduling English football fixtures over holiday periods. *J. Oper. Res. Soc.*, 59(6):743–755, 2008. doi:10.1057/PALGRAVE.JORS.2602382.

17 Fabian Kleintges-Topoll. Blog: Geringe Auswirkungen durch Strukturreform. https://www.mytischtennis.de/public/mytt-bloggt/18283/, 2023. Accessed: 13 Mar 2025.

18 Sigrid Knust and Daniel Lücking. Minimizing costs in round robin tournaments with place constraints. *Computers & Operations Research*, 36(11):2937–2943, 2009. doi:10.1016/J.COR.2009.01.004.

19 Jari Kyngäs, Kimmo Nurmi, Nico Kyngäs, George Lilley, Thea Salter, and Dries Goossens. Scheduling the Australian football league. *Journal of the Operational Research Society*, 68(8):973–982, 2017. doi:10.1057/S41274-016-0145-8.

20 Jeffrey Larson and Mikael Johansson. Constructing schedules for sports leagues with divisional and round-robin tournaments. *Journal of Quantitative Analysis in Sports*, 10(2):119–129, 2014.

21 Miao Li, Morteza Davari, and Dries R. Goossens. Multi League Repository. https://www.sportscheduling.ugent.be/RobinX/multiLeagueRepo.php, 2023.

22 Miao Li, Morteza Davari, and Dries R. Goossens. Multi-league sports scheduling with different leagues sizes. *Eur. J. Oper. Res.*, 307(1):313–327, 2023. doi:10.1016/J.EJOR.2022.10.010.

23 Miao Li and Dries Goossens. Grouping and scheduling multiple sports leagues: an integrated approach. *Journal of the Operational Research Society*, pages 1–19, 2024.

24 Vardges Melkonian. An Integer Programming Approach for Scheduling a Professional Sports League. *American Journal of Computational Mathematics*, 14(4):401–423, 2024.

25 Ryuhei Miyashiro, Hideya Iwasaki, and Tomomi Matsui. Characterizing Feasible Pattern Sets with a Minimum Number of Breaks. In Edmund K. Burke and Patrick De Causmaecker, editors, *Practice and Theory of Automated Timetabling IV, 4th International Conference, PATAT 2002*, pages 78–99. Springer, 2002. doi:10.1007/978-3-540-45157-0_5.

26 Douglas Moody, Graham Kendall, and Amotz Bar-Noy. Youth sports leagues scheduling. In *Proceedings of the 8th International Conference on the Practice and Theory of Automated Timetabling*, pages 283–293, 2010.

27 Laurent Perron and Frédéric Didier. CP-SAT. https://developers.google.com/optimization/cp/cp_solver/.

28 Diego Recalde, Ramiro Torres, and Polo Vaca. Scheduling the professional Ecuadorian football league by integer programming. *Comput. Oper. Res.*, 40(10):2478–2484, 2013. doi:10.1016/J.COR.2012.12.017.

29 Robert M Saltzman and Richard M Bradford. Optimal realignments of the teams in the National Football League. *European Journal of Operational Research*, 93(3):469–475, 1996.

30 Jörn Schönberger. Scheduling of sport league systems with inter-league constraints. Technical report, Diskussionsbeiträge aus dem Institut für Wirtschaft und Verkehr, 2015.

31 Jörn Schönberger. The championship timetabling problem-construction and justification of test cases. In *Proceedings of MathSport International 2017 Conference*, volume 330, 2017.

32 Ling-Huey Su, Yufang Chiu, and TC Edwin Cheng. Sports tournament scheduling to determine the required number of venues subject to the minimum timeslots under given formats. *Computers & Industrial Engineering*, 65(2):226–232, 2013. doi:10.1016/J.CIE.2013.02.021.

33 Túlio A. M. Toffolo, Jan Christiaens, Frits C. R. Spieksma, and Greet Vanden Berghe. The sport teams grouping problem. *Ann. Oper. Res.*, 275(1):223–243, 2019. doi:10.1007/S10479-017-2595-Z.

34 David Van Bulck, Dries R Goossens, and Frits CR Spieksma. Scheduling a non-professional indoor football league: a tabu search based approach. *Ann. Oper. Res.*, 275(2):715–730, 2019. doi:10.1007/S10479-018-3013-X.

35 Nils Weidmann. KeyGenerator. https://github.com/NilsWeidmann/KeyGenerator.

36 Nils Weidmann. Multi-League Sports Scheduling. https://github.com/NilsWeidmann/Multi-League-Sports-Scheduling, 2025.

A Appendix

A.1 Performance Evaluation Results - Runtime (in s)

Config.		1	2	3	4	5	6	7	8	9	10
# Divisions	20	0,365	0,232	0,301	0,435	0,246	0,359	0,483	0,335	0,208	0,512
	50	2,084	2,793	2,735	1,803	2,814	2,385	3,206	2,784	2,786	4,817
	100	51,301	54,279	58,318	112,013	302,04	302,002	55,759	50,888	44,981	49,422
	200	301,603	302,127	302,082	301,738	301,888	301,895	301,719	302,308	302,037	302,175
	500	302,417	302,403	302,439	301,462	302,024	301,689	302,085	302,635	301,829	302,617
# Teams per divis.	6	0,288	0,238	0,310	0,409	0,245	0,243	0,299	0,272	0,517	0,208
	8	1,422	1,136	1,120	0,325	0,306	0,300	1,846	1,684	0,805	1,754
	10	2,071	2,030	1,976	2,850	2,273	3,595	2,880	3,073	5,864	3,478
	12	2,525	2,528	2,332	3,060	1,503	3,002	3,045	3,320	3,451	2,936
	14	2,826	2,879	3,013	3,203	3,716	3,806	3,829	3,924	3,444	3,359
# Clubs	20	301,449	302,021	302,143	301,490	301,597	301,649	301,605	302,090	301,909	302,104
	50	2,344	2,134	2,166	2,905	2,410	2,735	4,210	2,388	301,990	5,037
	100	2,052	3,688	2,056	2,955	3,220	2,661	2,973	2,910	5,462	2,674
	200	0,439	1,794	2,855	2,629	2,060	2,629	2,474	0,524	0,509	0,529
	500	0,558	0,549	0,542	1,019	1,045	0,601	0,631	0,623	0,607	1,232
# Week schemes	3	301,637	12,035	0,372	2,616	2,683	17,639	302,044	302,012	2,625	2,576
	5	2,431	2,004	2,240	2,405	3,739	2,772	2,279	3,636	2,489	0,140
	7	3,919	1,261	2,127	3,100	0,495	4,529	3,952	0,468	0,498	2,567
	9	2,148	0,425	3,332	2,326	2,944	4,396	1,265	0,616	3,435	1,901
	11	1,457	1,164	1,756	2,245	2,501	1,765	0,550	1,671	0,512	1,803
% Teams with dep.	75	2,082	1,967	2,120	2,952	2,540	2,945	2,855	2,799	2,809	2,654
	80	2,138	2,216	2,273	2,708	3,024	2,265	2,868	2,486	2,938	2,613
	85	2,177	2,211	2,275	2,796	2,326	3,648	2,713	2,574	2,873	2,628
	90	2,315	2,214	2,487	3,026	3,292	2,746	2,793	2,995	2,964	3,065
	95	2,508	2,512	25,594	3,104	2,958	17,205	3,031	4,380	4,477	301,966
% Clubs with f. a.	10	2,032	2,398	3,187	2,473	2,439	2,437	2,726	2,647	2,628	2,835
	20	1,911	1,935	2,700	2,419	0,432	3,669	4,231	4,172	1,538	3,031
	30	2,358	1,735	0,839	2,244	0,413	0,407	0,985	2,008	1,141	3,090
	40	2,483	1,471	1,873	0,976	1,465	0,366	1,356	1,042	0,417	2,341
	50	0,303	0,301	0,322	1,443	1,365	0,324	2,944	0,361	1,832	0,308

● Optimal ● Feasible ● Infeasible

A.2 Performance Evaluation Results - Objective Function Values

Config.		1	2	3	4	5	6	7	8	9	10
# Divisions	20	1	1	4	1	1	1	2	1	1	1
	50	4	2	6	5	3	5	3	6	4	2
	100	11	3	6	12	34	46	10	7	9	10
	200	262	274	212	225	246	258	275	285	256	254
	500	1202	1225	1272	1248	1202	1175	1148	1166	1218	1197
# Teams per divis.	6	1	0	2	1	4	2	2	4	1	3
	8	1	3	5	0	2	0	3	3	2	4
	10	6	3	12	7	8	3	6	6	6	5
	12	9	7	12	3	6	5	12	12	9	2
	14	3	10	11	9	3	4	11	9	8	8
# Clubs	20	68	69	62	79	68	55	60	73	65	63
	50	3	9	6	8	5	12	5	8	22	5
	100	2	6	4	0	3	2	2	1	5	4
	200	2	4	13	3	2	4	5	0	0	1
	500	0	2	1	1	4	4	3	0	1	3
# Week schemes	3	7	11	0	4	10	10	15	21	10	12
	5	2	5	8	6	4	5	5	5	9	0
	7	4	7	4	5	3	6	2	2	3	5
	9	0	1	2	3	2	6	4	3	4	3
	11	2	6	3	3	7	2	4	2	2	3
% Teams with dep.	75	11	4	6	2	6	4	6	2	3	7
	80	4	5	6	6	4	3	3	6	4	2
	85	6	5	6	3	7	5	4	3	4	4
	90	3	4	5	5	4	8	8	4	5	12
	95	8	7	6	9	9	6	4	7	5	28
% Clubs with f. a.	10	2	5	2	5	5	5	4	4	6	3
	20	4	7	4	3	3	5	11	11	2	3
	30	9	20	23	14	7	0	16	19	21	24
	40	28	38	28	20	15	0	23	20	0	31
	50	0	0	0	42	47	0	32	0	38	0

● Optimal ● Feasible ● Infeasible

Reducing Quantum Circuit Synthesis to #SAT

Dekel Zak
Leiden University, The Netherlands

Jingyi Mei
Leiden University, The Netherlands

Jean-Marie Lagniez
Artois University, Lens, France

Alfons Laarman
Leiden University, The Netherlands

Abstract

Quantum circuit synthesis is the task of decomposing a given quantum operator into a sequence of elementary quantum gates. Since the finite target gate set cannot exactly implement any given operator, approximation is often necessary. Model counting, or #SAT, has recently been demonstrated as a promising new approach for tackling core problems in quantum circuit analysis. In this work, we show for the first time that the universal quantum circuit synthesis problem can be reduced to maximum model counting. We formulate a #SAT encoding for exact and approximate depth-optimal quantum circuit synthesis into the Clifford+T gate set. We evaluate our method with an open-source implementation that uses the maximum model counter d4Max as a backend. For this purpose, we extended d4Max with support for complex and negative weights to represent amplitudes. Experimental results show that existing classical tools have potential for the quantum circuit synthesis problem.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Quantum computation theory

Keywords and phrases Maximum weighted model counting, quantum circuit synthesis

Digital Object Identifier 10.4230/LIPIcs.CP.2025.38

Supplementary Material *Software (Source Code)*: `https://github.com/System-Verification-Lab/Quokka-Sharp.git`

Funding This work is supported by the Dutch National Growth Fund, as part of the Quantum Delta NL program.

1 Introduction

Quantum algorithms are typically specified using higher-level constructs such as (classical) procedures and quantum Fourier transform (QFT) [30]. For their efficient implementation on physical devices, which typically operate with a finite gate set, they need to be broken down into a quantum circuit, a sequence of quantum gates. Error-corrected hardware implementations, which realize the ideal quantum computational model formalized in a quantum Turing machine [8], often use the universal Clifford+T set $\{S, H, CX, T\}$ [20, 2, 42]. Moreover, better optimal synthesis solutions can have significant implications for quantum complexity theory, such as reducing the stabilizer rank of magic gadgets [31], and relating classical and quantum resources [11, 12]. However, exact synthesis is not always feasible due to the discrete nature of this gate set, often necessitating approximation techniques to balance precision and circuit depth [30]. The computational complexity of synthesis is formidable, with classical approaches exhibiting doubly exponential time in the worst case [33], underscoring the need for scalable and efficient methods.

© Dekel Zak, Jingyi Mei, Jean-Marie Lagniez, and Alfons Laarman;
licensed under Creative Commons License CC-BY 4.0

Classical methods, like decision diagram [53, 13, 50], tree-automata [16], and SAT [48] have proven to be highly effective in analyzing quantum circuits. Recent advances have highlighted weighted model counting, or #SAT, as a powerful tool for addressing hard problems in quantum circuit analysis, including simulation [38] and equivalence checking [39]. These methods leverage off-the-shelf solvers to tackle the exact versions of these problems, known as #P-complete [18]. Building on this promise, we explore whether #SAT can be harnessed for the universal quantum circuit synthesis problem – a challenge that also requires approximation since discrete universal gate sets cannot exactly implement arbitrary quantum operators. The relevant complexity class here is QMA, as approximate circuit equivalence checking is complete for it [26]. Moreover, the Solovay-Kitaev theorem [17] guarantees that any unitary can be ϵ-approximated with a $\mathsf{polylog}(\frac{1}{\epsilon})$-depth circuit.

This work presents a novel reduction of quantum circuit synthesis to maximum model counting, focusing on depth-optimal and approximate synthesis into the Clifford+T gate set. To achieve this, we overcome two obstacles. First, to enable reduction to the maximum weighted model counting, we show how equivalence checking can be performed with a single model counting call, contrasting previous approaches [39] which required linearly many calls. In the process, we provide two new ways of encoding equivalence checking. Second, we show how the fidelity (a measure of circuit similarity) between two circuits can be computed by weighted model counting, generalizing our exact equivalence checking methods to support approximate equivalence checking. Our approach then encodes the synthesis task as a weighted conjunctive normal form (CNF) formula, where satisfying assignments correspond to valid gate sequences and weights reflect approximation fidelity. We demonstrate that this reduction enables both exact and approximate synthesis, with applications to circuit optimization. We provide an open-source implementation, called `Quokka#-syn`, to validate our method.

The scalability of quantum circuit synthesis remains a critical bottleneck, with existing methods struggling to handle large qubit counts or gate depths without resorting to corner-case omissions [37, 44]. We experimentally evaluate our method to compare the different encodings and test its scalability. In addition, we compare it against a state-of-the-art tool focusing on depth-optimal approximate and exact synthesis. While the comparison has limitations, it shows that our #SAT-based approach has merit and can offer improvement over existing methods. However, it falls short of fully resolving the synthesis problem's inherent complexity. We identify multiple avenues for improvement, such as enhancing maximum model counters with support for incremental solving. By laying this groundwork, we pave the way for integrating advanced classical solvers into the quantum computing toolkit, advancing the practical realization of quantum algorithms.

2 Preliminaries

2.1 Quantum Computing

In this section, we give the necessary notions and notations on quantum computing. For an entailed explanation, for example on tensor product ($\otimes$), see [42].

Quantum states. Let $\mathcal{H}^{\otimes n}$ be the 2^n-dimensional Hilbert space. An n-qubit quantum state, denoted as $|\varphi\rangle$ using Dirac notation, is a column vector $\left[\alpha_{00\ldots00}, ..., \alpha_{11\ldots11}\right]^T$ in $\mathcal{H}^{\otimes n}$, where $|\alpha_b|^2 \in [0,1]$ are *amplitudes*, satisfying: $|\alpha_{00\ldots00}|^2 + ... + |\alpha_{11\ldots11}|^2 = 1$. Its complex adjoint $\langle\varphi|$ is a row vector with conjugated entries: $\langle\varphi| = |\varphi\rangle^\dagger = \left[\alpha^*_{00\ldots00}, ..., \alpha^*_{11\ldots11}\right]$, therefore

$\langle\varphi|\varphi\rangle = 1$. A quantum state vector can be decomposed in the *computational basis*, written as $|\varphi\rangle = \sum_{b\in\{0,1\}^n} \alpha_b |b\rangle$, where $|b\rangle$ is a *computational basis state* defined as a vector with all entries setting to 0 except at index b setting to 1.

Another way to represent a state $|\varphi\rangle$ is as a *density matrix* $\rho = |\varphi\rangle\langle\varphi|$. The *trace* of a density matrix is 1, denoted as $\mathrm{tr}(\rho) = 1$, where trace is defined as the sum of the diagonal elements of a matrix. A density matrix can be decomposed in the *Pauli basis* as in Equation 1. To understand Pauli basis decomposition, we introduce *Pauli matrices* and *Pauli strings*. The Pauli matrices are $I = \left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right], X = \left[\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right], Y = \left[\begin{smallmatrix}0&-i\\i&0\end{smallmatrix}\right], Z = \left[\begin{smallmatrix}1&0\\0&-1\end{smallmatrix}\right]$. An n-qubit *Pauli string* $\mathcal{P}$ is a parallel composition of n Pauli gates, such that $\mathcal{P} \in \{I, X, Y, Z\}^{\otimes n}$. For instance, $X \otimes Z \otimes I$ is a three-qubit Pauli string. It is worth noting that while a density matrix ρ may contain complex numbers, the basis *coefficients* β_i in Equation 1 are all real numbers [39]. Throughout this paper, we denote $[n] = \{0, ..., n-1\}$.

$$\rho = \sum_{j\in[4^n]} \beta_j \cdot \mathcal{P}_j \quad \text{for the Pauli strings} \quad \mathcal{P}_j \in \{I, X, Y, Z\}^{\otimes n} \tag{1}$$

Quantum gates. An n-qubit quantum gate G can be expressed by a $2^n \times 2^n$ unitary matrix U, i.e., $U^\dagger \cdot U = U \cdot U^\dagger = I^{\otimes n}$. A single qubit quantum gate U operating on qubit $j \in [n]$ can be represented as $U_j = I^{\otimes j} \otimes U \otimes I^{\otimes n-j-1}$. Updating a quantum state in vector form $|\varphi\rangle$ is done by matrix-vector multiplication, i.e. $|\psi\rangle = U|\varphi\rangle$. Applying a unitary U to a density matrix $\rho = |\varphi\rangle\langle\varphi|$ should be done through *conjugation*, i.e.: $U\rho U^\dagger = U|\varphi\rangle\langle\varphi|U^\dagger = |\psi\rangle\langle\psi|$ for $|\psi\rangle = U|\varphi\rangle$. We consider the well-known universal gate set Clifford $\{S, H, CX\} + T$ and the gates *Toffoli* and CH, which are defined as follows:

$$H = 1/\sqrt{2}\left[\begin{smallmatrix}1&1\\1&-1\end{smallmatrix}\right],\ S = \left[\begin{smallmatrix}1&0\\0&i\end{smallmatrix}\right],\ CX = \left[\begin{smallmatrix}I&\mathbf{0}\\\mathbf{0}&X\end{smallmatrix}\right],\ T = \left[\begin{smallmatrix}1&0\\0&i\end{smallmatrix}\right],\ \mathit{Toffoli} = \left[\begin{smallmatrix}I_6&\mathbf{0}\\\mathbf{0}&X\end{smallmatrix}\right],\ CH = \left[\begin{smallmatrix}I&\mathbf{0}\\\mathbf{0}&H\end{smallmatrix}\right]$$

where I_k is a k-dimensional identity matrix and $\mathbf{0}$ is the all zero matrix. We ignore the index when the dimension is 2, i.e. $I = I_2 = \left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$. We denote a gate set as $\mathcal{G}$. For an n-qubit circuit, let $\mathcal{G}(k) \subseteq \mathcal{G}$ be a subset of gates applied to $k \in [n]$ qubits, such that $\mathcal{G}(1)$ is the set of single-qubit gates, $\mathcal{G}(2)$ is the set of two-qubit gates and so on. For example, if $\mathcal{G} = \{H, T, CX, \mathit{Toffoli}\}$, $\mathcal{G}(1) = \{H, T\}$, $\mathcal{G}(2) = \{CX\}$ and $\mathcal{G}(3) = \{\mathit{Toffoli}\}$. We always assume $I \in \mathcal{G}$, also if not explicitly stated.

Quantum circuits. The evolution of a quantum system is modeled by a *quantum circuit*, a sequence of *quantum gate layers*, or *layers* in short, applied to all qubits at each time step. A *layer* D is a set of gates such that each qubit has at most one gate applied to it. Thus, it contains only mutually parallel gates. For example, a *layer* $D = \{G_i \mid i \in [n]\}$ applies a single-qubit local gate G on each qubit i. For any layer D, we will denote its unitary matrix with U_D. Let $\mathcal{D}_n$ be the set of all possible layers for n qubits. For a *circuit* $\mathcal{C} = (D^0, ..., D^{d-1})$, with $D^i \in \mathcal{D}_n$, we thus have its unitary $U_\mathcal{C} = U_{D^{d-1}} \cdots U_{D^0}$. We define a circuit's *depth* as the minimal number of layers (with parallel gates) that it contains.

Jamiołkowski fidelity The *fidelity* between two quantum states $|\psi\rangle$ and $|\varphi\rangle$ is defined as $\mathrm{Fid}(|\varphi\rangle, |\psi\rangle) \triangleq \mathrm{tr}(|\psi\rangle\langle\psi| \cdot |\varphi\rangle\langle\varphi|) = |\langle\psi|\varphi\rangle|^2$. The fidelity between states can be extended to measure the distance between unitary operators with the help of *Jamiołkowski isomorphism* that maps unitary U on $\mathcal{H}^{\otimes n}$ to a state $|\varphi_U\rangle = (U \otimes I)|\Psi_n\rangle$ on $\mathcal{H}^{\otimes 2n}$, where $|\Psi_n\rangle = \frac{1}{\sqrt{2^n}} \sum_{i\in\{0,1\}^n} |ii\rangle$ and $|ii\rangle$ is short for $|i\rangle \otimes |i\rangle$. Thus, *Jamiołkowski fidelity* [45] between unitary operators can be formally defined as:

$$\mathrm{Fid}_J(U, V) = \mathrm{Fid}(|\varphi_U\rangle, |\varphi_V\rangle) = |\langle\varphi_U|\varphi_V\rangle|^2. \tag{2}$$

In the particular case where $\mathrm{Fid}_J(U,V) = |\langle\varphi_U|\varphi_V\rangle|^2 = 1$, it follows that $U = \lambda V$ with $|\lambda|^2 = 1$. This result arises because $|\langle\varphi_U|\varphi_V\rangle|^2 = 1$ implies $|\varphi_U\rangle = \lambda|\varphi_V\rangle$, and from the definitions of $|\varphi_U\rangle$ and $|\varphi_V\rangle$, we conclude $U = \lambda V$.

2.2 Maximum Weighted Model Counting

Let A be a set of Boolean variables $\{a_1, ..., a_m\}$. A *literal* ℓ is a variable or its negation, e.g., a or $\neg a$ (written as $\bar{a}$). A *clause* is a disjunction of literals $\ell_1 \vee ... \vee \ell_m$. A propositional formula F in *Conjunctive Normal Form* (CNF) is a conjunction of clauses. We write $F(A)$ to indicate that F is defined on the set of variables A. Let $\mathbb{B}$ be $\{0,1\}$ and $\mathbb{R}$ be the set of real numbers. An *assignment* τ maps variables A to $\mathbb{B}$, where $0, 1$ represents *False* and *True* respectively. The satisfiability problem is determining whether an assignment τ to A exists, for which a propositional formula $F(A)$ is true. We denote $SAT(F(A)) \triangleq \{\tau \mid \tau \text{ satisfies } F(A)\}$ as the set of all satisfying assignments for F. The model counting problem is to compute the number of satisfying assignments, denoted as $\#SAT(F) \triangleq |SAT(F)|$.

A weight function $W : A \times \mathbb{B} \to \mathbb{R}$ maps a variable with its true or false assignment, or, viewed alternatively, a literal, to a weight. Given an assignment τ, the weight of this assignment, written $W(\tau)$, is the product of the weight of each variable with its assignment $W(\tau) = \prod_{a\in A} W(a, \tau(a))$. For notational convenience, we write the weights of literals, such that for a variable $a \in A$, we denote $W(a) = W(a, 1)$ and $W(\bar{a}) = W(a, 0)$. If unspecified, the weight of a variable $a \in A$ is assumed to be $W(a) = W(\bar{a}) = 1$. We call such variables *unbiased*. The weighted model counting (WMC) problem asks for the sum of the weights of the satisfying assignments, i.e., $\#SAT_W(F) = \sum_{\tau\in \mathrm{SAT}(F(A))} W(\tau)$.

The maximum weighted model counting problem (MWMC) extends WMC by finding an assignment to a subset of variables that maximizes the weight of the WMC problem (Definition 1). In Section 4, we show that quantum circuit synthesis can be reduced to it.

▶ **Definition 1** (MWMC [5]). *Given a propositional formula $F(A, B)$ over disjoint sets of variables A and B, and a weight function W over $(A \cup B) \times \mathbb{B}$, the MWMC problem is to determine an assignment τ to A that maximizes $\#SAT_W(F(A, B))$.*

For notation, we define an oracle function $Max\#SAT$, which takes the quantified Boolean formula $F(A, B)$ with its weight function W and returns an assignment τ to A that maximizes the weighted model count of the formula F and its maximal weight w_{max}:

$$Max\#SAT_W(F(A, B)) = (\tau(A), w_{max}).$$

2.3 Reducing Quantum Computing to Model Counting

We present two encodings: a computational basis encoding, referred to as **CB**, where the computational basis decomposition of the state vector is encoded directly, and a Pauli basis encoding, referred to as **PB**, where the basis states are the Pauli strings and we encode the density matrix decomposition (see Equation 1). A quantum state is encoded as a set of satisfying assignments where each satisfying assignment with its weight represents a *basis state* with its *amplitude* (in **CB**) or *coefficient* (in **PB**). Here, we briefly introduce both. For details, we refer to [38, 39, 40].

For an n-qubit quantum state $|\varphi\rangle$, we denote its WMC encoding with $F_{|\varphi\rangle}$. Note that in the **PB** basis, quantum states are represented using density matrices, so the corresponding encoding is technically $F_{|\varphi\rangle\langle\varphi|}$. However, we abuse the notation and write $F_{|\varphi\rangle}$ in both bases for simplicity. We reserve propositional variables $\vec{q} = (q_0, ..., q_{n-1})$ in the **CB**, and $\vec{q} = (\vec{q}_0, ..., \vec{q}_{n-1}) = (x_0, z_0, ..., x_{n-1}, z_{n-1})$ in the **PB**. (Here, we let $\overline{xz} \equiv I$, $x\bar{z} \equiv X$, $xz \equiv Y$

and $\overline{x}z \equiv Z$, and $\overline{x_k}z_k \wedge \bigwedge_{i\in[n]\setminus\{k\}} \overline{x_i z_i} \equiv Z_k$, etc, as in [1].) The variables in $\vec{q}$ remain unbiased. In addition, when needed, we introduce auxiliary variables to represent weights. Since the assignment to these auxiliary variables is always determined by the assignment to $\vec{q}$, we often omit these variables, writing $F_{|\varphi\rangle}(\vec{q})$ instead of $F_{|\varphi\rangle}(\vec{q},\vec{u})$. Table 1 gives encoding examples.

Applying a quantum gate maps a quantum state to another. The encoding of a quantum gate G is given by a Boolean function written as $F_G(\vec{q},\vec{q}')$, where $\vec{q}$ is the input state and $\vec{q}'$ is output state after applying G, such that $F_{|\psi\rangle}(\vec{q}') = F_{|\varphi\rangle}(\vec{q}) \wedge F_G(\vec{q},\vec{q}')$ for $|\psi\rangle = G|\varphi\rangle$. As with state encodings, we introduce auxiliary variables when needed to represent weights introduced by the gates, and often omit these variables from the function signature. We give an example of how to encode the gates H, T, and CX in Table 2. A layer D is encoded by conjoining the encodings of local gates, each applied to the variables of the relevant qubits. For example, a two-qubit layer $D = \{H_0, T_1\}$ is encoded by $F_D(\vec{q},\vec{q}') = F_H(\vec{q}_0,\vec{q}'_0) \wedge F_T(\vec{q}_1,\vec{q}'_1)$.

A quantum circuit $\mathcal{C} = (D^0,...,D^{d-1})$ is encoded by reserving variables $\vec{q}^0,...,\vec{q}^d$ for representing the initial, intermediate, and final states, and conjoining all the encoding of layers over these variables, i.e., $F_{\mathcal{C}}(\vec{q}^0,...,\vec{q}^d) = \bigwedge_{j\in[d]} F_{D^j}(\vec{q}^j,\vec{q}^{j+1})$. When we don't need to name the intermediate states $\vec{q}^1,...,\vec{q}^{d-1}$, we will omit them, writing $F_{\mathcal{C}}(\vec{q},\vec{q}') = F_{\mathcal{C}}(\vec{q}^0 = \vec{q},...,\vec{q}^d = \vec{q}')$.

Lemma 2 shows that both WMC encodings allow for the strong simulation of any quantum circuit according to the usual definition of computing output amplitudes or coefficients [28].

▶ **Lemma 2** ([40]). *Given an input state* $|\varphi\rangle = \sum_{b\in\{0,1\}^n} \alpha_b |b\rangle$, *such that* $|\varphi\rangle\langle\varphi| = \sum_{j\in[4^n]} \beta_j \mathcal{P}_j$, *an n-qubit circuit* $\mathcal{C}$, *a computational basis state* $|b\rangle$ $(b \in \{0,1\}^n)$ *and a*

Table 1 State encoding in both bases: The auxiliary variables, marked in gray, depend fully on the unbiased variables $\vec{q}$ (in **CB**) and $\vec{x},\vec{z}$ (in **PB**) representing basis states.

	Comp. Basis (CB)	**Pauli Basis (PB)**	**Auxiliary weights**
Variables	$\vec{q} = (q_0,...,q_{n-1})$	$\vec{q} = (x_0,z_0,...,x_{n-1},z_{n-1})$	h, r, g, w
$\|0\rangle \equiv$	$F_{\|0\rangle}(q) = \overline{q}$	$F_{\|0\rangle}(x,z,g) = F_{\frac{I+Z}{2}} = \overline{x}g$	$W(g) = (1/2)$
$\|-\rangle \equiv$	$F_{\|-\rangle}(q,h,r) = h(r \Leftrightarrow q)$	$F_{\|-\rangle}(x,z,g) = F_{\frac{I-X}{2}} = \overline{z}g$	$W(h) = (1/\sqrt{2})$, $W(r) = (-1)$
[25]: $\|A\rangle \equiv$	$F_{\|A\rangle}(q,h,w) = h(w \Leftrightarrow q)$	$F_{\|A\rangle}(x,z,g,h) = F_{\frac{I}{2}+\frac{X+Y}{\sqrt{2}}}$	$W(w) = \omega$ where $\omega = \sqrt{i} = \frac{i+1}{\sqrt{2}}$
$\|00\rangle \equiv$	$F_{\|00\rangle}(q_0,q_1) = \overline{q_0 q_1}$	$F_{\|00\rangle}(x_0,z_0,x_1,z_1) = \overline{x_0 x_1}$	none
$\|++\rangle \equiv$	$F_{\|++\rangle}(q_0,q_1,g) = g$	$F_{\|++\rangle}(x_0,z_0,x_1,z_1) = \overline{z_0 z_1}$	$W(g) = (1/2)$

Table 2 Gate encoding in both bases using the same weights as in Table 1.

	Comp. Basis (CB)	**Pauli Basis (PB)**
1-qubit unitary $(U1)$	$F_{U1}(\vec{q},\vec{q}')$ where $\vec{q} = (q_0)$	$F_{U1}(\vec{q},\vec{q}')$ where $\vec{q} = (x,z)$
n-qubit unitary (Un)	$F_{Un}(\vec{q},\vec{q}')$ where $\vec{q} = (q_0,...,q_n)$	$F_{Un}(\vec{q},\vec{q}')$ where $\vec{q} = (x_0,z_0,...,x_{n-1},z_{n-1})$
$H = 1/\sqrt{2}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$	$F_H(q,q',r,h) = h \wedge (r \Leftrightarrow qq')$	$F_H(q,q',r) = (r \Leftrightarrow xz) \wedge (z' \Leftrightarrow x) \wedge (x' \Leftrightarrow z)$
$T = \begin{bmatrix}1 & 0\\ 0 & \omega\end{bmatrix}$	$F_T(q,q',w) = (q \Leftrightarrow q') \wedge (w \Leftrightarrow q)$	*given in* [38]
$CX = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}$	$(q'_0 \Leftrightarrow q_0) \wedge (q'_1 \Leftrightarrow (q_1 \oplus q_0))$	*given in* [38]

Pauli string $\mathcal{P}_j \in \{I, X, Y, Z\}^{\otimes n}$*, the following equations hold:*

$$\#SAT_W(F_{|\varphi\rangle}(\vec{q}) \wedge F_{\mathcal{C}}(\vec{q}, \vec{q}') \wedge F_{|b\rangle}(\vec{q}')) = \alpha_b \quad \textit{in} \quad \textbf{CB},$$
$$\#SAT_W(F_{|\varphi\rangle}(\vec{q}) \wedge F_{\mathcal{C}}(\vec{q}, \vec{q}') \wedge F_{\mathcal{P}_j}(\vec{q}')) = \beta_j \quad \textit{in} \quad \textbf{PB}.$$

We emphasize that strong simulation is canonical for quantum complexity classes [52], and therefore generalizes naturally to computing any measurement outcome probability [38] and determining circuit equivalence [39].

3 Problem Statement

The quantum circuit synthesis problem seeks to construct a circuit that implements a given specification, which is provided as either a circuit or a unitary operator. A key component is to determine if a guessed candidate circuit is *equivalent* to the desired specification by checking unitary equivalence, as formalized in Definition 3.

▶ **Definition 3** (Unitary equivalence)**.** *Let* U*,* V *be two* n*-qubit unitaries, Then* U *and* V *are* ϵ*-equivalent, written as* $U \simeq_\epsilon V$*, if and only if the Jamiołkowski fidelity between* U *and* V *is not smaller than* $1 - \epsilon$*, i.e.* $\mathrm{Fid}_J(U, V) \geq 1 - \epsilon$*, where* $\epsilon \in [0, 1]$*.*

In the above definition, we use the Jamiołkowski fidelity defined in Equation 2 to measure the distance between two unitaries. In particular, if and only if $\mathrm{Fid}_J(U, V) = 1$ ($\epsilon = 0$), we have *exact equivalence*, denoted as $U \equiv V$. In this case, U and V are equivalent up to a global phase λ satisfying $|\lambda|^2 = 1$, i.e. $U = \lambda V$. Building on equivalence checking, Problem 4 gives the formal definition of (exact and approximate) quantum circuit synthesis.

▶ **Problem 4** (Quantum circuit synthesis)**.** *Given a specification represented by a circuit* $\mathcal{C}_1$ *in a gate set* $\mathcal{G}_1$ *or unitary* $U_{\mathcal{C}_1}$ *and an accuracy parameter* $\epsilon \in (0, 1]$*, the approximate synthesis problem asks for a depth-optimal quantum circuit* $\mathcal{C}_2$ *in a target gate set* $\mathcal{G}_2$*, such that* $U_{\mathcal{C}_1} \simeq_\epsilon U_{\mathcal{C}_2}$*. For* $\epsilon = 0$*, this is the exact synthesis problem.*

We consider Clifford+T as the elementary target gate set for synthesis because of its universality and importance in error-corrected quantum computing [20, 2]. While unitaries can be synthesized in Clifford+T (with additional ancillas) exactly if and only if their matrix entries belong to $\mathbb{Z}[\frac{1}{\sqrt{2}}, i]$ [22], all unitaries can be synthesized approximately up to an arbitrarily small ϵ [17].

In this work, we consider both exact and approximate synthesis and, for the first time, show that both problems can be reduced to MWMC.

4 Exact Quantum Circuit Synthesis

This section presents our reduction from the exact quantum circuit synthesis problem to the MWMC problem. We first provide an overview of the chosen approach. Recalling Problem 4, the exact synthesis problem has the following components:

- **Input:** A quantum circuit $\mathcal{C}_1$ in gate set $\mathcal{G}_1$, or a unitary $U = U_{\mathcal{C}_1}$, and a finite target gate set $\mathcal{G}_2$.
- **Output:** A depth-optimal quantum circuit $\mathcal{C}_2$ in gate set $\mathcal{G}_2$, such that $U_{\mathcal{C}_2} \equiv U_{\mathcal{C}_1}$, if possible.

The problem can be expressed as exhaustively searching over all possible layers $\mathcal{D}_n$ for each layer in $\mathcal{C}_2$. To achieve depth-optimality, we increment the depth d until the following holds:

$$\exists D^0, ..., D^{d-1} \in \mathcal{D}_n \ : \ U_{\mathcal{C}_1} \equiv U_{\mathcal{C}_2} \text{ where } \mathcal{C}_2 = (D^0, ..., D^{d-1}). \tag{3}$$

In the remainder of this section, we first give the encoding of the input specification. Then we present different encodings for checking the exact equivalence between the input and output. Next, we encode a generic gate layer $\mathcal{D}_n$ and extend the encoding progressively to construct candidate circuits $\mathcal{C}_2$ of increasing depth, as specified in Equation 3. We conclude by showing how MWMC can find a depth-optimal output circuit $\mathcal{C}_2$ using the encoding for generic layers and equivalence checking.

Encoding input specifications. Due to the reversibility of quantum circuits, verifying the equivalence of circuits $\mathcal{C}_1$ and $\mathcal{C}_2$ is reducible to checking if the circuit $\mathcal{C}_2 \cdot \mathcal{C}_1^\dagger$ is equivalent to the identity. Therefore, we encode the input circuit $\mathcal{C}_1$ as its inverse $\mathcal{C}_1^\dagger$, denoted as $F_{\mathcal{C}_1^\dagger}(\vec{q}, \vec{q}')$, as explained in Section 2.1. If the input is an n-qubit unitary operator $U^\dagger$, we can encode directly it as $F_{U^\dagger}(\vec{q}, \vec{q}')$ using weighted auxiliary variables to represent the unique components of the unitary, in the same way we encode individual gates [40].

Verifying exact equivalence. The exact equivalence checking problem is efficiently tackled with WMC in [39], using a so-called *linear encoding*, which requires $2n$ separate WMC calls. In Theorem 5, we extend this encoding to *cyclic encoding* and *linear-cyclic encoding*. These two new encodings solve the problem with a single call to the weighted model counter, *as we require in the proposed synthesis approach.*

▶ **Theorem 5.** *Let $\mathcal{C}$ be an n-qubit circuit, which is encoded by $F_\mathcal{C}$ with the corresponding weight function W. Then, the following four statements are equivalent to each other:*

- *$\mathcal{C} \equiv I^{\otimes n}$ ($\mathcal{C}$ is equivalent to the identity circuit $I^{\otimes n} = I_{2^n}$).*
- *Encoding the circuit $\mathcal{C}$ in* **PB**, *the* ***linear encoding*** *has weighted model count [39]:*

$$\#SAT_W(F_\mathcal{P}(\vec{q}) \wedge F_\mathcal{C}(\vec{q}, \vec{q}') \wedge F_\mathcal{P}(\vec{q}')) = 1 \text{ for all } \mathcal{P} \in \{X_j, Z_j \mid j \in [n]\}. \tag{4}$$

- *Encoding the circuit in either* **CB** *or* **PB**, *the* ***cyclic encoding*** *has weighted model count (this approach can be viewed as checking "overlap" with the identity $I^{\otimes n}$):*

$$\#SAT_W(F_\mathcal{C}(\vec{q}, \vec{q}') \wedge F_{I^{\otimes n}}(\vec{q}, \vec{q}')) = c \text{ with } |c| = 2^n \text{ for } \mathbf{CB} \text{ and } c = 4^n \text{ for } \mathbf{PB}. \tag{5}$$

- *Encoding the circuit $\mathcal{C}$ in* **PB**, *the* ***linear-cyclic encoding*** *has weighted model count:*

$$\#SAT_W\Big(\bigvee_{\mathcal{P} \in \{X_j, Z_j \mid j \in [n]\}} F_\mathcal{P}(\vec{q}) \wedge F_\mathcal{C}(\vec{q}, \vec{q}') \wedge F_{I^{\otimes n}}(\vec{q}, \vec{q}')\Big) = 2n. \tag{6}$$

where $\vec{q}, \vec{q}'$ are Boolean variables encoding the initial and final quantum state, respectively. Note that $F_{I^{\otimes n}}(\vec{q}, \vec{q}') = \bigwedge_{i \in [n]}(q_i \Leftrightarrow q_i')$, where $q_i \Leftrightarrow q_i'$ is shorthand for $(x_i \Leftrightarrow x_i') \wedge (z_i \Leftrightarrow z_i')$ in the Pauli basis.

We give a detailed proof in Appendix A.1, and illustrate the theorem in Example 6.

▶ **Example 6.** Consider two circuits $\mathcal{C}_1 = (S)$ and $\mathcal{C}_2 = (T, T)$. To check their equivalence, we first encode the circuits $\mathcal{C} = \mathcal{C}_2 \cdot \mathcal{C}_1^\dagger$ and I:

$$F_\mathcal{C}(q^0, q^3) := F_{S^\dagger}(q^0, q^1) \wedge F_T(q^1, q^2) \wedge F_T(q^2, q^3) \quad \text{and} \quad F_I(q, q') := q \Leftrightarrow q'$$

Then we check if $\mathcal{C} \equiv I$ as follows:

- Linear encoding: $\#SAT_W(F_\mathcal{P}(q^0) \wedge F_\mathcal{C}(q^0, q^3) \wedge F_\mathcal{P}(q^3)) = 1$ for $\mathcal{P} \in \{X, Z\}$ in **PB**.
- Cyclic encoding: $\#SAT_W(F_\mathcal{C}(q^0, q^3) \wedge F_I(q^0, q^3)) = c$ with $|c| = 2$ (**CB**) or $c = 4$ (**PB**).
- Linear-cyclic encoding: $\#SAT_W((F_X \vee F_Z) \wedge F_\mathcal{C} \wedge F_I) = 2$ in **PB**. ⌟

Encoding synthesis layers. Building up the output circuit is done by incrementally finding a sequence of gate layers implementing the input specification, as specified in Equation 3. To explore the space of possible gate layers, we encode the complete set $\mathcal{D}_n$ by introducing gate-selecting variables. Once these variables are fixed, they uniquely determine a specific gate layer within the set.

Given the target gate set $\mathcal{G}$, we first consider single-qubit gates $\mathcal{G}(1) \subseteq \mathcal{G}$. For each gate $G \in \mathcal{G}(1)$, we define the Boolean variable $p_{G,i}$ for each qubit $i \in [n]$, where $p_{G,i}$ is true if and only if G_i is included in the layer. Thus, we can encode the single-qubit gates of the layer as:

$$F_{\mathcal{G}(1)}(\vec{q}, \vec{q}', \vec{p}(1)) = \bigwedge_{i\in[n]} \bigwedge_{G\in\mathcal{G}(1)} (p_{G,i} \Rightarrow F_G(q_i, q_i')), \tag{7}$$

where the variables $\vec{q}$ and $\vec{q}'$ encode the states before and after the layer respectively, and $\vec{p}(1) = \{p_{G,i} \mid G \in \mathcal{G}(1), i \in [n]\}$ are the single-qubit gate-selecting variables.

Similarly, for a two-qubit gates $\mathcal{G}(2) \subseteq \mathcal{G}$, we introduce quadratically many variables for all combinations of the two qubits: $\vec{p}(2) = \{p_{G,i,j} \mid G \in \mathcal{G}(2), i, j \in [n], i \neq j\}$. Then encoding is:

$$F_{\mathcal{G}(2)}(\vec{q}, \vec{q}', \vec{p}(2)) = \bigwedge_{i,j\in[n],j\neq i} \bigwedge_{G\in\mathcal{G}(2)} (p_{G,i,j} \Rightarrow F_{G_{i,j}}(\vec{q}, \vec{q}')), \tag{8}$$

Since we consider the universal gate set Clifford+T, which only includes single-qubit gates and two-qubit gates, the set of gate-selecting variables is given by $\vec{p} = \vec{p}(1) \cup \vec{p}(2)$ of the synthesis layer. It is worth noting that the identity operator I is included when considering multi-qubit circuits for completeness. For example when considering $\{CX, H, T\}$, we define the set of single-qubit gates as $G(1) = \{H, T, I\}$.

A layer is valid only when applying exactly one gate to each qubit. For example, $p_{H,1}$ and $p_{T,1}$ should not be true simultaneously. Therefore, we define $\texttt{EXO}(V) = \bigvee_{v\in V} v \wedge \bigwedge_{u,v\in V,u\neq v} (\overline{v} \vee \overline{u})$, a constraint ensuring that exactly one variable in the set V is true. Now, for each qubit $i \in [n]$, we apply:

$$F_{\texttt{EXO}}(\vec{p}) = \bigwedge_{i\in[n]} \texttt{EXO}(\vec{p}_i), \text{ where } \vec{p}_i = \{p_{G,i} \mid G \in \mathcal{G}(1)\} \cup \{p_{G,i,j}, p_{G,j,i} \mid G \in \mathcal{G}(2), j \in [n]\}. \tag{9}$$

Combining the three constraints in Equation 7, Equation 8 and Equation 9 gives us the layer encoding:

$$F_{\mathcal{D},\mathcal{G}}(\vec{q}, \vec{q}', \vec{p}) = F_{\mathcal{G}(1)}(\vec{q}, \vec{q}', \vec{p}(1)) \wedge F_{\mathcal{G}(2)}(\vec{q}, \vec{q}', \vec{p}(2)) \wedge F_{\texttt{EXO}}(\vec{p}) \tag{10}$$

If there are gates in the target gate set $\mathcal{G}$ applied to more than two qubits, the encoding is expanded accordingly.

▶ **Example 7.** The encoding of a layer in the universal gate set $\mathcal{G} = \{CX, H, T\}$ is as follows:

$$\begin{aligned} F_{\mathcal{D},\mathcal{G}}(\vec{q}, \vec{q}', \vec{p}) = &\bigwedge_{i\in[n]} \big(p_{H,i} \Rightarrow F_H(q_i, q_i') \wedge p_{T,i} \Rightarrow F_T(q_i, q_i') \wedge p_{I,i} \Rightarrow F_I(q_i, q_i')\big) \\ &\wedge \bigwedge_{i,j\in[n],j\neq i} \big(p_{CX,i,j} \Rightarrow F_{CX_{i,j}}((q_i, q_j), (q_i', q_j'))\big) \quad \wedge \bigwedge_{i\in[n]} \texttt{EXO}(\vec{p}_i), \end{aligned}$$

where $\vec{p} = \bigcup_{i\in[n]} \vec{p}_i$ and $\vec{p}_i = \{p_{I,i}, p_{H,i}, p_{T,i}\} \cup \bigcup_{j\in[n],j\neq i} \{p_{CX,i,j}, p_{CX,j,i}\}$.

To encode multiple layers, we reserve state variables $\vec{q}^t$ and gate-selecting variables $\vec{p}^t$ for the t-th layer and substitute the above formula with the variables for each time step, i.e. $F_{\mathcal{D},\mathcal{G}}(\vec{q}^t, \vec{q}^{t+1}, \vec{p}^t)$. Encoding d layers is given by a conjunction of the encodings:

$$\bigwedge_{t\in[d]} F_{\mathcal{D},\mathcal{G}}(\vec{q}^t, \vec{q}^{t+1}, \vec{p}^t). \tag{11}$$

Encoding exact synthesis. After giving the encoding of the input specification $U_{\mathcal{C}_1}$, the encoding of exact equivalence checking, and the encoding of gate layers, we now show the encoding of exact synthesis. The main idea is to determine a minimal sequence of gate layers $\mathcal{C}_2 = (D^1, ..., D^d)$ such that $U_{\mathcal{C}_2} \equiv U_{\mathcal{C}_1}$, which can be represented with the cyclic encoding as:

$$Syn_{\mathbf{C},\mathbf{B},\mathcal{G},\mathcal{C}_1,d}(P,Q) \quad = \quad \overbrace{F_{U^\dagger_{\mathcal{C}_1}}(\vec{q},\vec{q}^0) \wedge \bigwedge_{t\in[d]} F_{\mathcal{D},\mathcal{G}}(\vec{q}^t, \vec{q}^{t+1}, \vec{p}^t)}^{U_{\mathcal{C}_2}U^\dagger_{\mathcal{C}_1}} \quad \wedge \quad \overbrace{(\vec{q} \Leftrightarrow \vec{q}^d)}^{F_{I^{\otimes n}}}, \tag{12}$$

where $\mathbf{C}$ denotes the encoding is cyclic, $\mathbf{B}$ denotes the chosen basis (which can be either $\mathbf{PB}$ or $\mathbf{CB}$), $P = \bigcup_{t\in[d]} \vec{p}^t$ is the set of gate-selecting variables, and $Q = \vec{q} \cup \bigcup_{t\in[d+1]} \vec{q}^t \cup \vec{u}^t$ is the set of all state variables $\vec{q}$ and auxiliary variables $\vec{u}$. For a linear-cyclic checking (denoted by $\mathbf{LC}$)(only for $\mathbf{PB}$), the encoding is done by adding constraints to the initial state variables

$$Syn_{\mathbf{LC},\mathbf{PB},\mathcal{G},\mathcal{C}_1,d}(P,Q) = \bigvee_{\mathcal{P}\in\{X_j,Z_j|j\in[n]\}} F_{\mathcal{P}}(\vec{q}) \wedge Syn_{\mathbf{C},\mathbf{PB},\mathcal{G},\mathcal{C}_1,d}(P,Q). \tag{13}$$

Proposition 8 shows how the exact synthesis problem is reduced to the MWMC problem. It essentially reduces the problem of finding an assignment for the gate-selecting variables that maximizes the weighted model count of the above encodings in Equation 12 and Equation 13. The correctness of Proposition 8 follows directly from the cyclic and linear-cyclic encodings of Theorem 5, which concludes this section.

▶ **Proposition 8.** *Given a quantum circuit $\mathcal{C}_1$ (or its unitary $U_{\mathcal{C}_1}$) and an integer d, there exists a d-depth circuit $\mathcal{C}_2$ such that $U_{\mathcal{C}_1} \equiv U_{\mathcal{C}_2}$ iff*

$$Max\#SAT_W\big(Syn_{\mathbf{E},\mathbf{B},\mathcal{G},\mathcal{C}_1,d}(P,Q)\big) = (c, \tau(P)),$$

where for $\mathbf{E} = \mathbf{LC}$ (linear-cyclic encoding) and $\mathbf{B} = \mathbf{PB}$ we have that $c = 2n$, and for $\mathbf{E} = \mathbf{C}$ (cyclic encoding) if $\mathbf{B} = \mathbf{PB}$ we have that $c = 4^n$ and if $\mathbf{B} = \mathbf{CB}$ we have that $|c| = 2^n$. When the maximal value is achieved, the output circuit $\mathcal{C}_2$ can be directly determined from the satisfying assignment $\tau(P)$.

▶ **Example 9.** Let us consider the circuit $\mathcal{C}_1 = (S)$ and the target gate set $\mathcal{G} = \{CX, H, T\}$. Since it is a single-qubit circuit, the encoding of one general layer is, where for gate G we denote $F_G = F_G(q_0, q_0')$

$$F_{\mathcal{D},\mathcal{G}}(\vec{q} = (q_0), \vec{q}' = (q_0'), \vec{p}) = \big((p_{H,0} \Rightarrow F_H) \wedge (p_{T,0} \Rightarrow F_T) \wedge (p_{I,0} \Rightarrow F_I)\big) \wedge \bigwedge_{i\in[n]} \texttt{EXO}(\vec{p}),$$

We synthesize the circuit by calling $Max\#SAT_W$ to find an assignment for the gate-selection variables such that the function achieves maximal value:

$$Max\#SAT_W(Syn_{\mathbf{LC},\mathbf{PB},\mathcal{G},\mathcal{C}_1,1}(P,Q)) = (0.854,\ \tau_1 = \{p^0_{H,0} \leftarrow 0, p^0_{T,0} \leftarrow 1, p^0_{I,0} \leftarrow 0\})$$
$$Max\#SAT_W(Syn_{\mathbf{LC},\mathbf{PB},\mathcal{G},\mathcal{C}_1,2}(P,Q)) = (1,\ \tau_2 = \{p^0_{H,0} \leftarrow 0, p^0_{T,0} \leftarrow 1, p^0_{I,0} \leftarrow 0,$$
$$p^1_{H,0} \leftarrow 0, p^1_{T,0} \leftarrow 1, p^1_{I,0} \leftarrow 0\})$$

From the assignment τ_2, we get the synthesized circuit $\mathcal{C}_2 = (T, T)$.

5 Approximate Quantum Circuit Synthesis

In this section, we focus on approximate synthesis as given in Problem 4. The input and output of the problem are now as follows:

- **Input:** A quantum circuit $\mathcal{C}_1$ in a gate set $\mathcal{G}_1$ or directly the unitary $U_{\mathcal{C}_1}$, a finite target gate set $\mathcal{G}_2$, and an error bound $\epsilon \in (0,1]$.
- **Output:** A depth-optimal quantum circuit $\mathcal{C}_2$ in gate set $\mathcal{G}_2$ such that $\mathcal{C}_2 \simeq_\epsilon \mathcal{C}_1$.

Since synthesis relies on equivalence checking, our first aim is to lift the latter to approximate equivalence checking. Theorem 10 shows that the cyclic encoding in both **PB** and **CB** computes the Jamiołkowski fidelity between two circuits, thus both encodings can determine approximate equivalence checking based on Definition 3.

▶ **Theorem 10.** *Given two unitary matrices U and V on an n-qubit Hilbert space $\mathcal{H}^{\otimes n}$, the Jamiołkowski fidelity can be computed by*

$$\mathrm{Fid}_J(U,V) \quad = \quad \begin{cases} \frac{1}{4^n} \cdot \#SAT_W(F_{U^\dagger V}(\vec{q},\vec{q}') \wedge (\vec{q} \Leftrightarrow \vec{q}')) & \text{in} \quad \mathbf{PB} \\ \frac{1}{4^n} \cdot |\#SAT_W(F_{U^\dagger V}(\vec{q},\vec{q}') \wedge (\vec{q} \Leftrightarrow \vec{q}'))|^2 & \text{in} \quad \mathbf{CB} \end{cases}.$$

Proof. To compute Jamiołkowski fidelity of two given n-qubit circuits U and V on $\mathcal{H}^{2^n}$, one takes the maximally entangled state $|\Psi_n\rangle$ on $\mathcal{H}^{2^n} \otimes \mathcal{H}^{2^n}$ as input and compute the fidelity between the output states $(U \otimes I^{\otimes n})|\Psi_n\rangle$ and $(V \otimes I^{\otimes n})|\Psi_n\rangle$. We first prove this in the **CB** and then move to **PB**.

In **CB**, the Jamiołkowski fidelity is given by

$$\begin{aligned} \mathrm{Fid}_J(U,V) = \mathrm{Fid}((U \otimes I^{\otimes n})|\Psi_n\rangle, (V \otimes I^{\otimes n})|\Psi_n\rangle) &= \left|\langle\Psi_n|\,(U^\dagger \otimes I^{\otimes n})(V \otimes I^{\otimes n})\,|\Psi_n\rangle\right|^2 \\ = \left|\frac{1}{2^n} \cdot \sum_{b\in\{0,1\}^n} \sum_{b'\in\{0,1\}^n} \langle i|\,U^\dagger V\,|j\rangle \cdot \langle b|\,I^{\otimes n}\,|b'\rangle\right|^2 &= \left|\frac{1}{2^n} \cdot \sum_{b\in\{0,1\}^n} \langle b|\,U^\dagger V\,|b\rangle\right|^2, \end{aligned}$$

Based on Lemma 2, it holds that:

$$\begin{aligned} \#SAT_W(F_{U^\dagger V}(\vec{q},\vec{q}') \wedge (\vec{q} \Leftrightarrow \vec{q}')) &= \sum_{b\in\{0,1\}^n} \#SAT_W(F_{|b\rangle}(\vec{q}) \wedge F_{U^\dagger V}(\vec{q},\vec{q}') \wedge F_{|b\rangle}(\vec{q}')) \\ &= \sum_{b\in\{0,1\}^n} \langle b|\,U^\dagger V\,|b\rangle, \end{aligned}$$

giving us that $\mathrm{Fid}_J(U,V) = \frac{1}{4^n}\,|\#SAT_W(F_{U^\dagger V}(\vec{q},\vec{q}') \wedge (\vec{q} \Leftrightarrow \vec{q}'))|^2$ as we wanted to show.

Moving to **PB**, we first represent $|\Psi_n\rangle$ and the Jamiołkowski fidelity in the Pauli basis, and then explain the encoding. The density operator of the maximally entangled state can be decomposed in the Pauli basis as: $|\Psi\rangle\langle\Psi| = \frac{1}{2^n}\sum_{i\in[n],j\in[n]} |ii\rangle\langle jj| = \frac{1}{4^n}\sum_{\mathcal{P}_i\in\{X,Y,Z,I\}^{\otimes n}} \mathcal{P}_i \otimes \mathcal{P}_i^T$, which is shown in [15]. The Jamiołkowski fidelity can also be defined in the Pauli basis as:

$$\begin{aligned} \mathrm{Fid}_J(U^\dagger V, I) &= \mathrm{Tr}\big((U^\dagger V \otimes I)\,|\Psi\rangle\langle\Psi|\,(V^\dagger U \otimes I)\,|\Psi\rangle\langle\Psi|\big) \\ &= \frac{1}{16^n} \sum_{j,k\in[4^n]} \mathrm{Tr}\big(U^\dagger V \mathcal{P}_j V^\dagger U \mathcal{P}_k \otimes \mathcal{P}_j^T \mathcal{P}_k^T\big) = \frac{1}{8^n} \sum_{j\in[4^n]} \mathrm{Tr}\big(U^\dagger V \mathcal{P}_j V^\dagger U \mathcal{P}_j\big) \end{aligned} \tag{14}$$

Based on Proposition 1 in [39], each of the summand $\frac{1}{2^n} \cdot \mathrm{Tr}\big(U^\dagger V \mathcal{P}_j V^\dagger U \mathcal{P}_j\big)$ can be computed by the weighted model counting of $F_{\mathcal{P}_j}(\vec{q}) \wedge F_{U^\dagger V}(\vec{q},\vec{q}') \wedge F_{\mathcal{P}_j}(\vec{q}')$, or equally, $F_{\mathcal{P}_j}(\vec{q}) \wedge F_{U^\dagger V}(\vec{q},\vec{q}') \wedge (\vec{q} \Leftrightarrow \vec{q}')$. To compute the summation, one should go over all possible Pauli strings, which is equal to setting the variables in $\vec{q}$ to be free. Hence, the Jamiolkowski fidelity is obtained by $\mathrm{Fid}_J(U,V) = \frac{1}{4^n}\#SAT_W(F_{U^\dagger V}(\vec{q}^0,\vec{q}'^n) \wedge (\vec{q}^0 \Leftrightarrow \vec{q}'^n))$. ◀

We can thus reuse the cyclic encoding from Equation 12 to encode approximate synthesis.

▶ **Proposition 11.** *Given a quantum circuit $\mathcal{C}_1$, an integer d and an error bound $\epsilon \in (0, 1]$, there exists a d-depth circuit $\mathcal{C}_2$ such that $U_{\mathcal{C}_2} \simeq_\epsilon U_{\mathcal{C}_1}$ iff $Max\#SAT_W\left(Syn_{\mathbf{C},\mathbf{B},\mathcal{G},\mathcal{C},d}(P, Q)\right) = (c, \tau(P))$, such that $\frac{1}{2^n}|c| > 1 - \epsilon$ if $\boldsymbol{B} = \mathbf{CB}$ and $\frac{1}{4^n}c > 1 - \epsilon$ if $\boldsymbol{B} = \mathbf{PB}$. In that case, $\mathcal{C}_2$ can be determined by the satisfying assignment $\tau(P)$.*

Like in exact synthesis, to get the depth-optimal ϵ-approximate circuit, we apply the above Proposition 11 for an increasing depth d. Example 12 demonstrates approximate synthesis.

▶ **Example 12.** Consider the circuit $\mathcal{C}_1 = (R_Z(\frac{\pi}{8}))$ with $\epsilon = 0.05$. Synthesizing one gate layer, we get the result as

$$Max\#SAT_W(Syn_{\mathbf{B},\mathbf{PB},\mathcal{G},\mathcal{C},1}(P, Q)) = (3.848, \{p^0_{H,0} \leftarrow 0, p^0_{T,0} \leftarrow 1, p^0_{I,0} \leftarrow 0\}),$$

which determine the output circuit as $\mathcal{C}_2 = (T)$ and the fidelity $\mathrm{Fid}_J(U_{\mathcal{C}_1}, U_{\mathcal{C}_2}) = \frac{1}{4} \times 3.848 = 0.962 > 1 - \epsilon$. Thus, the synthesis procedure stops, and the output circuit is (T).

6 Related Work

Clifford circuit synthesis. Synthesis of Clifford circuits is substantially simpler than that of universal quantum circuits due to the ability to exploit the algebraic structure of the symplectic group, which significantly constrains the search space. Any Clifford operation on n qubits can be efficiently represented by a $2n \times 2n$ symplectic matrix over $\mathbb{F}_2$, rather than the exponentially larger $2^n \times 2^n$ unitary matrix typically required for general quantum operations [14]. Building on this, Maslov and Roetteler [36] employ the Bruhat decomposition of the symplectic group to generate shorter Clifford circuits. Similarly, Rengaswamy et al. [46] develop Clifford synthesis algorithms via symplectic geometry, targeting logical-level Clifford operations with an emphasis on practical implementations on physical qubits. While these approaches produce efficient Clifford circuits, they do not guarantee optimality in terms of depth or gate count. To address this limitation, Schneider et al. [48] reformulate Clifford synthesis as a satisfiability problem. This encoding enables the use of SAT and MaxSAT solvers [10] to identify optimal Clifford circuits for a fixed depth, offering a rigorous method to achieve minimality.

Exact Clifford+T circuit synthesis. In error-corrected quantum computing, the relevant universal gate is Clifford+T. There are many works considering the exact synthesis of quantum circuits in the gate set Clifford+T[3, 37, 22, 23, 29, 43], i.e., the desired specification is realized without any rounding errors. Approaches like [22, 43] synthesize the unitary matrix representing the specification in a local fashion, i.e., column by column. However, they do not give the optimal solution and leave significant room for improvement. To achieve optimality, the meet-in-the-middle algorithm [37, 23] performs an exhaustive search over the space of all Clifford+T circuits up to a given depth.

Approximate Clifford+T circuit synthesis. Since not all unitaries can be implemented exactly in the Clifford+T gate set, other works have focused on synthesizing circuits under different approximation metrics. Most of these works [30, 47, 49, 19] focus on single-qubit operators, especially rotation gates, while [21, 44] consider multi-qubit operators.

7 Experimental Evaluation

7.1 Implementation

To evaluate and test our proposed method, we implemented it in a tool called `Quokka#-syn`, as part of the open source toolkit `Quokka#`*. In addition to the core method described in the previous sections, we also implemented several optimization rules. Since no existing tool can solve our problem, we extended the MWMC solver d4Max [5] with support for negative and complex weights. The following paragraphs describe these two components in more detail.

Encoding optimization. While the encoding in Equation 11 allows for any circuit of depth d, many encoded circuits are redundant as they fall in the same equivalence class. For example, two consecutive Hadamard gates can be reduced to the identity. So if the H is applied to the j-th qubit at depth t, we can safely exclude the case where another H is applied to the same qubit at depth $t+1$. In the encoding, this corresponds to enforcing that if $p_{H_{j,t}}$ is set to 1, then $p_{H_{j,t+1}}$ must be 0, which can be encoded as $\bigwedge_{k\in[d-1]}\bigwedge_{i\in[n]}(\overline{p}^k_{H,i} \vee \overline{p}^{k+1}_{H,i})$. We apply similar reasoning to patterns like $T^8 = I$ and $CX_{i,j}CX_{i,j} = I^{\otimes 2}$, which are never part of an optimal circuit. Eliminating such cases does not affect optimality and helps reduce the search space. We introduce additional encodings to prune such redundant structures, as detailed in Appendix A.2.

d4Max extension. To support the encodings, we first extended d4Max to support negative weights (for **PB**) and complex weights (for **CB**). For number representation, we use arbitrary precision arithmetic from the GMP library, as in the original version of d4Max. Because of the negative weights, computing an upper bound for a sub-formula becomes more complex. In the original version of d4Max, upper bounds can be easily computed by assuming the formula is a tautology. This allows the solver to prune branches of the search tree when it is clear that no better value can be reached than the current best solution. As this is non-trivial to resolve in the presence of negative weights, we turned this optimization off. Another removed feature is the ability to compute intermediate approximations before processing all connected components. As a result of these changes, the current version is slower at providing intermediate solutions compared to the original version.

7.2 Performance

In this section, we explore the performance of our method as implemented in `Quokka#-syn`. Our implementation synthesizes circuits with the gate set $\{H, CX, T, T^\dagger\}$. This set is also universal since $S = TT$, where we replace the S gate with the $T^\dagger$ gate.

We demonstrate the feasibility and scalability of our method based on two classes of benchmarks for exact synthesis: random circuits on $\{H, CX, T, T^\dagger\}$ and commonly used unitary operators such as the CH and *Toffoli* gates. We use randomly generated circuits as benchmarks, as they are the standard method for evaluating performance since they represent a hard case, which is also employed in practice to prove quantum supremacy [4].

Since our method guarantees a depth-optimal result, we compare with the state-of-the-art method `mitms` [3], which targets the same task. Our experiments were run on a single-core AMD Ryzen 9 7900X Processor and 64 GB of memory.

*`https://github.com/System-Verification-Lab/Quokka-Sharp`

Table 3 Synthesis benchmarks for random circuits. **CB** and **PB** indicate cyclic encoding in each basis; L**PB** uses linear-cyclic encoding in **PB**. Memory is reported by d4max; for short runtimes, usage is below 2 GB and denoted by < 2. Averages include $\pm$ standard deviation. **-** indicates all 10 samples failed; $\circ$ marks untested depths due to low prior success.

#Qb	Depth	Rate			Time (s)			Memory (GB)		
		CB	**LPB**	**PB**	**CB**	**LPB**	**PB**	**CB**	**LPB**	**PB**
2	1	1.0	1.0	1.0	0.05 ± 0.00	0.05 ± 0.00	0.05 ± 0.00	< 2	< 2	< 2
	2	1.0	1.0	1.0	0.07 ± 0.00	0.08 ± 0.00	0.08 ± 0.00	< 2	< 2	< 2
	3	1.0	1.0	1.0	0.11 ± 0.01	0.11 ± 0.01	0.14 ± 0.02	< 2	< 2	< 2
	4	1.0	1.0	1.0	0.29 ± 0.09	0.22 ± 0.07	0.40 ± 0.10	< 2	< 2	< 2
	5	1.0	1.0	1.0	2.46 ± 1.51	1.46 ± 0.72	4.53 ± 3.16	2.13 ± 0.03	2.09 ± 0.00	2.11 ± 0.02
	6	1.0	1.0	1.0	21.25 ± 12.54	14.26 ± 10.33	55.69 ± 42.54	2.46 ± 0.28	2.16 ± 0.07	2.38 ± 0.27
	7	0.9	0.9	0.3	115.31 ± 64.99	97.28 ± 46.98	206.37 ± 17.36	3.84 ± 1.31	2.48 ± 0.21	3.41 ± 0.54
	8	0.0	0.0	$\circ$	-	-	$\circ$	-	-	$\circ$
3	1	1.0	1.0	1.0	0.05 ± 0.00	0.05 ± 0.00	0.05 ± 0.00	< 2	< 2	< 2
	2	1.0	1.0	1.0	0.08 ± 0.01	0.10 ± 0.01	0.10 ± 0.02	< 2	< 2	< 2
	3	1.0	1.0	1.0	1.43 ± 0.97	0.44 ± 0.24	2.45 ± 1.91	2.11 ± 0.02	< 2	2.11 ± 0.01
	4	1.0	1.0	1.0	51.56 ± 57.11	11.26 ± 4.95	105.40 ± 64.98	3.15 ± 1.25	2.15 ± 0.04	2.87 ± 0.67
	5	0.1	0.7	0.0	145.96	147.37 ± 49.30	-	5.08	3.20 ± 0.53	-
	6	$\circ$	0.0	$\circ$	$\circ$	-	$\circ$	$\circ$	-	$\circ$
4	1	1.0	1.0	1.0	0.05 ± 0.00	0.06 ± 0.00	0.05 ± 0.00	< 2	< 2	< 2
	2	1.0	1.0	1.0	0.37 ± 0.20	0.30 ± 0.14	0.62 ± 0.48	2.09	< 2	2.09 ± 0.01
	3	1.0	1.0	0.5	102.55 ± 85.73	22.65 ± 16.03	85.70 ± 42.76	4.44 ± 2.07	2.26 ± 0.13	2.69 ± 0.34
	4	0.0	0.2	0.0	-	246.80 ± 21.38	-	-	4.51 ± 0.20	-
5	1	1.0	1.0	1.0	0.05 ± 0.00	0.06 ± 0.01	0.06 ± 0.00	< 2	< 2	< 2
	2	1.0	1.0	1.0	2.93 ± 2.17	3.01 ± 1.62	11.85 ± 11.15	2.14 ± 0.04	2.11 ± 0.02	2.18 ± 0.10
	3	0.0	0.1	0.0	-	243.71	-	-	4.67	-
6	1	1.0	1.0	1.0	0.05 ± 0.00	0.12 ± 0.06	0.06 ± 0.00	< 2	< 2	< 2
	2	0.9	1.0	0.7	70.49 ± 82.13	54.47 ± 44.75	72.71 ± 69.85	3.74 ± 2.03	2.69 ± 0.61	2.80 ± 0.81
	3	0.0	0.0	0.0	-	-	-	-	-	-

Exact synthesis. In the first experiment, we compare the performances of the different encodings to determine the best one and assess the scalability of the method, as this determines its effectiveness in practice [3]. We test all three synthesis encodings for circuit inputs: **CB** with the cyclic encoding and **PB** with the cyclic and linear-cyclic encodings.

Random circuits are generated layer by layer, with each layer applying exactly one gate per qubit. Gates are selected uniformly at random from the gate set and assigned to unassigned qubits. Since a random circuit of depth d could be equivalent to a shallower circuit, we retain only those circuits that cannot be synthesized with a depth less than d, until we obtain 10 samples. Using these samples, we then evaluated all three encodings for increasing depths, stopping when fewer than half the circuits are solved within the time limit per instance. We set the time limit to 300 seconds to accommodate a large number of benchmarks. Experiments were run on circuits with 2 to 6 qubits; the 1-qubit case is omitted here as it is evaluated for approximate synthesis below.

The results are shown in Table 3. For each encoding, qubit count, and depth, we report the success rate (the fraction of random circuit samples that can be solved within the time limit), the average runtime of solved cases, and the average memory needed by d4Max for the solved cases. The results highlight the significant impact of the qubit count and depth on the runtime. For every number of qubits, there is a specific depth threshold where problem-solving becomes challenging; before reaching this point, the success rate is perfect. We observe that the linear encoding consistently demonstrates the best performance among the three encoding methods.

Table 4 Experimental results on synthesis for controlled-gates. We present the running time and memory usage reported by d4Max, as the encoding time is less than 0.01 seconds. We report N/A when d4Max does not give the memory usage, denoted by < 2000. Here '-' denotes timeout cases except for the case `mitms` with 2-qubit and 8-depth, where the data is not given in [3].

#Qubits \Depth				1	2	3	4	5	6	7	8
2	`Quokka#-syn`	Cyclic (unitary)	Time (s)	0.023	0.025	0.059	0.357	3.890	29.861	436.450	-
			Mem (MB)	< 2000	< 2000	< 2000	< 2000	2200.990	2767.230	10666.570	-
		Linear-cyclic	Time (s)	0.023	0.028	0.053	0.270	2.337	6.470	197.810	2353.696
			Mem (MB)	< 2000	< 2000	< 2000	< 2000	2148.300	2183.760	4814.550	16091.660
	`mitms`		Time (s)	0.002	0.019	0.188	0.248	12.433	32.766	-	-
			Mem (MB)	0.002	0.016	0.147	1.013	6.249	84.622	-	-
3	`Quokka#-syn`	Cyclic (unitary)	Time (s)	0.024	0.106	3.310	177.220	-	-	-	-
			Mem (MB)	< 2000	< 2000	2188.920	5398.730	-	-	-	-
		Linear-cyclic	Time (s)	0.029	0.140	3.413	76.293	3.066.258	-	-	-
			Mem (MB)	< 2000	< 2000	2148.740	2688.360	18860.120	-	-	-
	`mitms`		Time (s)	0.027	1.409	53.2	2311,023	-	-	-	-
			Mem (MB)	5.633	0.179	6.737	215.970	-	-	-	-

The second experiment compares the performance of `Quokka#-syn` with both unitary and circuit inputs to that of `mitms`, a tool that performs the same task, only with target gate set $\{H, CNOT, S, S^\dagger, T, T^\dagger\}$. For comparison, we selected the 2-qubit CH gate and the 3-qubit *Toffoli* gate. We encode the gates once as unitary matrices in **CB**, as described in Section 4, and once as Clifford+T circuits, as described in [42], in **PB**. We choose to use **CB** encoding for unitary inputs, since even for sparse unitaries, such as the Toffoli gate and its generalizations, the encoding of the Pauli decomposition may blow up, as shown in [39]. For the circuit encoding, we use the *linear-cyclic encoding* in **PB**, as it showed the best performance in the previous experiment. For both encodings, the timeout is set to one hour (including the encoding time and calling of d4Max for all depths). Since we failed to compile `mitms` within our experimental setup due to its reliance on older libraries, we instead refer to the performance data reported in [3] to provide a comparative reference. The results are shown in Table 4. While the comparison is not entirely fair – due to differences in platform, target gate set, and benchmark design – we observe that `Quokka#-syn` tends to exhibit higher memory usage but lower runtime compared to `mitms`.

It is also worth noting that the performance of **PB** with the linear-cyclic encoding given in Table 3 are better than Table 4 in many cases, for example, 3-qubit and 5-depth cases in Table 3 feature 70% success rate with average runtime of 147.37 seconds, while in Table 4, it takes 3066.26 seconds. One explanation is that, since the maximal weight is known, as shown in Theorem 5, `Quokka#-syn` provides a threshold to the solver, allowing it to terminate as soon as the optimal value is achieved. Consequently, if a circuit can be successfully synthesized within the given depth, as with the solved cases in Table 3, the tool completes early. In contrast, for unsuccessful cases, such as for each depth in Theorem 5, the solver must explore the entire search space, resulting in longer run times. In addition, we observe substantial variability in the run times, indicating that the performance is highly dependent on the specific characteristics of each case.

Approximate synthesis. Important gates that often need to be synthesized approximately in Clifford+T are the general rotation gates: R_x, R_y, R_z [30]. Our method supports a general rotation angle in circuits, i.e., the rotation angle can be any real number. The encoding will be the same Boolean formula, but with a weight function dependent on the rotation angle [38]. Thus, different rotation angles do not significantly affect the performance per depth of the approximate synthesis of the rotation gate R_z. Hence, to demonstrate the use of approximate synthesis, we consider the quantum gate $R_z(\frac{\pi}{8}) = \begin{bmatrix} e^{-i\pi/16} & 0 \\ 0 & e^{i\pi/16} \end{bmatrix}$. We

Table 5 $R_Z(\pi/8)$ synthesis using **CB** with cyclic encoding. Statistics are shown per synthesis layer. Memory usage under short runtimes is omitted (< 2). Only fidelity-improving iterations are shown. At 24 layers, the program crashed due to resource limits (-).

depth	1	10	15	24
#variables	11	92	137	218
#clauses	37	382	577	928
#literals	90	960	1460	2360
#Selecting variables	4	40	60	96
fidelity	0.962	0.975	0.997	-
Time (s)	0.021	0.092	2.685	>1560.51
mem (GB)	<2	<2	2.2	>33.31

choose to use the **CB** with the *cyclic encoding* as it outperforms the **PB** with the *cyclic encoding*, according to the results in Table 3 (Recall that **PB** with the *linear-cyclic encoding* cannot perform *approximate* synthesis). We report statistics for each synthesis layer where an improvement in the achieved fidelity is observed. The results are presented in Table 5. The corresponding output circuits are as follows.

- **1 Layer:** with $\epsilon = 0.1$, $\mathcal{C}' = (T)$
- **10 Layer:** with $\epsilon = 0.1$, $\mathcal{C}' = (T^\dagger, H, T^\dagger, H, T^\dagger, H, T^\dagger, H, T^\dagger, H)$
- **15 Layer:** with $\epsilon = 0.01$, $\mathcal{C}' = (H, T^\dagger, H, T, H, T, H, T^\dagger, H, T, H, T, H, T^\dagger, H)$

Other tools also target approximate synthesis. For example, the optimal tool `mitms` and other non-optimal tools such as `gridsynth`[47] use operator norm as a metric, while [41] uses the Jamiołkowski fidelity, though the implementation of the latter one is not open-source. We do not include a performance comparison because, to the best of our knowledge, no available tool performs optimal approximate synthesis with fidelity as a metric.

8 Conclusion

The results presented in this work demonstrate that maximum weighted model counting can be effectively employed for both exact and approximate quantum circuit synthesis. This approach benefits from the generality and extensibility of weighted model counting techniques as used in circuit simulation and equivalence checking, particularly their compatibility with diverse gate sets and representation bases. However, in its current form, the maximum weighted model counting exhibits limited performance in this new context.

There are, nonetheless, several promising directions for further development: (1) The current implementation builds upon a prototype version of `d4Max`, which lacks several optimizations to accommodate the extension to negative and complex weights. The design of dedicated algorithms tailored to this application remains an open avenue for future research. (2) Our layer-by-layer synthesis approach could benefit from incremental solving, as used in bounded model checking [9, 24]. Although incremental approaches have been studied for sampling and weighted model counting [54, 55], their application to maximum weighted model counting has not yet been explored. (3) Since the inception of the Model Counting Competition,[†] continuous progress has been observed in the capabilities of model counters. By contributing our benchmarks to the community, we anticipate further methodological

[†] `https://mccompetition.org/`

improvements that may enhance the reliability and performance of this approach. (4) The particular characteristics in the encoded CNF, such as an abundance of XOR clauses [40], are not yet exploited by the solver. This could lead to significant performance gains [32]. (5) Lastly, incorporating symmetry considerations may further improve the efficiency of model counting, as has been demonstrated in related domains [7].

The depth-optimal synthesis presented here enables more efficient implementations of multi-qubit gates, a critical requirement in fault-tolerant and error-corrected quantum computing [27]. Future work will also address minimizing the T count, which remains an important objective in error-corrected architectures [30]. While the current study focuses on the Clifford+T gate set, the underlying encoding is readily generalizable to other gate sets. Moreover, extending maximum weighted model counting to the stochastic SAT setting [35, 34] could broaden its applicability to further quantum circuit optimization problems. This line of work may also lead to the derivation of novel lower bounds for computationally hard problems in quantum circuit analysis, based on established results from the reasoning and satisfiability domains [6].

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 Panos Aliferis, Daniel Gottesman, and John Preskill. Quantum accuracy threshold for concatenated distance-3 codes. *Quantum Inf. Comput.*, 6(2):97–165, March 2006. doi:10.26421/QIC6.2-1.

3 Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 32(6):818–830, 2013. doi:10.1109/TCAD.2013.2244643.

4 Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019. doi:10.1038/s41586-019-1666-5.

5 Gilles Audemard, Jean-Marie Lagniez, and Marie Miceli. A new exact solver for (weighted) max#sat. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:20, Haifa, Israel, August 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SAT.2022.28.

6 Max Bannach and Markus Hecher. On weighted maximum model counting: complexity and fragments. In *36th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2024, Herndon, VA, USA, October 28-30, 2024*, pages 1–9. IEEE, 2024. doi:10.1109/ICTAI62512.2024.00010.

7 Anicet Bart, Frédéric Koriche, Jean-Marie Lagniez, and Pierre Marquis. Symmetry-driven decision diagrams for knowledge compilation. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 51–56. IOS Press, 2014. doi:10.3233/978-1-61499-419-0-51.

8 Ethan Bernstein and Umesh V. Vazirani. Quantum complexity theory. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 11–20. ACM, 1993. doi:10.1145/167088.167097.

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, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

11 Sergey Bravyi, Dan E. Browne, Padraic Calpin, Earl T. Campbell, David Gosset, and Mark Howard. Simulation of quantum circuits by low-rank stabilizer decompositions. *Quantum*, 3:181, September 2019. doi:10.22331/q-2019-09-02-181.

12 Sergey Bravyi, Graeme Smith, and John A. Smolin. Trading classical and quantum computational resources. *Phys. Rev. X*, 6:021043, June 2016. doi:10.1103/PhysRevX.6.021043.

13 Lukas Burgholzer and Robert Wille. Advanced equivalence checking for quantum circuits. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 40(9):1810–1824, September 2020. doi:10.1109/tcad.2020.3032630.

14 A. R. Calderbank, E. M. Rains, P. W. Shor, and N. J. A. Sloane. Quantum error correction and orthogonal geometry. *Phys. Rev. Lett.*, 78:405–408, January 1997. doi:10.1103/PhysRevLett.78.405.

15 Senrui Chen, Sisi Zhou, Alireza Seif, and Liang Jiang. Quantum advantages for pauli channel estimation. *Physical Review A*, 2021. URL: https://api.semanticscholar.org/CorpusID:237213441.

16 Yu-Fang Chen, Kai-Min Chung, Ondrej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. An automata-based framework for verification and bug hunting in quantum circuits. *Proc. ACM Program. Lang.*, 7(PLDI):1218–1243, June 2023. doi:10.1145/3591270.

17 Christopher M. Dawson and Michael A. Nielsen. The solovay-kitaev algorithm. *Quantum Inf. Comput.*, 6(1):81–95, 2006. doi:10.26421/QIC6.1-6.

18 M. Van den Nest. Classical simulation of quantum computation, the gottesman-knill theorem, and slightly beyond. *arXiv:0811.0898*, 2008. doi:10.48550/arXiv.0811.0898.

19 Austin G. Fowler. Constructing arbitrary steane code single logical qubit fault-tolerant gates. *Quantum Inf. Comput.*, 11(9&10):867–873, September 2011. doi:10.26421/QIC11.9-10-10.

20 Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A*, 86(3):032324, 2012. doi:10.1103/PhysRevA.86.032324.

21 Vlad Gheorghiu, Michele Mosca, and Priyanka Mukhopadhyay. T-count and T-depth of any multi-qubit unitary. *npj Quantum Information*, 8(1), November 2022. doi:10.1038/s41534-022-00651-y.

22 Brett Giles and Peter Selinger. Exact synthesis of multiqubit Clifford + T circuits. *Physical Review A*, 87(3):032332, 2013. doi:10.1103/PhysRevA.87.032332.

23 David Gosset, Vadym Kliuchnikov, Michele Mosca, and Vincent Russo. An algorithm for the T-count, 2013. doi:10.48550/arXiv.1308.4134.

24 Henning Günther and Georg Weissenbacher. Incremental bounded software model checking. In Neha Rungta and Oksana Tkachuk, editors, *2014 International Symposium on Model Checking of Software, SPIN 2014, Proceedings, San Jose, CA, USA, July 21-23, 2014*, pages 40–47. ACM, 2014. doi:10.1145/2632362.2632374.

25 Yifei Huang and Peter Love. Approximate stabilizer rank and improved weak simulation of Clifford-dominated circuits for qudits. *Phys. Rev. A*, 99:052307, May 2019. doi:10.1103/PhysRevA.99.052307.

26 Zhengfeng Ji and Xiaodi Wu. Non-identity check remains QMA-complete for short circuits. *arXiv:0906.5416*, 2009. doi:10.48550/arXiv.0906.5416.

27 N. Cody Jones. Logic synthesis for fault-tolerant quantum computers, 2013. doi:10.48550/arXiv.1310.7290.

28 Aleks Kissinger and John van de Wetering. Simulating quantum circuits with zx-calculus reduced stabiliser decompositions. *Quantum Science and Technology*, 7(4):044001, 2022. doi:10.48550/arXiv.2109.01076.

29 Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Fast and efficient exact synthesis of single qubit unitaries generated by Clifford and T gates, 2013. doi:10.48550/arXiv.1206.5236.

30 Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Practical approximation of single-qubit unitaries by single-qubit quantum clifford and T circuits. *IEEE Transactions on Computers*, 65(1):161–172, 2016. doi:10.1109/TC.2015.2409842.

31 Lucas Kocia and Mohan Sarovar. Improved simulation of quantum circuits by fewer Gaussian eliminations. *Physical Review A*, 103(2), 2021. doi:10.1103/physreva.103.022603.

32 Frédéric Koriche, Jean-Marie Lagniez, Pierre Marquis, and Samuel Thomas. Knowledge compilation for model counting: Affine decision trees. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 947–953. IJCAI/AAAI, 2013. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6574.

33 Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. *ACM Sigplan Notices*, 45(6):316–329, 2010. doi:10.1145/1806596.1806632.

34 Nian-Ze Lee, Yen-Shi Wang, and Jie-Hong R Jiang. Solving stochastic boolean satisfiability under random-exist quantification. In *IJCAI*, pages 688–694, 2017. doi:10.24963/ijcai.2017/96.

35 Michael L Littman, Stephen M Majercik, and Toniann Pitassi. Stochastic boolean satisfiability. *Journal of Automated Reasoning*, 27:251–296, 2001. doi:10.1023/A:1017584715408.

36 Dmitri Maslov and Martin Roetteler. Shorter stabilizer circuits via bruhat decomposition and quantum circuit transformations. *IEEE Transactions on Information Theory*, 64(7):4729–4738, July 2018. doi:10.1109/tit.2018.2825602.

37 Olivia Di Matteo and Michele Mosca. Parallelizing quantum circuit synthesis. *Quantum Science and Technology*, 1(1):015003, October 2016. doi:10.1088/2058-9565/1/1/015003.

38 Jingyi Mei, Marcello M. Bonsangue, and Alfons Laarman. Simulating quantum circuits by model counting. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part III*, volume 14683 of *Lecture Notes in Computer Science*, pages 555–578. Springer, 2024. doi:10.1007/978-3-031-65633-0_25.

39 Jingyi Mei, Tim Coopmans, Marcello M. Bonsangue, and Alfons Laarman. Equivalence checking of quantum circuits by model counting. In Christoph Benzmüller, Marijn J. H. Heule, and Renate A. Schmidt, editors, *Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part II*, volume 14740 of *Lecture Notes in Computer Science*, pages 401–421. Springer, 2024. doi:10.1007/978-3-031-63501-4_21.

40 Jingyi Mei, Jan Martens, and Alfons Laarman. Disentangling the gap between quantum and #sat. In Chutiporn Anutariya and Marcello M. Bonsangue, editors, *Theoretical Aspects of Computing - ICTAC 2024 - 21st International Colloquium, Bangkok, Thailand, November 25-29, 2024, Proceedings*, volume 15373 of *Lecture Notes in Computer Science*, pages 17–40, Berlin, Heidelberg, 2024. Springer. doi:10.1007/978-3-031-77019-7_2.

41 Richard Meister, Cica Gustiani, and Simon C Benjamin. Exploring ab initio machine synthesis of quantum circuits. *New Journal of Physics*, 25(7):073018, July 2023. doi:10.1088/1367-2630/ace077.

42 Michael A Nielsen and Isaac L Chuang. Quantum information and quantum computation. *Cambridge: Cambridge University Press*, 2(8):23, 2000.

43 Philipp Niemann, Robert Wille, and Rolf Drechsler. Advanced exact synthesis of clifford+t circuits. *Quantum Information Processing*, 19(9):317, August 2020. doi:10.1007/s11128-020-02816-0.

44 Anouk Paradis, Jasper Dekoninck, Benjamin Bichsel, and Martin T. Vechev. Synthetiq: Fast and versatile quantum circuit synthesis. *Proc. ACM Program. Lang.*, 8(OOPSLA1):55–82, April 2024. doi:10.1145/3649813.

45 Maxim Raginsky. A fidelity measure for quantum channels. *Physics Letters A*, 290(1-2):11–18, 2001. doi:10.1016/S0375-9601(01)00640-5.

46 Narayanan Rengaswamy, A. Robert Calderbank, Henry D. Pfister, and Swanand Kadhe. Synthesis of logical clifford operators via symplectic geometry. In *2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018*, pages 791–795. IEEE, 2018. doi:10.1109/ISIT.2018.8437652.

47 Neil J. Ross and Peter Selinger. Optimal ancilla-free clifford+t approximation of z-rotations. *CoRR*, abs/1403.2975, 2016. doi:10.48550/arXiv.1403.2975.

48 Sarah Schneider, Lukas Burgholzer, and Robert Wille. A SAT encoding for optimal clifford circuit synthesis. In Atsushi Takahashi, editor, *Proceedings of the 28th Asia and South Pacific Design Automation Conference, ASPDAC 2023, Tokyo, Japan, January 16-19, 2023*, ASPDAC '23, pages 190–195, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3566097.3567929.

49 Peter Selinger. Efficient Clifford+T approximation of single-qubit operators, 2014. doi:10.48550/arXiv.1212.6253.

50 Meghana Sistla, Swarat Chaudhuri, and Thomas W. Reps. Symbolic quantum simulation with quasimodo. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III*, volume 13966 of *Lecture Notes in Computer Science*, pages 213–225. Springer, Springer, 2023. doi:10.1007/978-3-031-37709-9_11.

51 Dimitrios Thanos, Tim Coopmans, and Alfons Laarman. Fast equivalence checking of quantum circuits of clifford gates. In Étienne André and Jun Sun, editors, *Automated Technology for Verification and Analysis - 21st International Symposium, ATVA 2023, Singapore, October 24-27, 2023, Proceedings, Part II*, volume 14216 of *Lecture Notes in Computer Science*, pages 199–216, Cham, 2023. Springer. doi:10.1007/978-3-031-45332-8_10.

52 Dimitrios Thanos, Alejandro Villoria, Sebastiaan Brand, Arend-Jan Quist, Jingyi Mei, Tim Coopmans, and Alfons Laarman. Automated reasoning in quantum circuit compilation. In *SPIN 2024*, 2024. URL: https://spin-web.github.io/SPIN2024/assets/preproceedings/SPIN2024-paper6.pdf.

53 Lieuwe Vinkhuijzen, Tim Coopmans, David Elkouss, Vedran Dunjko, and Alfons Laarman. LIMDD: A decision diagram for simulation of quantum computing including stabilizer states. *Quantum*, 7:1108, 2023. doi:10.22331/q-2023-09-11-1108.

54 Suwei Yang, Victor C. Liang, and Kuldeep S. Meel. INC: A scalable incremental weighted sampler. In Alberto Griggio and Neha Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, volume 3, pages 205–213. TU Wien Academic Press, IEEE, 2022. doi:10.34727/2022/isbn.978-3-85448-053-2_27.

55 Suwei Yang and Kuldeep S. Meel. Towards projected and incremental pseudo-boolean model counting. *CoRR*, abs/2412.14485, 2024. doi:10.48550/arXiv.2412.14485.

A Appendix

A.1 Proof of Theorem 5

▶ **Theorem 5.** *Let $\mathcal{C}$ be an n-qubit circuit, which is encoded by $F_{\mathcal{C}}$ with the corresponding weight function W. Then, the following four statements are equivalent to each other:*

- *$\mathcal{C} \equiv I^{\otimes n}$ ($\mathcal{C}$ is equivalent to the identity circuit $I^{\otimes n} = I_{2^n}$).*
- *Encoding the circuit $\mathcal{C}$ in* **PB**, *the* ***linear encoding*** *has weighted model count [39]:*

$$\#SAT_W(F_{\mathcal{P}}(\vec{q}) \wedge F_{\mathcal{C}}(\vec{q},\vec{q}') \wedge F_{\mathcal{P}}(\vec{q}')) = 1 \text{ for all } \mathcal{P} \in \{X_j, Z_j \mid j \in [n]\}. \tag{4}$$

- *Encoding the circuit in either* **CB** *or* **PB**, *the* ***cyclic encoding*** *has weighted model count (this approach can be viewed as checking "overlap" with the identity $I^{\otimes n}$):*

$$\#SAT_W(F_{\mathcal{C}}(\vec{q},\vec{q}') \wedge F_{I^{\otimes n}}(\vec{q},\vec{q}')) = c \text{ with } |c| = 2^n \text{ for } \mathbf{CB} \text{ and } c = 4^n \text{ for } \mathbf{PB}. \quad (5)$$

- *Encoding the circuit $\mathcal{C}$ in* **PB**, *the* ***linear-cyclic encoding*** *has weighted model count:*

$$\#SAT_W(\bigvee_{\mathcal{P}\in\{X_j,Z_j|j\in[n]\}} F_{\mathcal{P}}(\vec{q}) \wedge F_{\mathcal{C}}(\vec{q},\vec{q}') \wedge F_{I^{\otimes n}}(\vec{q},\vec{q}')) = 2n. \quad (6)$$

where $\vec{q},\vec{q}'$ are Boolean variables encoding the initial and final quantum state, respectively. Note that $F_{I^{\otimes n}}(\vec{q},\vec{q}') = \bigwedge_{i\in[n]}(q_i \Leftrightarrow q_i')$, where $q_i \Leftrightarrow q_i'$ is shorthand for $(x_i \Leftrightarrow x_i') \wedge (z_i \Leftrightarrow z_i')$ in the Pauli basis.

Proof. First of all, from [39, Cor. 1], the circuit $\mathcal{C}$ is equivalent to the identity $I^{\otimes n}$ if and only if Equation 4 holds.

Next, we show that all three equations Equation 6 and Equation 5 in both bases are equivalent.

For Equation 6, as stated in Corollary 1 and the proof of Lemma 2 in [39], we have

$$\#SAT_W(F_{\mathcal{P}}(\vec{q}) \wedge F_{\mathcal{C}}(\vec{q},\vec{q}') \wedge F_{\mathcal{P}}(\vec{q}')) \leq 1.$$

for all $\mathcal{P} \in \{X_j, Z_j \mid j \in [n]\}$. Thus we infer that

$$\sum_{\mathcal{P}\in\{X_j,Z_j|j\in[n]\}} \#SAT_W(F_{\mathcal{P}}(\vec{q}) \wedge F_{\mathcal{C}}(\vec{q},\vec{q}') \wedge F_{\mathcal{P}}(\vec{q}')) \leq 2n,$$

where the value achieves $2n$ if and only if each of the summands achieves 1. Therefore Equation 6 is true if and only if Equation 4 is true, as demonstrated in [39, Prop. 1].

For Equation 5 in **PB**, the idea is similar. The value of the weighted model count of Equation 5 is equivalent to

$$\sum_{\mathcal{P}\in\{X,Y,Z,I\}^{\otimes n}} \#SAT_W(F_{\mathcal{P}}(\vec{q}) \wedge F_{\mathcal{C}}(\vec{q},\vec{q}) \wedge F_{\mathcal{P}}(\vec{q})) \leq 4^n,$$

which can achieve 4^n if and only if for all 4^n Pauli strings $\mathcal{P}$, the weighted model counting of $\mathcal{P}$ achieves 1. Since $\{X_j, Z_j \mid j \in [n]\} \subseteq \{X, Y, Z, I\}^{\otimes n}$, we have *Equation* 5 $\Rightarrow$ *Equation* 4. From [51], if two unitaries are equivalent over $\{X_j, Z_j \mid j \in [n]\}$, they are equivalent over $\{X, Y, Z, I\}^{\otimes n}$. We have Equation 4$\Rightarrow$ Equation 5. Therefore Equation 4$\Leftrightarrow$ Equation 5.

For Equation 5 in **CB**, since

$$\begin{aligned}
U_{\mathcal{C}} = \lambda \cdot I_{2^n} &\Leftrightarrow \langle b| U_{\mathcal{C}} |b\rangle = \lambda \text{ for } b \in \{0,1\}^n \\
&\Leftrightarrow \sum_{b\in\{0,1\}^n} \langle b| U_{\mathcal{C}} |b\rangle = \lambda \cdot 2^n, \\
&\Leftrightarrow \sum_{b\in\{0,1\}^n} \Big(\#SAT_W\big(F_{|b\rangle}(\vec{q}) \wedge F_{\mathcal{C}}(\vec{q},\vec{q}') \wedge F_{|b\rangle}(\vec{q}')\big)\Big) = \lambda \cdot 2^n, \qquad (\textit{Lemma } 2) \\
&\Leftrightarrow \sum_{b\in\{0,1\}^n} \Big(\#SAT_W\big(F_{|b\rangle}(\vec{q}) \wedge F_{\mathcal{C}}(\vec{q},\vec{q}') \wedge F_{I_{2^n}}(\vec{q},\vec{q}')\big)\Big) = \lambda \cdot 2^n, \\
&\Leftrightarrow \#SAT_W(F_{\mathcal{C}}(\vec{q},\vec{q}') \wedge F_{I_{2^n}}(\vec{q},\vec{q}')) = \lambda \cdot 2^n,
\end{aligned}$$

where $|\lambda|^2 = 1$, we have Equation 5 $\Leftrightarrow \mathcal{C} \equiv I^{\otimes n}$. ◀

A.2 Optimization rules

To optimize the synthesis encoding, we implement additional constraints.

The first set of rules ensures that we avoid redundant combinations of gates, such as HH, since they can be replaced with I gates:

- **Rule 1**: No two H gates in a row on the same qubit, since $HH = I$:

$$F^d_{R1}(P) = \bigwedge_{k\in[d-1]} \bigwedge_{i\in[n]} (\overline{p}^k_{H,i} \vee \overline{p}^{k+1}_{H,i})$$

- **Rule 2**: No 8 T gates in a row on the same qubit since $T^8 = I$:

$$F^d_{R2}(P) = \bigwedge_{k\in[d-7]} \bigwedge_{i\in[n]} \bigvee_{j\in[8]} \overline{p}^{k+j}_{T,i}$$

- **Rule 3**: No two CX gates in a row on the same qubits since $CX_{i,j} \cdot CX_{i,j} = I$:

$$F^d_{R3}(P) = \bigwedge_{k\in[d-1]} \bigwedge_{i,j\in[n],j\neq i} (\overline{p}^k_{CX,i,j} \vee \overline{p}^{k+1}_{CX,i,j})$$

The second set of rules aims to have a canonical representation for a given set of gates. Our guideline is that every non-I is pushed back as far as possible. This means not allowing any single qubit gate to follow an I gate, other than I itself. For two-qubit gates, we do not allow following I on both qubits.

- **Rule 4**: No single qubit gates other than I after an I gate:

$$F^d_{R4}(P) = \bigwedge_{k\in[d-1]} \bigwedge_{i\in[n]} (p^k_{I,i} \Rightarrow (p^{k+1}_{I,i} \vee \bigvee_{j\in[n],j\neq i} (p^{k+1}_{CX,i,j} \vee p^{k+1}_{CX,j,i})))$$

- **Rule 5**: No CX gate following I gate on both qubits:

$$F^d_{R5}(P) = \bigwedge_{k\in[d-1]} \bigwedge_{i,j\in[n],j\neq i} (p^{k+1}_{CX,i,j} \Rightarrow (\overline{p}^k_{I,i} \vee \overline{p}^k_{I,j}))$$

The 3-Decomposition Conjecture: A SAT-Based Approach with Specialized Propagators

Tianwei Zhang
Algorithms and Complexity Group, TU Wien, Austria

Stefan Szeider
Algorithms and Complexity Group, TU Wien, Austria

Abstract

We investigate the 3-decomposition conjecture, which states that every connected cubic graph can be decomposed into a spanning tree, a collection of cycles, and a matching. Using a SAT-based approach enhanced with specialized propagators, we verify the conjecture for all relevant graphs up to 28 vertices. Our method extends the Satisfiability Modulo Symmetries (SMS) framework with specialized propagators that exploit theoretical properties of minimal counterexamples (counterexamples with the minimal number of vertices), enabling efficient pruning. We demonstrate that graphs containing certain substructures cannot be minimal counterexamples to the conjecture, allowing us to exclude these patterns during the search dynamically. Our experimental results quantify the impact of different propagator configurations and forbidden subgraph constraints on solving efficiency, showing significant performance improvements when leveraging these techniques. The approach scales effectively to graphs of 28 vertices. Our work illustrates how combining SAT solving with specialized constraint propagation techniques can successfully address challenging combinatorial problems in contemporary graph theory.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Mathematics of computing → Graph theory; Theory of computation → Logic and verification; Mathematics of computing → Combinatorics; Theory of computation → Automated reasoning; Mathematics of computing → Graph algorithms

Keywords and phrases SAT, Symmetry Breaking, Subgraphs, Propagators, Combinatorics

Digital Object Identifier 10.4230/LIPIcs.CP.2025.39

Supplementary Material *Other*: `https://doi.org/10.5281/zenodo.15623903`

Funding The project leading to this publication has received funding from the European Union's Horizon 2020 research and innovation programme under the grant agreement No. 101034440 and was supported by the Austrian Science Fund (FWF) within the project 10.55776/P36688.

1 Introduction

Over the last few years, computational methods have played an important role in mathematical discovery. On the one hand, machine-learning guided heuristic search has successfully established new results on the existence of certain combinatorial objects [22, 23]. On the

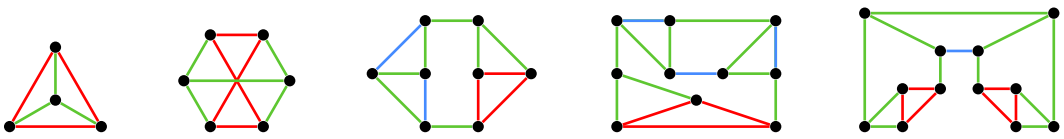


Figure 1 Examples of 3-decompositions of cubic graphs of order 4, 6, 8, 10, and 12. Cubic graphs only have an even number of vertices since the number of edges in a cubic graph with n vertices is $3n/2$. Note that it is possible to have an empty matching, as is shown in the first two examples.

© Tianwei Zhang and Stefan Szeider;
licensed under Creative Commons License CC-BY 4.0

other hand, for non-existence or optimality results (like what is the smallest object with a certain property), where the search space must be explored rigorously, constraint-based methods have been successful [5, 7, 8, 14, 17, 25].

In this paper, we tackle the *3-Decomposition Conjecture (3DC)* with constraint-based methods. The conjecture, formulated by Hoffmann-Ostenhof [6] in 2009, is an open problem in discrete mathematics that attracted a lot of research interest [9]. The conjecture states that every connected cubic graph (each vertex has exactly three neighbors) can be partitioned into three edge-disjoint subgraphs: a spanning tree, a collection of cycles, and a matching. Figure 1 shows several small cubic graphs with the 3-decomposition indicated by colors: green edges represent the spanning tree, red edges the cycles, and blue edges the matching. The conjecture has been verified for connected cubic graphs that are planar, Hamiltonian, traceable, claw-free, or 4-chordal, and for connected cubic graphs of treewidth ≤ 3 [4, 15, 2, 19, 1]. However, these results on special cases only cover a small fraction of all connected cubic graphs for which, despite considerable efforts, the conjecture remains open.

Our main results are as follows.

(1) We verify the 3DC for all connected cubic graphs up to 28 vertices with constraint-based methods.

(2) We systematically compute a list of graphs with up to 18 vertices such that a minimal counterexample to the 3DC cannot contain any graph from the list as a subgraph.

(3) We make an experimental study on how much the use of the graphs from (2) speeds up the search for a 3DC counterexample.

Result (1) is a significant improvement over the known bound of 20 which was established by listing all connected cubic graphs up to $n = 20$ vertices, modulo isomorphism, and checking for each of them whether it has a 3-decomposition [4]. There are $f(20) = 510.489$ such graphs [11]; hence, this generate-and-test approach is feasible. However, for larger n, this becomes quickly infeasible as we have $f(22) > 7.3 \times 10^6$, $f(24) > 1.1 \times 10^8$, $f(26) > 2.0 \times 10^9$, and $f(28) > 4.0 \times 10^{10}$. Our approach circumvents this obstacle by integrating search and test with a combination of SAT encodings with specialized propagators.

Result (2) yields a complete list of all *reducible templates*, a specific type of subcubic graphs, up to index 6 and order 18, from which we deduce the list of subgraphs that a minimal counterexample to the 3DC cannot contain. Some reducible templates have already been identified by Bachtler et al. [4], but our search yields new instances. Observe that Result (2) has a direct effect on Result (1): if we know the forbidden subgraphs, then we can use them to prune the search tree for a 3DC counterexample and thus make the search for (1) more efficient. While Result (1) confirms the conjecture for a finite number of graphs (modulo isomorphism), Result (2) applies to minimal counterexamples of any size, i.e., to an infinite number of graphs (modulo isomorphism), hence can be considered stronger from a mathematical point of view.

Excluding subgraphs during the search requires repeated subgraph testing, known to be NP-complete. We efficiently accomplish this by tightly integrating the Glasgow Subgraph Solver [21] into our method. However, as these tests are frequent, they become computationally costly. Hence, we face a tradeoff: when is the subgraph testing too costly to outweigh the speedup achieved by search space pruning? Result (3) gives a systematic insight into this question. We provide rigorous experimental data that allows us to pinpoint a good balance between the two conflicting factors. Our analysis shows that incorporating forbidden subgraph constraints derived from smaller cases can drastically reduce search times for larger cases. We report on the specific performance improvements for graphs of various sizes.

Our work illustrates how SAT and CP techniques can be combined to tackle challenging combinatorial problems. Our dynamic forbidden subgraph approach naturally extends to many graph conjecture verification tasks where "minimality" can be meaningfully defined. Rather than pre-computing constraints, our method discovers forbidden patterns during the search process itself, creating increasingly powerful pruning rules as exploration progresses. This approach is directly applicable to coloring problems, extremal graph theory conjectures, and Ramsey-type problems, where minimal counterexamples must satisfy specific structural properties. The key insight is that local patterns discovered in one search branch can inform constraints for the entire remaining search space. This "learn-as-you-go" paradigm is particularly valuable for problems where theoretical analysis alone cannot easily identify all forbidden patterns, potentially transforming how we approach computational verification across combinatorial mathematics.

2 Preliminaries

For positive integers $k < \ell$, we write $[k] = \{1, 2, \ldots, k\}$, and $[k, \ell] = \{k, \ldots, \ell\}$.

CNF formulas. A *literal* is a (propositional) variable x or a negated variable $\overline{x}$. A *clause* is a disjunction of a finite set of literals. A *formula* in *conjunctive normal form (CNF)* or *clausal normal form* is a conjunction of a finite set of clauses.

Simple graphs. A *simple graph* is a graph without parallel edges or self-loops. In this paper, the term *graphs* refers to *simple graphs* unless otherwise specified. A simple graph G consists of a set $V(G)$ of vertices and a set $E(G)$ of edges; we denote the edge between vertices $u, v \in V(G)$ by uv or, equivalently, vu. We thus write (V, E) to denote a graph, where V is the vertex set and E the edge set. Two vertices $u, v \in V(G)$ are *adjacent* if $uv \in E(G)$. The *degree* of a vertex $v \in V(G)$, denoted as $\deg v$, is the number of vertices in G that it is adjacent to. For $k \geq 0$, a k-regular graph is a graph where each vertex has exactly degree k. A 3-regular graph is also called a *cubic* graph. A graph where each vertex has at most 3 neighbors is a *subcubic* graph. A graph is *connected* if there is a path between any distinct pair of vertices. A *cycle* is a connected 2-regular graph. A *matching* is a graph where no two edges share a common vertex. A graph is a *tree* if there is a unique path between any pair of distinct vertices. A graph T is a *spanning tree* of a graph G if T is a tree, $V(T) = V(G)$, and $E(T) \subseteq E(G)$. A graph H is a *subgraph* of G if there is an injective mapping $\varphi : V(H) \to V(G)$ such that for any $u, v \in V(H)$, if $uv \in E(H)$ then $\varphi(u)\varphi(v) \in E(G)$. H is an *induced subgraph* of G if the mapping above additionally satisfies that for any $u, v \in V(H)$, $uv \in E(H)$ if $\varphi(u)\varphi(v) \in E(G)$. If two graphs are induced subgraphs of each other, then we say the two graphs are *isomorphic*, and we call the witnessing mappings *isomorphisms*. Given $E \subseteq E(G)$, the subgraph of G *induced* by E is the graph H, where $V(H) := \bigcup_{uv \in E} \{u, v\}$ and $E(H) := E$. The *adjacency matrix* A of a graph G with $|V(G)| = n$ is the $n \times n$ $\{0, 1\}$-matrix where the element at row v and column u, denoted by $A(v, u)$, is 1 if and only if $vu \in E(G)$. Let $e(i, j)$ denote the propositional *edge variable* for the possible edge between vertices i and j. If $e(i, j) = 1$, then the possible edge is present; otherwise, the possible edge is absent.

SAT modulo symmetries (SMS). SMS [18] is a framework that augments a CDCL (conflict-driven clause learning) SAT solver [10, 20]. The framework is specifically designed for enumerating graphs that satisfy given properties expressed in the form of a formula in

propositional logic. Usually, properties of graphs that we take into consideration are invariant under isomorphisms, and SMS benefits from only considering one adjacency matrix out of the numerous different adjacency matrices that represent the same graph (modulo isomorphism). SMS implements this idea by connecting a CDCL SAT solver to an external propagator, which checks from time to time whether the current assignment can be extended to a lexicographically minimal (*canonical*) matrix (more precisely, only those copies are kept that are when considering the rows of the adjacency matrix concatenated into a single vector), and gives feedback to the CDCL solver in the form of *symmetry breaking clauses* accordingly. This synergy between the CDCL solver and the external propagator is not restricted to generating canonical matrices. In fact, customized external propagators can be added to learn clauses and trim the search tree to suit the specific task at hand. As of today, SMS has incorporated many commonly sought-after functionalities, such as generating graphs of certain connectivity, graphs without certain subgraphs, and so on. It is possible to specify a partition of the vertex set and restrict the symmetry breaking to those permutations that preserve the partition. For a full description of SMS, we refer to the original work where the framework was introduced [18].

Partially defined graphs. The notion of partially defined graphs was introduced in the context of SMS [17] to capture the combinatorial object represented by a partial truth assignment over the edge variables of a graph. A *partially defined graph* is a graph where some edges are undefined in the sense that their presence in the graph is open. Formally, a partially defined graph G is a graph whose edge set $E(G)$ is split into disjoint sets $E_d(G)$ and $E_n(G)$, the sets of defined edges and defined non-edges, respectively. G is *fully defined* if $uv \in E_d(G) \cup E_n(G)$ for any $u, v \in V(G)$ such that $u \neq v$.

Partially defined subgraphs occur naturally in the process of SAT solving, and many graph properties can already be shown when we only know for sure some edges exist (or do not exist) in a given graph, e.g., the existence of certain (induced) subgraphs. Thus, the benefits of considering partially defined graphs during SAT-based graph enumeration are at least twofold. First, if we can falsify the desired property in a partially defined graph, then we can block the corresponding branch of the search tree early on and speed up the search. Second, properties that are not in the class NP are usually hard to express concisely in CNF, and therefore can introduce enormous time overhead for the solver due to the large size of the formula. An external propagator, therefore, serves as a way to circumvent the clumsy encoding because it only gives the clauses for the desired properties to the solver when these clauses become useful in blocking the search path.

3 Verifying the conjecture up to $n = 28$

Let's first give a formal statement of the conjecture.

▶ **Conjecture 1** (3-Decomposition Conjecture, 3DC). *Every connected cubic graph can be decomposed into a spanning tree T, a collection of cycles C, and a (possibly empty) matching M.*

We verify the conjecture for all connected cubic graphs with n vertices up to $n = 28$. We achieve this by performing an exhaustive search for counterexamples to the conjecture through all cubic graphs with n vertices, starting with $n = 4$ and increasing by 2 each time since all cubic graphs have an even number of vertices, until we reach $n = 28$. Thus, we can safely assume that if a counterexample is found, it is a minimal counterexample. Given a

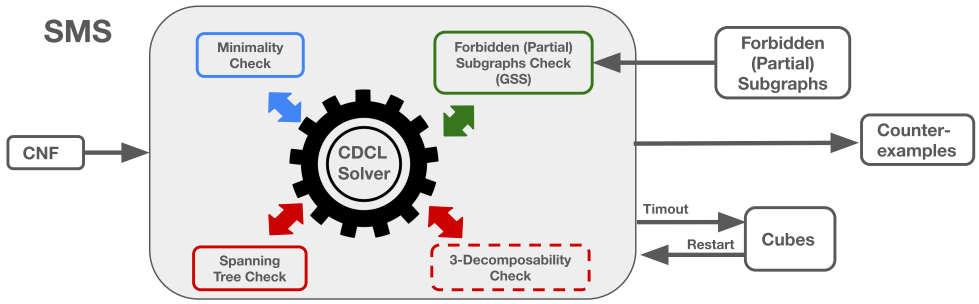


Figure 2 A roadmap of the process for verifying the 3-decomposition conjecture up to $n = 28$.

fixed n, we give the following CNF formula to SMS, augmented with specialized propagators.

$$F_{\text{cubic}}(n) := \bigwedge_{i\in[n]} \sum_{j\in[n],j\neq i} e(i,j) = 3.$$

$F_{\text{cubic}}(n)$ describes the constraints for a graph with n vertices to be cubic. The summation sign here represents a cardinality constraint, which specifies how many variables in a given set are set to true. We express cardinality constraints using cardinality networks [3]. The CNF formula and the propagators together ensure that SMS outputs all and only counterexamples, i.e., cubic graphs with n vertices that are without a 3-decomposition. This way, if SMS outputs no graphs, then we know that the conjecture holds for all graphs with n vertices. For larger cases, a timeout of one day is set. When this timeout is reached, the solving stops and various *cubes*, which are sets of literals that represent subproblems split from the original problem [13], are generated. We then restart the solver with each cube as a set of assumptions. We do this iteratively until all cubes are solved before the timeout. This process is visualized in Figure 2.

3.1 Propagators

As is shown in Figure 2, we use in total four external propagators in combination with the CDCL SAT solver. The propagator that checks whether the current graph has a 3-decomposition is only activated for checking fully defined graphs, and is enclosed in dashed lines in the figure. The other three propagators are activated for checking partially defined graphs, to which fully defined graphs also belong, and are enclosed in solid lines. The minimality check, shown in blue, is the default propagator in SMS and is responsible for filtering out non-canonical adjacency matrices as early as possible in the process of generation. The other three propagators are either modified or newly added to SMS, and we will now explain them in detail.

Forbidden (partial) subgraphs check. The forbidden (partial) subgraph check, shown in green in Figure 2, uses the Glasgow Subgraph Solver (GSS) to forbid the generation of graphs containing certain subgraphs. This functionality has already been realized in previous iterations of SMS [16]. We modified it to allow partially-defined graphs to be forbidden as subgraphs. Recall that a graph H is a *subgraph* of G if there is an injective mapping $\varphi : V(H) \to V(G)$ such that for any $u, v \in V(H)$, if $uv \in E(H)$ then $\varphi(u)\varphi(v) \in E(G)$. If H is a subgraph to be forbidden, then when the forbidden subgraphs check detects such a mapping φ, it sends the clause $\{\, \overline{\varphi(u)\varphi(v)} : uv \in E(V) \,\}$ to the SAT solver.

For our purposes, we define a relation similar to the subgraph relation for partially defined graphs. Given two partially defined graphs G and H, we say that G is a *partial subgraph* of H if there is a mapping $\varphi : V(G) \to V(H)$ such that for any $u, v \in V(G)$ we have that $\varphi(u)\varphi(v) \in E_d(H)$ if $uv \in E_d(G)$ and $\varphi(u)\varphi(v) \in E_n(H)$ if $uv \in E_n(G)$. Note that a fully defined graph G can also be seen as a simple graph with $E(G) = E_d(G)$. Given two fully defined graphs G and H, the subgraph relations differ when they are seen as simple graphs and when they are seen as partially defined graphs. In fact, G is a partial subgraph of H if and only if G is an induced subgraph of H. The reason for this definition is that we want it to be the case that G is a partial subgraph of H witnessed by φ if and only if given any way to assign the undefined edges in H to derive H', there is a way to assign the undefined edges in G to derive G', such that G' is an induced subgraph of H' witnessed by φ. If H is a partial subgraph to be forbidden, then when the forbidden partial subgraphs check detects such a mapping φ, it sends the clause $\{ \overline{\varphi(u)\varphi(v)} : uv \in E_d(V) \} \cup \{ \varphi(u)\varphi(v) : uv \in E_u(V) \}$ to the SAT solver.

In practice, the set of forbidden (partial) subgraphs we use are transformed from results by Bachtler et al. [4]. The details of the transformation are explained in Section 4.3.

Spanning tree check and 3-decomposability check. The two red propagators in Figure 2 are new additions to SMS, and both are specifically designed to filter out graphs that have a 3-decomposition. Both propagators are based on the following Theorem 2.

▶ **Theorem 2.** *Let G be a counterexample to the 3-decomposition conjecture, and T be a spanning tree of it. Then there is an edge in $E(G) \setminus E(T)$ that connects a vertex of degree 1 and another of degree 2 in T. In other words,*

$$B(T) := \{ uv : u, v \in V(T), uv \notin E(T), deg_T(u) = 1, deg_T(v) = 2 \} \neq \emptyset.$$

Proof. Assume the contrary. Let H be a simple graph such that $V(H) := \{ v \in V(G) : \deg_T(v) < 3 \}$ and $E(H) := E(G)/E(T)$. By definition, for all $v \in V(H)$, we have $\deg_H(v) = 3 - \deg_T(v)$. Since we assume that for all $uv \in E(H)$, either $\deg_T(u) = \deg_T(v) = 1$ or $\deg_T(u) = \deg_T(v) = 2$, it follows that $\deg_H(u) = \deg_H(v) = 1$ or $\deg_H(u) = \deg_H(v) = 2$. This means that H is a union of a 2-regular graph and a 1-regular graph, and therefore, we have a 3-decomposition. This contradicts our assumption. Therefore, $B(T) \neq \emptyset$. ◀

Given a spanning tree T, define the formula

$$\text{OneTwo}(T) := \left(\bigwedge_{uv \in E(T)} e(u, v) \right) \to \left(\bigvee_{uv \in B(T)} e(u, v) \right).$$

OneTwo(T) states that if all edges of the spanning tree T are present in the graph, then at least one edge that connects a vertex of the tree of degree one and another of degree two must also be present in the graph. The spanning tree check works with partially defined graphs. Given a partially defined graph G, it searches for a spanning tree T consisting of defined edges. If it finds one, and $B(T) \cap E(G) = \emptyset$, it sends the solver the formula OneTwo(T); otherwise, it does nothing. The 3-decomposition check works only with fully defined graphs, and searches for a 3-decomposition for a fully defined graph exhaustively. If it finds a decomposition (T, C, M) (note that it is necessary that $B(T) \cap E(G) = \emptyset$ due to Theorem 2), then it sends to the solver OneTwo(T); otherwise, it does nothing.

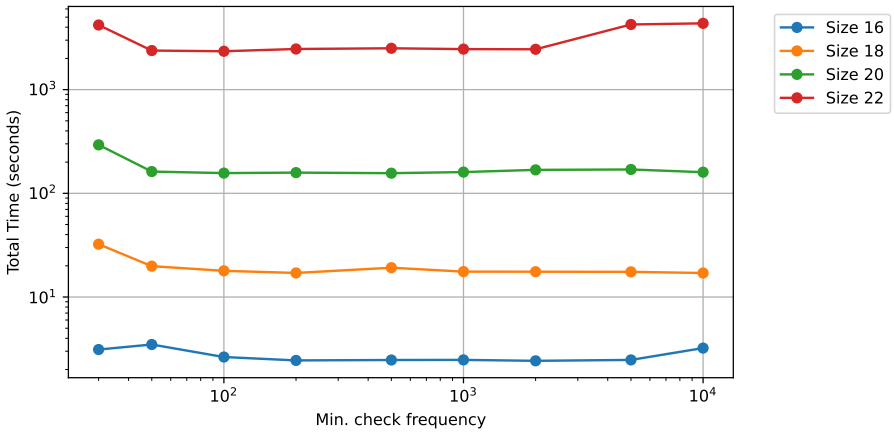


Figure 3 Time spent in solving the case of $n \in \{16, 18, 20, 22\}$ with the frequency of minimality check $f \in \{30, 50, 100, 200, 500, 1000, 2000, 5000, 10000\}$.

3.2 Results

We implemented the approach with the four propagators as mentioned above and verified the conjecture for every even $4 \leq n \leq 28$. The time spent in each n is shown in Table 1[1]. The frequency for the minimality check for all cases in the table is the default value of 30. In Table 2, we provide statistics of the propagators for the bigger cases. In figure 3, we compare the time spent in solving the middle cases with the minimality check invoked with different frequencies. As we can see, in general, the time first decreases and then increases as we invoke the minimality check more seldom, but the difference is not significant.

Table 1 Time spent verifying that all connected cubic graphs with n vertices have a 3-decomposition.

n	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**	**20**	**22**	**24**	**26**	**28**
time	$<$ 1ms	2.6ms	6.1ms	45ms	0.23s	0.92s	3.1s	32s	4.9m	1.2h	20h	26d	4.0y

▶ **Theorem 3.** *The 3DC holds for all connected cubic graphs up to* 28 *vertices.*

4 Search for forbidden (partial) subgraphs

In this section, we search for patterns that cannot appear in a minimal counterexample to the conjecture. This is useful because if we know certain patterns cannot appear in a counterexample, we can close off the search branch once we detect the subgraph in the partially defined graphs and thereby shorten the search time. We search for particular pairs

[1] The computational tasks throughout the paper were carried out on a Sun Grid Engine (SGE) cluster consisting of heterogeneous machines running Ubuntu 18.04.6 LTS. The cluster contains nodes with the following architectures: 2× Intel Xeon E5540 with 2.53 GHz Quad Core, 2× Intel Xeon E5649 with 2.53 GHz 6-core, 2× Intel Xeon E5-2630 v2 with 2.60GHz 6-core, 2× Intel Xeon E5-2640 v4 with 2.40GHz 10-core and 2× AMD EPYC 7402 with 2.80GHz 24-core. All relevant scripts and code can be found at `https://doi.org/10.5281/zenodo.15623903`.

Table 2 Statistics of the propagators for the cases $n \in \{18, 20, 22, 24, 26, 28\}$ of Table 1. min. is short for minimality check, forb. is short for forbidden subgraph check, span. is short for spanning tree check, and 3-dec. is short for 3-decomposition check. For each n, three columns of data are given. The first column shows the time each propagator uses. The second column shows the percentage that time takes up in the total solving time. The third column shows the percentage of the propagator calls that yield learned clauses.

n	**18**			**20**			**22**		
min.	11s	35%	67%	1.5m	29%	52%	15m	21%	35%
forb.	10s	32%	29%	2.0m	41%	15%	28m	40%	5.0%
span.	93ms	2.9‰	1.9‰	1.7s	5.6‰	< 1‰	34s	8.0‰	< 1‰
3-dec.	0.28s	8.7‰	100%	7.9s	2.7%	100%	4.2m	6.1%	100%
n	**24**			**26**			**28**		
min.	2.4h	12%	21%	3.0d	11%	11%	98d	6.8%	9.9%
forb.	4.7h	23%	1.2%	2.1d	7.8%	1.2%	80d	5.5%	9.9‰
span.	10m	8.5‰	< 1‰	2.5h	3.8‰	< 1‰	3.8d	2.6‰	< 1‰
3-dec.	2.4h	12%	100%	8.5d	32%	100%	2.9y	73%	100%

of patterns called *reducible extensions*, which are pairs of subcubic graphs X and Y with $|V(X)| < |V(Y)|$, such that if we have a counterexample containing Y, then we can always obtain a smaller counterexample by replacing Y with X. This way, Y cannot appear as a subgraph in a minimal counterexample to the 3-decomposition conjecture.

We start with reviewing some existing concepts and definitions that we built our concepts on. In particular, we introduce the concept of *templates*, *reducible extensions*, and *reducible templates*, and a sufficient condition for automatically testing whether a pair of subcubic graphs belongs to the latter. Then, we show that the said condition can be relaxed. Finally, we implement a search procedure using both the relaxed condition and SMS and obtain all reducible templates up to a certain size measure.

4.1 Templates, replacement, coloring, and possible behaviors

In this section, we review some useful definitions and theorems from Bachtler et al. [4]. Some of the definitions have been slightly modified to better formulate the concepts we work with.

Templates. A *template* is a graph X whose vertex set is partitioned into a set of *inner vertices* $I(X)$ and a set of *outer vertices* $O(X)$, such that all inner vertices have 3 neighbors and all outer vertices have 1 neighbor. For any $x \in O(X)$, we write $a_X(x)$ to denote the unique vertex in $I(X)$ that x is adjacent to. The *order* of a template is defined as its number of vertices, and the *index* of a template is its number of outer vertices. Let X and Y be two templates with $O(X) = O(Y)$. Let G be a cubic graph. We say template X is in G or G contains X as a template if there exists a mapping $\varphi : V(X) \to V(G)$ such that for any $x, y \in V(X)$, we have if $x \in I(X)$, then $\varphi(x) \neq \varphi(y)$ and $xy \in E(X)$ if and only if $\varphi(x)\varphi(y) \in E(G)$. We call such a mapping a *witness*.

Intuitively, a template is intended as a representation of "a piece of" a cubic graph where the inner vertices together with the edges among them form an induced subgraph of the entire graph, and the outer vertices correspond to all the other vertices that the vertices in this induced subgraph are adjacent to. We have defined what it means to be "a piece of" a cubic graph, and now we define what it means to replace one piece of a cubic graph with another.

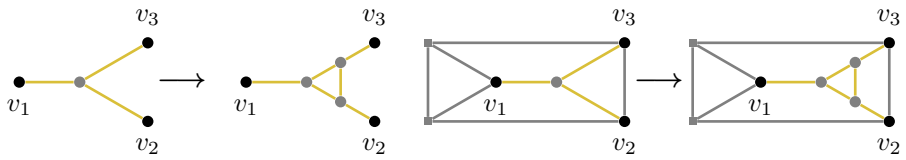


Figure 4 Examples of templates and replacement. On the left are two templates with the same set of outer vertices. The outer vertices are in black and the inner vertices gray. On the right, we show the result of replacing the first template with the second in a cubic graph of original size 6. The vertices that are not part of the templates are shown as gray squares.

Replacement. Suppose G contains X as a template. Then we can replace X in G with Y and get $G_\varphi[X \to Y]$ where $V(G_\varphi[X \to Y]) := (V(G)/\varphi(I(X))) \cup I(Y)$ and $E(G_\varphi[X \to Y]) := \{\, uv : u, v \in V(G)/\varphi(I(X)), uv \in E(G) \,\} \cup \{\, xy : x, y \in I(Y), xy \in E(Y) \,\} \cup \{\, \varphi(x)y : x \in O(Y), y \in I(Y), xy \in E(Y) \,\}$. When it is not necessary to spell out the mapping φ, we also write $G[X \to Y]$ for $G_\varphi[X \to Y]$. A pair (X, Y) of templates with $O(X) = O(Y)$ is called an (X, Y)-*transformation*. Transformations are called *extensions* if $|V(X)| < |V(Y)|$. The *order* and *index* of an extension (X, Y) is defined as the order and index of Y.

Figure 4 contains an example for the concepts of template and replacement. To reformulate the goal mentioned at the beginning of this section, we want to find pairs of templates X and Y such that if we have a counterexample containing Y, then we can always obtain a smaller counterexample by replacing Y with X. Now, we rigorously define such pairs of templates in the notion of *reducible extensions*.

Reducible extensions. An (X, Y)-extension is *3-compatible* if for every cubic graph G with a 3-decomposition and for every extension $H := G[X \to Y]$ of G, H also has a 3-decomposition. Given a (Y, X)-reduction, a mapping $\varphi : V(Y) \to V(G)$ is called *permissible* if $\varphi(x) = \varphi(y)$ implies $a_X(x) \neq a_X(y)$ for all $x, y \in O(Y)$. A 3-compatible (X, Y)-extension is called a *reducible extension* if, for any connected cubic graph G containing Y as a template witnessed by a permissible mapping for (Y, X), we have that $G_\varphi[Y \to X]$ is also a connected cubic graph. Such a template Y is called *reducible*. The reason why we can forbid reducible templates in our search for counterexamples to the 3DC is formally stated in the following Theorem 4.

▶ **Theorem 4.** *Let (X, Y) be a reducible extension and G be a connected cubic graph that contains Y witnessed by a permissible mapping for (Y, X). Then G cannot be a minimal counterexample to the 3-decomposition conjecture.*

Theorem 4 states why reducible extensions are useful, but we still lack a way to compute them, since a template can appear in infinitely many cubic graphs and thus it is impossible to test if two templates stand in the relation of reducible extensions by enumerating all cubic graphs they appear in. This problem is solved in Bachtler et al. in the following way. First, note the following proposition that allows us to filter out reducible extensions from 3-compatible extensions.

▶ **Proposition 5.** *Let (X, Y) be a 3-compatible extension. If X is connected, then (X, Y) is a reducible extension.*

Now, our task is shifted to finding 3-compatible extensions. Bachtler et al. introduced a sufficient condition for 3-compatible extension that only requires analysing the structure of the templates themselves. The idea is that even though a template X can appear in infinitely

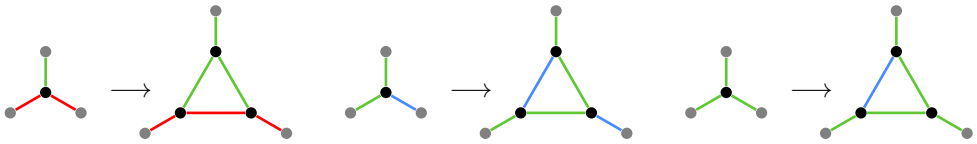


Figure 5 An example of applying Theorem 8 to determine membership of 3-compatibility. We use the same two templates as in Figure 4. We list all three possible behaviors of the smaller template (modulo reordering of the outer vertices), and each of the corresponding possible behaviors of the big template is given on the right.

many cubic graphs, there are only finitely many ways a template X can be decomposed as a part of a 3-decomposition. We call these ways *possible behaviors* of X. So if for each the possible behavior of X, there is a corresponding possible behavior of Y, such that the 3-decomposition still holds after replacement, then we can safely say that (X, Y) is a 3-compatible extension. This reasoning is rigorously formulated in Proposition 6, Corollary 7, and Theorem 8. We write RGB for the set $\{\text{red}, \text{green}, \text{blue}\}$. A *3-coloring* of a graph G is a mapping $cl : E(G) \to \text{RGB}$. A 3-decomposition can be seen as a 3-coloring, where the green edges represent the edges of the spanning tree, the red edges the cycles, and the blue edges the matching. We have the following proposition.

▶ **Proposition 6.** *A 3-coloring of a cubic graph is a 3-decomposition if and only if*

1. *The list of colors of the 3 incident edges of every vertex are either (green, green, green), (blue, green, green), and (green, red, red), and*
2. *the green edges form a spanning tree over the cubic graph.*

Proposition 6 gives us the following corollary.

▶ **Corollary 7.** *Given a template X and a 3-coloring over $E(X)$, the following conditions are necessary for the coloring to be a 3-decomposition over some cubic supergraph of X, restricted to $E(X)$.*

1. *The list of colors of the 3 incident edges of every $v \in I(X)$ are either [green, green, green], [blue, green, green] or [red, red, green], modulo reordering.*
2. *The green edges form a spanning forest, and each tree in the spanning forest contains at least one vertex from $O(X)$.*

Possible behaviors. We call any 3-coloring of X that satisfies the conditions in Corollary 7 a *possible behavior* of X. Let X, Y be two templates with $O(X) = O(Y)$ and $|X| < |Y|$. For any $v \in O(X) = O(Y)$, write $c(v)$ for the color of the unique edge incident to v in the coloring c. For a possible behavior c of X, define a partition P_c of $O(X)$ in which two vertices in $O(X)$ are in the same equivalence class if they are in the same tree of the spanning forest given c.

▶ **Theorem 8.** *Let X and Y be templates with $O(X) = O(Y) = O$ and $|V(X)| < |V(Y)|$. (X, Y) is a 3-compatible extension if for any possible behavior c of X, there is a possible behavior c' of Y satisfying the following conditions.*

1. *For any $v \in O$, $c'(v) = c(v)$.*
2. *$P_{c'} = P_c$.*

An example of applying Theorem 8 to determine membership of 3-compatibility is shown in Figure 5

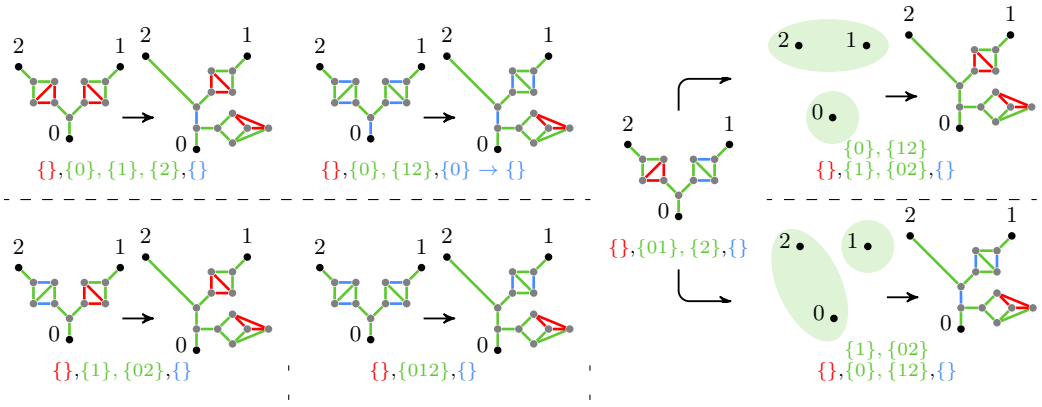


Figure 6 An example showing how Theorem 13 can be applied to affirm a reducible extension. All possible behaviors of the smaller template (on the left) come down to five different configurations in terms of the set of outer vertices adjacent to a red edge (displayed in red), the partition the green forest gives rise to over the outer vertices (displayed in green), and the set of vertices adjacent to a blue edge (displayed in blue). The larger template (on the right) has a possible behavior for four of the configurations with a spanning forest that gives rise to the same partition over the outer vertices. For the last configuration, the larger template does not have this, but instead has a corresponding possible behavior for each possibility of how the rest of the spanning tree can be.

4.2 Relaxing the Conditions

In this section, we show that the condition in Theorem 8 can be further relaxed. The key observation is as follows. Given templates X and Y with $O(X) = O(Y)$ and $|V(x)| < |V(Y)|$ and a possible behavior c of X, (X, Y) can still be a 3-compatible extension even if Y does not have a possible behavior c' such that $P_c = P_{c'}$. In fact, Y only needs to have a corresponding possible behavior for each of the possibilities that the rest of the spanning tree can be. We introduce Lemmas 9 and 11 that set the foundation for Definition 12, in which we rigorously characterize the description above. Then, we end this section with the relaxed condition and proof of its validity. Due to space limits, the proofs for this section are shown in Appendix A.

▶ **Lemma 9.** *Given two trees T_1 and T_2 with $E(T_1) \cap E(T_2) = \emptyset$, the combined graph $G := (V(T_1) \cup V(T_2), E(T_1) \cup E(T_2))$ is a tree if and only if $|V(T_1) \cap V(T_2)| = 1$.*

▶ **Definition 10.** *Let $\mathcal{S} = S_1, S_2, ..., S_n$ be a finite collection of sets. $\mathcal{S}$ is* mergeable *if there exists a permutation a of $[n]$ such that for each $i \in [n-1]$, $|(\bigcup_{j=1}^{i} S_{a(j)}) \cap S_{a(i+1)}| = 1$.*

▶ **Lemma 11.** *Let S be a set of vertices and $T_1, T_2, ..., T_n$ a set of trees such that $V(T_i) \cap S \neq \emptyset$ and $V(T_i) \cap V(T_j) \subseteq S$ for distinct $1 \leq i, j \leq n$. Define the combined graph $G := (\bigcup_{i=1}^{n} V(T_i), \bigcup_{i=1}^{n} E(T_i))$. Then G is a tree if and only if $\mathcal{S} := \{ S_i := V(T_i) \cap S : i \in [n] \}$ is mergebale.*

▶ **Definition 12.** *Given a partition $\mathcal{S}$ of a set S, we write $Comp(\mathcal{S})$ to denote the set of partitions $\mathcal{S}'$ of S such that $\mathcal{S} \cup \mathcal{S}'$ is mergeable.*

Now we present the following Theorem 13, which relaxes Theorem 8.

▶ **Theorem 13.** *Let X and Y be templates with $O(X) = O(Y) = O$ and $|V(X)| < |V(Y)|$. (X, Y) is a 3-compatible extension if for any possible behavior c of X and partition $P \in Comp(P_c)$, there is a possible behavior c' of Y satisfying the following conditions.*

1. *For any $v \in O$, $c'(v) =$ green *if* $c(v) =$ green*; $c'(v) =$ red *if and only if* $c(v) =$ red.*
2. *$P_{c'} \in Comp(P)$.*

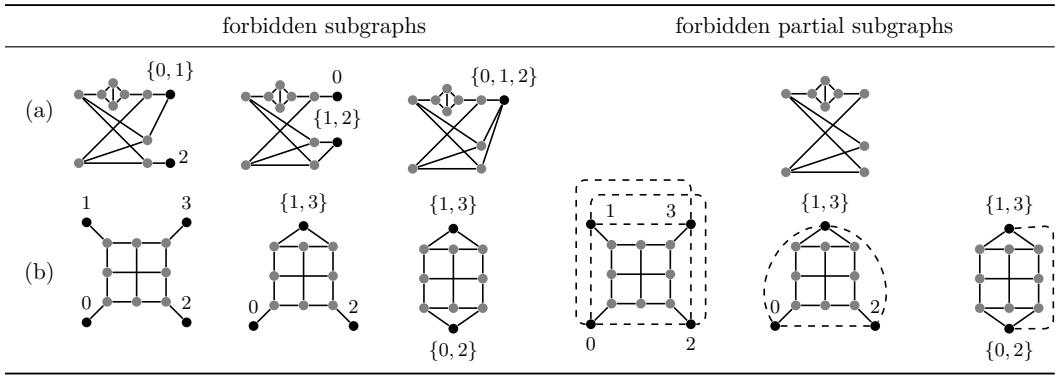


Figure 7 Two examples of the set of forbidden (partial) subgraphs we obtain from reducible extensions. Both of the reducible extensions are new discoveries, and are shown in Figure 9.

Theorem 13 and Proposition 5 give the criterion by which we search for reducible extensions using a C++ program. The program itself and the related toolchain will be explained in Section 4.4. An example of applying Theorem 13 to affirm a reducible extension is shown in Figure 6.

4.3 Converting reducible extensions to forbidden (partial) subgraphs

In this section, we explain how to convert reducible extensions into forbidden subgraphs and partial subgraphs. Recall that templates are intended as a representation of "a piece of" a graph where the inner vertices together with the edges among them form an induced subgraph of the entire graph, and the outer vertices correspond to all the other vertices that the vertices in this induced subgraph are adjacent to. The restriction that each outer vertex is adjacent to a unique inner vertex and none of the outer vertices allows us to consider fewer cases when enumeration such "pieces of a graph", but this also means that we need to consider the possibility that multiple outer vertices can represent the same vertex in the entire graph. We now describe in detail how a reducible extension is converted to a set of forbidden subgraphs and forbidden partial subgraphs, respectively.

For any template Z, let $\mathrm{Part}(Z)$ be the partition of $O(Z)$ such that two outer vertices are in the same equivalence class if and only if the inner vertices that they are uniquely adjacent to are the same. Define $\overline{\mathrm{Part}}(Z)$ to be the set obtained by removing singletons from $\mathrm{Part}(Z)$, and write $\overline{O}(X)$ to denote $\bigcup \overline{\mathrm{Part}}(Z)$. For both cases, we start with a reducible extension (X, Y) with $O(X) = O(Y)$. Note that if distinct $u, v \in O(X)$ are in the same equivalence class of $Part(X)$ and G is a graph containing Y where u and v correspond to the same vertex in G, then $G[Y \to X]$ is not a simple graph, since it might contain parallel edges. In other words, the existence of the (partial) subgraph obtained by mapping u and v to the same vertex in a minimal counterexample is not ruled out by (X, Y).

To deduce the set of forbidden subgraphs from (X, Y), we enumerate all partitions P of $O(X)$ such that if distinct $u, v \in O(X)$ are in the same equivalence class in $\mathrm{Part}(X)$, then u and v are not in the same equivalence class of P. For each P, we construct a simple graph G_P where $V(G_P) := P \cup I(Y)$ and $E(G_P) := \{\, uv : u, v \in I(Y), uv \in E(Y) \,\} \cup \{\, pv : p \in P, v \in I(Y), \exists u \in p.\, uv \in E(Y) \,\}$.

To deduce the set of forbidden partial subgraphs from (X, Y), we enumerate all partitions $\overline{P}$ of $\overline{O}(X)$ such that if distinct $u, v \in \overline{O}(X)$ are in the same equivalent class in $\overline{\mathrm{Part}}(X)$, then u and v are not in the same equivalent class of $\overline{P}$. For each $\overline{P}$, we construct a partially defined

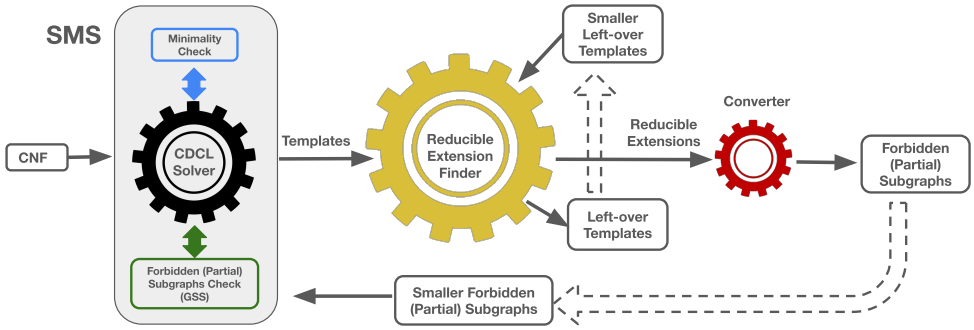


Figure 8 The toolchain for searching for forbidden (partial) subgraphs.

graph $\overline{G_P}$ where $V(\overline{G_P}) := \overline{P} \cup I(Y)$, $E_d(\overline{G_P}) := \{\, uv : u, v \in I(Y), uv \in Y \,\} \cup \{\, pv : p \in \overline{P}, v \in I(Y), \exists u \in p.\, uv \in E(Y) \,\}$, and $E_n(\overline{G_P}) := \{\, uv : u, v \in I(Y), uv \notin E(Y) \,\} \cup \{\, pv : p \in \overline{P}, v \in I(Y), \forall u \in p.\, uv \notin E(Y), \forall x \in O(X)/\overline{O}(X).\, xv \notin E(Y) \,\}$.

We implement a toolchain for computing forbidden (partial) subgraphs and explain it in detail in the next section. For now, we give two examples for the set of forbidden (partial) subgraphs we obtain from newly discovered reducible extensions in Figure 7.

4.4 Computing reducible templates

In this section, we explain the toolchain we built for computing reducible extensions. The toolchain is visualized in Figure 8. It is run iteratively, each time searching for reducible extensions of a fixed pair of order and index. The precise sequence in which the iterations are carried out is shown in Table 3. The reason for this sequence is to get reducible templates with as few vertices and edges as possible. In total, three tools are used.

The first tool is SMS with the minimality check and the forbidden (partial) subgraph check activated. We set SMS to enumerate all graphs with n vertices and give SMS the following CNF formula.

$$F_{\text{temp}}(k,n) := \left(\bigwedge_{i\in[k]} \sum_{j\in[k],j\neq i} \neg e(i,j) \right) \wedge \left(\bigwedge_{i\in[k]} \sum_{j\in[k+1,n]} e(i,j) = 1 \right) \wedge \left(\bigwedge_{i\in[k+1,n]} \sum_{j\in[n],j\neq i} e(i,j) = 3 \right).$$

Here we postulate that vertices $v \in [k]$ are the k outer vertices and the rest are inner vertices. $F_{\text{temp}}(k,n)$ states the conditions for a graph to be a template with index k and order n.

The second tool is a *reducible extensions finder*, which is a separate C++ program that takes in templates and finds reducible extensions according to Theorem 13. Here, smaller templates that are connected (see Proposition 5) and that are not the bigger graph for any reducible extensions found in previous iterations (shortened as *left-overs* in the figure) are used as candidates for the smaller template, and the templates generated by SMS are used as candidates for the bigger graph.

The last tool is a *converter* that converts reducible extensions found into forbidden (partial) subgraphs, which we have explained in Section 4.3.

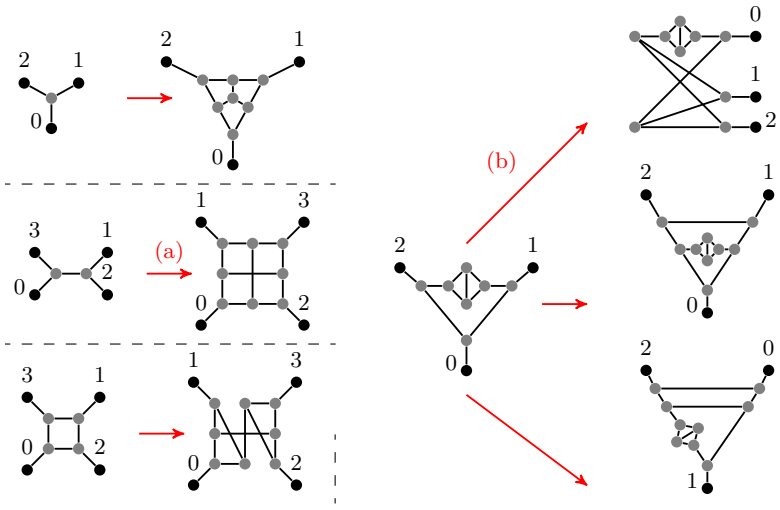


Figure 9 Newly discovered reducible extensions up to order 12. Reducible extensions with index < 3 are not shown, since all reducible extensions (X, Y) we found up to order 18 of index 2 have the unique smallest template of index 2 as X, and those with index 1 have the unique smallest template with index 1 as X.

By running this setup, we can determine all reducible templates we were looking for, and we arrive at the following main result of this section.

▶ **Theorem 14.** *The numbers of reducible templates of various indices and orders up to index* 6 *and order* 18 *are as shown in Table 3.*

Figure 9 shows the newly discovered reducible extensions up to order 12.

Table 3 The results of exhaustive search for reducible templates Y with index ≤ 6, up to order 18. Each row represents a different value for the number of vertices of the larger graph. Each row is divided into six groups, each group representing a different number of outer vertices. There are three numbers in each group. The first is the number of reducible extensions Y found. The second is the number of leftover templates. The third is the total number of templates generated by SMS. This table also shows the order of computation: on the same row, the groups are computed from left to right; across rows, the groups are computed from top to bottom. All graphs in the table can be found in the supplementary material.

$\|V(Y)\|$	$\|O(Y)\|$																	
	6			**5**			**4**			**3**			**2**			**1**		
4	-			-			-			0	1	1	-			-		
6	-			-			0	1	1	1	0	1	0	1	1	0	1	1
8	-			0	1	1	0	1	1	1	1	3	2	1	3	0	0	0
10	0	2	2	0	2	3	2	2	6	1	2	4	3	0	4	1	0	2
12	0	5	8	0	6	9	3	6	12	4	2	11	2	0	5	1	0	2
14	0	20	28	10	18	34	9	7	25	4	0	10	1	0	5	0	0	3
16	3	84	108	48	24	92	15	2	35	2	2	16	1	0	7	0	0	2
18	217	138	414	33	28	111	12	7	53	4	0	21	0	0	7	0	0	4

5 Speeding up the search with forbidden (partial) subgraphs

In this section, we test the effectiveness of forbidding (partial) subgraphs on shortening the search time for $n \in \{16, 18, 20, 22, 24\}$. We first gather all forbidden subgraphs (forbidden partial subgraphs, respectively) and order them in ascending order of the number of vertices. We then take the first $5 \cdot x$ forbidden (partial) subgraphs in the ordered list for $x \in [0, 25]$ and forbid them during our search. We also set the frequency at which the forbidden (partial) subgraphs check is called to freq $\in \{30, 100, 200, 500, 1000, 2000, 5000, 100000\}$. The forbidden (partial) subgraph check is called once per freq many edge variable decisions probabilistically. The time spent in search in each case is shown in Figure 10. The figure tells us the following.

1. Forbidding (partial) subgraphs generally reduces the search time, with the effect stronger as n increases.
2. The search time first decreases, then increases as the number of forbidden (partial) subgraphs grows. For subgraphs, the optimal number to forbid is around 25, and for partial subgraphs is 5.
3. Fixing the number of forbidden structures, the search with subgraphs forbidden is, in general, faster than the search with partial subgraphs forbidden. This is the result of two competing forces. On the one hand, partial subgraphs are more concise, with a forbidden partial subgraph more likely to rule out a partial assignment than a forbidden subgraph. On the other hand, partial subgraphs are represented as edge-labeled graphs in the Glasgow Subgraph Solver, which might lead to longer subgraph isomorphism testing times compared to unlabeled graphs. Apparently, the latter force overpowers the former in practice.
4. Except for $n \in \{16, 18\}$ with forbidden partial subgraphs, where the frequent call of the forbidden partial subgraph propagators visibly drags down the search speed, there is no uniform pattern between search time and the frequencies of subgraph isomorphism testing. This might be because, while subgraph isomorphism testing is time-consuming, frequent testing allows for earlier pruning of the search tree.

6 Conclusion and future work

In this work, we developed a SAT-based method enhanced by constraint programming techniques to verify the 3-Decomposition Conjecture for connected cubic graphs up to 28 vertices. We combined specialized propagators with forbidden subgraph checks to effectively prune the search space and reduce computation time. Our experiments provided clear insights into the trade-offs between the cost of constraint propagation and the benefits of early search pruning.

For future research, implementing a systematic and automated feature to gather and verify proofs and certificates generated by different tools in the SMS framework would be interesting. We explain here how it could work with respect to the tool chain presented in this paper.

The SMS uses CaDiCaL as the SAT solver, which is amenable to the production of DRAT [12]. To take into account the learned clauses produced by various specialized propagators, we can first run SMS with the propagators while collecting all learned clauses and then run CaDiCaL a second time with proof logging, feeding it the original formula together with all the recorded learned clauses. The DRAT proof produced this way can then be independently checked using verified proof checkers like DRAT-trim [24].

Besides generating a DRAT proof for the SAT solving process, we also need to verify that the propagators work the way intended and learn correct clauses. This is also doable given the SMS framework. The propagators are essentially approximation algorithms for NP problems

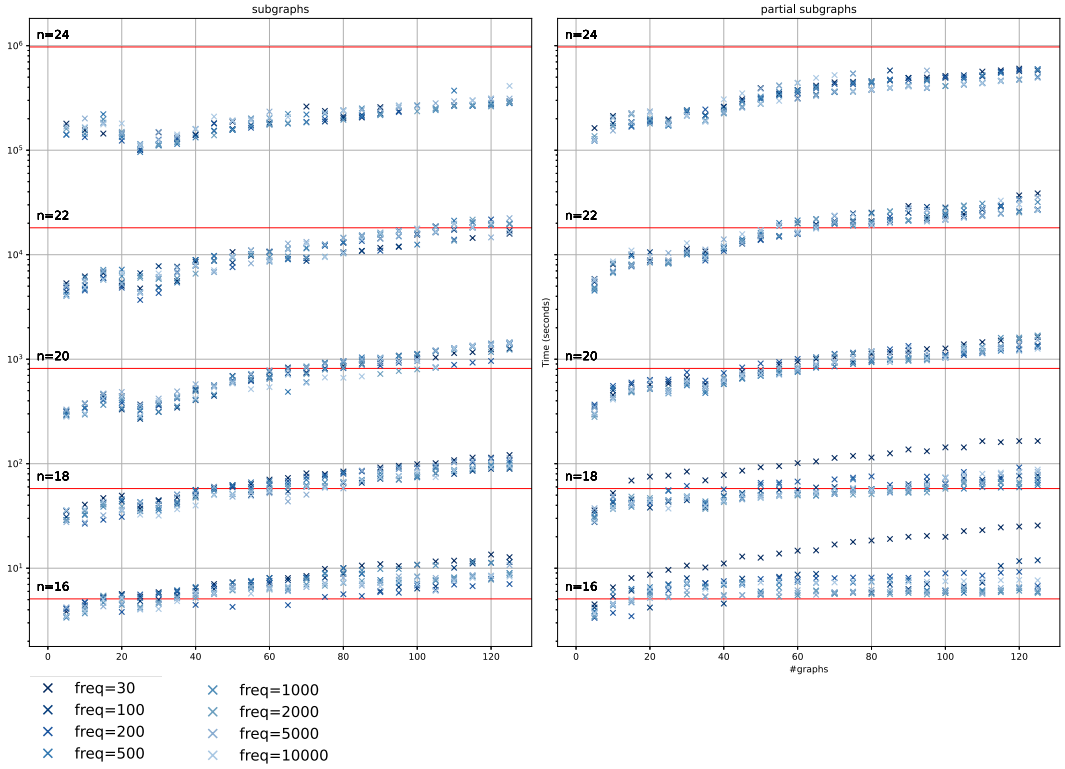


Figure 10 Time spent in the search for $n \in \{16, 18, 20, 22, 24\}$ forbidding different numbers of (partial) subgraphs with the forbidden (partial) subgraph check triggered with different frequencies. The data from forbidding subgraphs is shown on the left, and the data for forbidding partial subgraphs is shown on the right. The horizontal axis corresponds to the number of (partial) subgraphs forbidden, and the vertical axis corresponds to the time spent in search. Data points are marked as 'x', with the darker ones corresponding to the cases where the propagator is called more often. For each n, a red horizontal line is drawn at the height corresponding to the time spent in search without forbidding any (partial) subgraphs.

and are reliable when the answer is "yes" – they only learn clauses under that condition. Hence, a certificate can be generated each time a clause is learned, and these can be checked with a separate program. Note that NP certificates are simple to verify. Specifically for the tool chain in this paper, the minimality check can produce "nc-certificates" (see [18]); the forbidden (partial) subgraph check can output the mapping that witnesses subgraph isomorphism; the spanning tree check can output the spanning tree found that has $B(T) = \emptyset$; the 3-decompositon check can output 3-decompositions. In general, each propagator used in SMS can be equipped with a certificate-generating option and an accompanying verifier for verifying the certificates it produces.

Realizing a systematic way of verification for the SMS framework requires significant engineering that we leave for future work.

References

1 Elham Aboomahigir, Milad Ahanjideh, and Saieed Akbari. Decomposing claw-free subcubic graphs and 4-chordal subcubic graphs. *Discret. Appl. Math.*, 296:52–55, 2021. doi:10.1016/J.DAM.2020.01.016.

2 Saieed Akbari, Tommy R. Jensen, and Mark H. Siggers. Decompositions of graphs into trees, forests, and regular subgraphs. *Discret. Math.*, 338(8):1322–1327, 2015. doi:10.1016/J.DISC.2015.02.021.

3 Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 167–180. Springer, 2009. doi:10.1007/978-3-642-02777-2_18.

4 Oliver Bachtler and Irene Heinrich. Reductions for the 3-decomposition conjecture. *Procedia Computer Science*, 223:96–103, 2023. XII Latin-American Algorithms, Graphs and Optimization Symposium (LAGOS 2023). doi:10.1016/j.procs.2023.08.218.

5 Curtis Bright, Kevin K. H. Cheung, Brett Stevens, Ilias S. Kotsireas, and Vijay Ganesh. A sat-based resolution of Lam's problem. 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 3669–3676. AAAI Press, 2021. doi:10.1609/AAAI.V35I5.16483.

6 Peter J. Cameron. Research problems from the BCC22. *Discrete Mathematics*, 311(13):1074–1083, 2011. Selected Papers from the 22nd British Combinatorial Conference. doi:10.1016/j.disc.2011.02.024.

7 Michael Codish, Michael Frank, Avraham Itzhakov, and Alice Miller. Computing the Ramsey number R(4, 3, 3) using abstraction and symmetry breaking. *Constraints An Int. J.*, 21(3):375–393, 2016. doi:10.1007/S10601-016-9240-3.

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

9 Genghua Fan and Chuixiang Zhou. Hoffmann-Ostenhof's 3-decomposition conjecture. *Discrete Mathematics*, 348(7):114454, 2025. doi:10.1016/j.disc.2025.114454.

10 Johannes K. Fichte, Markus Hecher, Daniel Le Berre, and Stefan Szeider. The silent (r)evolution of SAT. *Communications of the ACM*, 66(6):64–72, June 2023. doi:10.1145/3560469.

11 The OEIS Foundation. Number of connected cubic unlabeled graphs with n nodes. https://oeis.org/A002851.

12 Marijn Heule, Warren Hunt, Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving*, pages 269–284, Cham, 2017. Springer International Publishing.

13 Marijn J. H. Heule, Oliver Kullmann, and Armin Biere. Cube-and-conquer for satisfiability. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 31–59. Springer, 2018. doi:10.1007/978-3-319-63516-3_2.

14 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.

15 Arthur Hoffmann-Ostenhof, Tomáš Kaiser, and Kenta Ozeki. Decomposing planar cubic graphs. *Journal of Graph Theory*, 88(4):631–640, 2018. doi:10.1002/jgt.22234.

16 Markus Kirchweger, Tomáš Peitl, and Stefan Szeider. Co-certificate learning with SAT modulo symmetries. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*, pages 1944–1953. ijcai.org, 2023. Main Track. doi:10.24963/IJCAI.2023/216.

17 Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, LIPIcs, pages 39:1–39:17. Dagstuhl, 2021. doi:10.4230/LIPIcs.CP.2021.34.

18 Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation and enumeration. *ACM Trans. Comput. Log.*, 25(3):1–30, 2024. doi:10.1145/3670405.

19 Panpan Li and Wenzhong Liu. Decompositions of cubic traceable graphs. *Discuss. Math. Graph Theory*, 40(1):35–49, 2020. doi:10.7151/DMGT.2132.

20 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.

21 Ciaran McCreesh, Patrick Prosser, and James Trimble. The Glasgow subgraph solver: Using constraint programming to tackle hard subgraph isomorphism problem variants. In Fabio Gadducci and Timo Kehrer, editors, *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings*, volume 12150 of *Lecture Notes in Computer Science*, pages 316–324. Springer Verlag, 2020. doi:10.1007/978-3-030-51372-6_19.

22 Bernardino Romera-Paredes et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi:10.1038/S41586-023-06924-6.

23 Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024. doi:10.1038/S41586-023-06747-5.

24 Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer Verlag, 2014. doi:10.1007/978-3-319-09284-3_31.

25 Tianwei Zhang and Stefan Szeider. Searching for smallest universal graphs and tournaments with SAT. 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 39:1–39:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CP.2023.39.

A Proofs for Section 4.2

Proof of Lemma 9.

Proof. We distinguish the following three cases. Suppose $|V(T_1) \cap V(T_2)| = 0$. Then G is clearly a forest consisting of two disconnected trees. Suppose $|V(T_1) \cap V(T_2)| > 1$, then there exists distinct $u, v \in V(T_1) \cap V(T_2)$. Since T_1 and T_2 are trees, it follows that there is a unique path between u and v in both T_1 and T_2. Given that $E(T_1) \cap E(T_2) = \emptyset$, these two paths are distinct and therefore together form a cycle. Suppose $V(T_1) \cap V(T_2) = \{s\}$. It suffices to show that for any distinct $u, v \in V(G)$, there is a unique path between u and v. Without loss of generality, assume $u \in V(T_1)$. We distinguish the following two cases.

1. $v \notin V(T_1)$. Since $\{s\}$ is separating in G, any path between u and v consists of a path between u and s in T_1 and a path between v and s in T_2. Since both of the latter exist and are unique, it follows that there is a unique path between u and v in G.
2. $v \in V(T_1)$. Then there exists a unique path between u and v in T_1. Suppose there is a different path between u and v in G. Then this path must contain at least one edge from $E(T_2)$ and therefore a vertex $w \in V(T_2)/\{s\}$. This means that this path consists of a path p_1 between u and w and a path p_2 between v and w. Since $\{s\}$ is separating in G, it follows that s is on both paths, and therefore s is visited twice on the path. Contradiction. ◀

Proof of Lemma 11.

Proof. We prove the statement from both directions.

1. Suppose $\mathcal{S}$ is mergeable and a is witnessing permutation of $[n]$. Then we can show by induction on i that $G_i := (\bigcup_{j=1}^{i} V(T_j), \bigcup_{j=1}^{i} E(T_j))$ is a tree. The base case where $i = 1$ is true by definition. The induction step is valid because of Lemma 9. This means that in particular $G = G_n$ is a tree.
2. Suppose G is a tree. We show by induction on the size of non-empty $M \subseteq [n]$ that if $G_M := (\bigcup_{i \in M} V(T_i), \bigcup_{i \in M} E(T_i))$ is a tree, then $\mathcal{S}_M := \{ S_i : i \in M \}$ is mergeable. The base case where $|M| = 1$ is trivially true. Define $I_M^x := V(T_x) \cap (\bigcup_{i \in M \setminus \{x\}} V(T_i))$. For the induction step, we first show that if G_M is a tree and $|M| > 1$, then there is $x \in M$ such that $|I_M^x| = 1$. Since G_M is connected, it follows that $|I_M^x| \geq 1$ for all $x \in M$. Now assume the contrary, that is, $|I_M^x| > 1$ for all $x \in M$. We will see this assumption allows us to find a cycle in G_M, which contradicts the premise that G_M is a tree. To illustrate this, we build sequences $b_1, b_2, ...b_{n+1} \in M$ and $c_1, c_2, ..., c_{n+1}$ where $c_i \in T_{b_i}$ for all $i \in [n+1]$. Start with any $b_1 \in M$, and let $c_1 \in I_M^{b_1}$. Given b_i and c_i, pick $b_{i+1} \neq b_i$ such that $c_i \in T_{b_{i+1}]}$ and $c_{i+1} \in I_M^{b_{i+1}} \setminus \{c_i\}$. According to the pigeonhole principle, there exists $x, y \in [n+1]$, $x < y$ such that $b_x = b_y$. Now we can obtain a cycle in G_M by concatenating the unique path from c_x to c_{x+1} in $T_{b_{x+1}}$, the unique path from c_{x+1} to c_{x+2} in $T_{b_{x+2}}$,..., the unique path from c_{y-2} to c_{y-1} in $T_{b_{y-1}}$ and the unique path from c_{y-1} to c_x in $T_{b_y} = T_{b_x}$. So far, we have established that if G_M is a tree and $|M| > 1$, then there is $x \in M$ such that $|I_M^x| = 1$. This means that $G_{M \setminus \{x\}}$ is a tree. By the induction hypothesis, there is a mapping $a : [|M| - 1] \to M \setminus \{x\}$ that witness the fact that $\mathcal{S}_{M \setminus \{x\}}$ is mergable. Now it is easy to see that $a' : [|M|] \to M$ defined as $a'(i) := a(i)$ for all $1 \leq i < |M|$ and $a'(|M|) := x$ witnesses the fact that $\mathcal{S}_M$ is mergable. Now we have established with induction that if $G_M := (\bigcup_{i \in M} V(T_i), \bigcup_{i \in M} E(T_i))$ is a tree, then $\mathcal{S}_M := \{ S_i : i \in M \}$ is mergeable. In particular, if G is a tree, then $\mathcal{S}$ is mergeable.

◀

Proof of Theorem 13.

Proof. Suppose X is a subgraph of a connected cubic graph G with a 3-decomposition c_G. Then $c := c_G|_{E(X)}$ is a possible behavior of X. Note that the green edges of $c_G|_{E(G/X)}$ form a spanning forest in G/X. Let P be the partition of $O(X)$ that the green edges of c_G in G/X induce. Since c_G is a 3-decomposition, it follows that $P \in \mathrm{Comp}(P_c)$. Now, suppose c' is a possible behavior of Y that satisfies the conditions above. It suffices to show that

$$d : E(G[X \to Y]) \to \mathrm{RGB}, \qquad d := \begin{cases} c'(e) & \text{if } e \in E(Y); \\ c_G(e) & \text{otherwise.} \end{cases}$$

is a 3-decomposition of $G[X \to Y]$, i.e., it satisfies the conditions in Proposition 6. The first condition is satisfied due to Condition 1 of Corollary 7 and Condition 1 of Theorem 13. The second condition is satisfied because $P'_c \in \mathrm{Comp}(P)$. ◀

An Efficient and Uniform CSP Solution Generator Generator

Ghiles Ziat
EPITA Reasearch Lab (LRE), Le Kremlin-Bicêtre, France

Martin Pépin
Université Caen Normandie, ENSICAEN, CNRS, Normandie Univ, GREYC UMR 6072, F-14000 Caen, France

Abstract

Constraint-based random testing is a powerful technique which aims at generating random test cases to verify functional properties of a program. Its objective is to determine whether a function satisfies a given property for every possible input. This approach requires firstly defining the property to satisfy, then secondly to provide a "generator of inputs" able to feed the program with the inputs generated. Besides, function inputs often need to satisfy certain constraints to ensure the function operates correctly, which makes the crafting of such a generator a hard task. In this paper, we are interested in the problem of manufacturing a uniform and efficient generator for the solutions of a CSP. In order to do that, we propose a specialized solving method that produces a well-suited representation for random sampling. Our solving method employs a dedicated propagation scheme based on the hypergraph representation of a CSP, and a custom split heuristic called *birdge-first* that emphasizes the interests of our propagation scheme. The generators we build are general enough to handle a wide range of use-cases. They are moreover uniform by construction, iterative and self-improving. We present a prototype built upon the *AbSolute* constraint solving library and demonstrate its performances on several realistic examples.

2012 ACM Subject Classification Computing methodologies → Randomized search

Keywords and phrases Constraint Programming, Property-based Testing

Digital Object Identifier 10.4230/LIPIcs.CP.2025.40

1 Introduction

The objective of this work is to propose a technique for building an efficient and uniform sampler of solutions from a Constraint Satisfaction Problem (CSP).

Generating uniformly distributed solutions to a Constraint Satisfaction Problem can be useful in various applications where fairness, diversity, unbiased testing, or comprehensive exploration of solution spaces is required. For resource allocation and fair division, applications like cloud computing and fair task distribution rely on uniform sampling to guarantee equity [27]. In machine learning, uniform solution generation helps creating diverse training datasets for constraint-based domains [1]. Cryptography benefits from uniform sampling for unpredictable key generation [6]. Also, applications like network testing leverage uniform sampling for unbiased evaluation of routing algorithms [12].

Also, crafting a uniform sampler is particularly useful in the context of Property Based Testing [3] (PBT), and more specifically in random testing in which they are refered to as *generators*. Random testing is a black-box testing technique where programs are tested by generating random, independent inputs, provided by a generator. Results of the output are compared against software specifications to verify that the test passes or fails. While generarating inputs uniformly at random can be straightforward, it is often necessary to generate inputs that satisfy certain number of prerequisites, while assuring a good coverage of the input space and maintaining reasonable execution time. Frameworks *à la* QuickCheck usually deal with this problem by providing *rejection sampler* combinators: a candidate input

© Ghiles Ziat and Martin Pépin;
licensed under Creative Commons License CC-BY 4.0

is generated using a base generator and kept if it satisfies some constraints, or discarded otherwise. In the latter case, the test is not executed and the process is generally repeated a fixed number of time until a candidate is found or giving up. This approach is simple, both to use and to implement, and is uniform – given that the base generator is uniform itself. However, it might not be efficient in scenarios where the solution space is sparse or where constraints are complex. Constraint solving provides more powerful mechanisms to travel the solution space, making it a valuable improvement over repeated rejection sampling. Even though this improvement comes with the cost of importing the heavy machinery of a constraint solver, several works have shown that it makes the overall process more time and resource-efficient when the solution space is sparse[22, 26, 29, 2].

QuickCheck's rejection sampling provides a straightforward way to create a generator, albeit one that may be very slow. In contrast, constraint-solving techniques adopt a more computationally intensive process to construct a fast generator. Our proposed incremental approach seeks to strike a balance between these, aiming to achieve an optimal trade-off between the speed of generator construction and runtime efficiency. This balance is particularly beneficial in scenarios where there is little or no prior information about how many times the generator will be used. In such cases, the upfront cost of constructing a highly optimized generator through constraint solving may be unjustified if the generator is used only a handful of times. Conversely, relying solely on a quickly constructed, but slow generator, may result in suboptimal performance if the generator is reused extensively. By incrementally refining the generator as needed, our approach adapts to different usage patterns, providing a flexible and efficient solution regardless of the frequency of generator usage. This scalability is achieved through our novel selection structure, which is designed to handle more complex forms, such as non-fixed-size structures (e.g., list matrices), a challenge that many existing approaches fail to address. Instead of immediately generating examples, our method abstracts away the generation process, allowing for dynamic adjustment and refinement of the generator based on constraints. It thus incrementally *generates* a generator that encapsulates the desired properties and constraints, hence the title of this paper.

1.1 Contributions of the paper

- A general method to produce fast and self-improving uniform samplers of CSP solutions using a dedicated solving algorithm for random generation.
- A graph based split heuristic and propagation scheme, well fit to partition a problem into independent sub-problems.
- A heuristic Huffman-like representation for the solutions which minimizes the cost of uniform choices.
- A property based testing API, *à la quickcheck*, allowing for a transparent usage of our hybrid approach between rejection sampling and constraint based solving. The implementation is available at:
  `https://osf.io/u4r5q/files/osfstorage?view_only=84af8ee65a6c495d98cb7b7bfad8c54b`

1.2 Outline

This paper is organized as follows: Section 2 defines the concept of generators in general and gives some insight of what is expected from a *good* generator. Section 3 introduces the main mechanisms needed to build a constraint-based generators. Section 4 is the main contribution of this work: it addresses the problem of building a *propagation-exploration*

scheme well-suited for the design of a constraint based generator. Section 6 presents our implementation and show its performances on some benchmark. Finally, Section 7 presents the related works and Section 8 summarizes our work and discusses its future continuations.

2 What is a (Good) Generator?

CSP's search spaces are defined by variables to which are associated finite bound domains. A point in this search space in generally called an instance, but for our purpose we will call those samples.

▶ **Definition 1** (Sample). *Given a set of variables $\mathcal{V}$, and a set of domains $\mathcal{D}$ denoting the possible values of variables, a sample is a total mapping from variables to values. We note the set of samples $\mathcal{S} = \mathcal{V} \to \mathcal{D}$*

We distinguish two kinds of samples, the ones that satisfies all the constraint of a given CSP *i.e.* the solutions, and the ones that violate at least one constraint.

▶ **Definition 2** (Solution Generator). *Given a CSP $(\mathcal{V}, \mathcal{D}, \mathcal{C})$, a solution generator is a function g that takes a random state and produces a solution $s \in S$, such that:*

$$\forall r, \forall c \in \mathcal{C}, \quad c(g(r)) \text{ holds}$$

where the first quantification is over all the possible random states r.

This definition ensures that every generated sample satisfies all constraints of the CSP. Moreover, the use of random states[1] is needed so that the generation process can be made reproducible by controlling or restoring the state. Good generators should be:

- **Correct:** They should respect the constraints they are subject to. For a data type representing positive integers, a generator should ensure that it only produces positive integers.
- **Uniform:** They should be able to thoroughly explore the input space. In other lines of work, diverseness is ensured by building surjective generators (*i.e.* every possible solution can be generated with *a non-zero probability* [9]). Here we have a stronger requirement: every possible solution must be generated with *the same probability*[2].
- **Efficient:** Testing time includes generating time, since test cases are generated dynamically. Hence, to be of practical use, generators must maintain reasonable performance, in particular in large codebases that run tests frequently.

Property-based testing frameworks *à la* QuickCheck generally resort to rejection sampling to produce random values that meet specific constraints. While this technique meets the first two of the above requirements, it can lead to a significant overhead, in particular when a large portion of generated values are rejected. The impact on performances depends on the efficiency of the rejection process and the likelihood of rejection. We briefly recall how it works.

[1] We do not specify the actual representation of random states, as it is irrelevant to our discussion and depends on the underlying implementation of the generator.

[2] Some framework focus on building corner-case generators, generally defined in an *ad-hoc* fashion, in which case uniformity is irrelevant. This is a complementary approach that we are not focused on in this work.

2.1 Rejection Sampling

In the most general sense, rejection sampling is the approach that consists in generating samples in a super-set of the set of objects we are interested in, repeatedly until a sample lying in the set of interest is found. For instance, in order to generate a point inside the disk of radius 1 in $\mathbb{R}^2$, one could generate points (X, Y) in the square $[-1, 1]^2$ until $X^2 + Y^2 \leq 1$. In a constraint solving context, this means generating samples within the bounds of the problem until a sample that satisfy the constraints is found. Algorithm 1 illustrates this method.

Algorithm 1 Rejection sampling procedure.

```
1: function SAMPLE(V,D,C)
2:     candidate ← spawn (V,D)          ▷ draw at random in (V, D)
3:     if sat(candidate, C) then
4:         return candidate
5:     else
6:         return sample(V,D,C)
```

This algorithm takes as input a set of variables $\mathcal{V}$, their associated range of values $\mathcal{D}$, and a set of constraints $\mathcal{C}$. The `spawn` function generates a candidate at random (for some probability distribution $\mathbb{P}$) from the given variables and their domains. Then, the `sat` function evaluates whether the generated candidate satisfies all the constraints in $\mathcal{C}$. The procedure is called recursively until a solution is found.

Note that the number of iteration of this algorithm is a random variable. The probability of accepting a sample X at line 3 is the probability $P(X \in A)$ that X lies inside the solution space A. Provided that this probability is non-zero, this algorithms terminates, and its number of iterations follows a geometric law of parameter $\mathbb{P}(A)$. It thus makes $\frac{1}{\mathbb{P}(X \in A)}$ iterations in expectation.

Finally, a key observation is that Algorithm 1 implements the probability distribution $\mathbb{P}$ *conditioned* to only draw elements of the solution space A. In particular, if the `spawn` function draws uniform samples in the domain $\mathcal{D}$, then Algorithm 1 is a uniform sampler of solution. The contribution of the present paper is to devise a good `spawn` function that can guarantee that this algorithm is uniform while providing good performance.

3 Constraint-Based Generators

In this article, we use a constraint-based approach for random sampling. To achieve this, we rely on the general abstract solving method described in [21], which we summarize here.

Algorithm 2 constructs a cover of the solution space using abstract elements (e.g., boxes, octagons, polyhedra, etc.). This cover consists of two sets: inner elements (I) and outer elements (O). The set I under-approximates the solution set, while $I \cup O$ over-approximates it. The algorithm starts by initializing an abstract element and inserting it into O. It then iterates through the following steps: an element from O is selected, filtered, and, if it satisfies the constraints, added to I. If it does not, it is split into sub-elements that are reinserted into O.

As presented, this algorithm may not terminate. In practice, various termination criteria can be employed, such as limiting the size of the elements considered or the depth of the solving tree. Note that, in general, split elements can overlap without preventing the algorithm from terminating and producing correct results. However, in our case, ensuring they are non-overlapping is necessary for uniformity, as we will discuss later.

Algorithm 2 Abstract solving method.

```
1: function SOLVE(𝒳, 𝒟, 𝒞)          ▷ 𝒳:variables, 𝒟: domains, 𝒞: constraints
2:     I ← ∅                          ▷ inner solutions
3:     O ← {init(𝒟)}                  ▷ outer solutions
4:     while O ≠ ∅ do
5:         e ← select(O)
6:         e′ ← filter(e, 𝒞)
7:         if e′ ≠ ⊥ then
8:             if solution(e′, 𝒞) then
9:                 I ← I ∪ {e′}
10:            else
11:                O ← O ∪ split(e)
12: return I, O
```

Although the algorithm is parametric with respect to the representation being used, we only use boxes in our implementation, as they enable straightforward and efficient uniform random sampling. Recall that given variables $v_1, \ldots, v_n$ over finite continuous domains $d_1, \ldots, d_n$, a box is defined as a Cartesian product of intervals within $d_1 \times \cdots \times d_n$. A random sample of a box is thus the Cartesian product of random samples of such intervals.

3.1 Constrained Based Sampling

In [29], authors build upon this constraint-solving method an algorithm for uniform sampling under constraints, as shown in Algorithm 3. It repeatedly selects an element e from either the inner or outer sets using a `select` function. Here, `select` performs a weighted choice based on the volume of the elements, with the largest elements having the best chance of being chosen. We discuss in the next section the design of a data structure that enables an efficient implementation of this function. The algorithm generates a candidate value i within e. If e belongs to the inner set or i satisfies the constraints $\mathcal{C}$, the algorithm returns i; it repeats these steps otherwise.

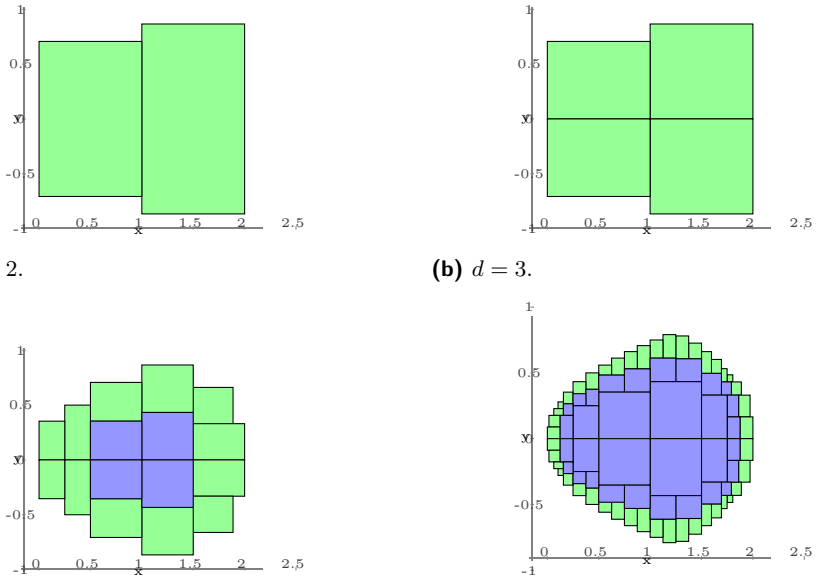
Algorithm 3 Random Sampling for Covers.

```
1: function GENERATE(inner,outer,𝒞)
2:     while true do
3:         e ← select(inner,outer)
4:         i ← spawn(e)
5:         if e ∈ inner ∨ sat(i,𝒞) then return i
```

This algorithm ensures uniform sampling under three conditions: the elements in $I \cup O$ must not overlap, points within each element e must be sampled uniformly $P(i \mid e) = \frac{1}{v(e)}$, and the probability of selecting an element e must be proportional to its volume ($P(e) = \frac{v(e)}{\sum_{e' \in I \cup O} v(e')}$). These conditions ensure that the sampling process remains uniform across the domain, giving all points in the union of elements an equal likelihood of being chosen.

▷ Claim 3. Algorithm 3 samples solutions uniformly. While uniformity was an implicit goal in [29], we provide a formal proof to establish it rigorously.

Proof. In appendix A. ◁



(a) $d = 2$. **(b)** $d = 3$.

(c) $d = 5$. **(d)** $d = 8$.

Figure 1 Solving state obtained for depth d of resolution for a problem with two variables constrained by $x^2 + 4y^2 - 4 \leq 0$ and $2y^2 - x \leq 0$. Darker boxes indicate inner elements.

Refining the solving process can potentially reduce the rejection rate – though not always – but it never increases it. However, this refinement inevitably increases the selection time. At some point, this may even become counterproductive; when the increase in selection time exceeds the gains obtained from reducing the rejection rate, further refinement is no longer beneficial. Our goal is to find the best trade-off between constraint solving and rejection sampling, minimizing unnecessary exploration while ensuring efficiency.

3.2 Efficient selection of an abstract element

In order to implement the `select` function, we need a data structure for storing a collection of abstract elements that supports:

- efficient sampling of an elements with probability proportional to its volume;
- and (we will see later), an efficient way to replace an abstract element with a collection of smaller elements.

In [29], sorted list in decreasing order of volume are used for element selection, so that the elements with highest probability are met faster during sampling. However, if elements are of the same size, this approach offers no advantage, as all selections become equally likely. Also, they do not require to update their structure in their work. We can do better by arranging these elements in a binary tree. The idea is to store the abstract elements at the leaves of the tree and to maintain, in every node, the sum of the volumes of all the leaves below that node. Using such a data structure, drawing a abstract element proportional to its volume corresponds to drawing a uniform real variable between 0 and the total volume of the tree, and to recursively descend in the tree, choosing between the left and right child based on their weights. This is illustrated in Algorithm 4.

Here, the cost of selecting an element is proportional to the length of the path from the root to the selected leaf. Given the volume of every abstract element (and thus their probability of being drawn), there is an optimal way to arrange these elements in the tree

Algorithm 4 Random generation of an abstract element using the tree data structure.

```
function randomTreeSelect(T)
    r ← Unif(0, volume(T))
    return treeSelect(r, T)
```

Require: $0 \leq r < \text{volume}(T)$

```
function treeSelect(r,T)
    if T is a leaf then return the abstract element stored in T
    (T_L, T_R) ← the children of T
    if r < volume(T_L) then return treeSelect(r, T_L)
    else return treeSelect(r − volume(T_L), T_R)
```

in order to minimise the expected cost of the generation. Information theoretic results tell us that the expected path length between the root and the sample is lower bounded by the entropy of the probability distribution, that is

$$\mathcal{H} = \sum_x p_x \ln \frac{1}{p_x}$$

where the sum ranges over the abstract elements stored in the tree, and p_x is the sampling probability of element x. The optimal way to organise the tree in order to remain close to this lower bound is to use a *Huffman tree* [16]:

- start from a collection of leaves,
- iteratively pair the two smallest elements of the collection as a binary node,
- this process terminates when every leaves belong to the same tree.

A tree built this way has the property that its expected path length to a leaf is at most $\mathcal{H}+1$.

Unfortunately, the good properties of Huffman trees are hard to maintain efficiently when we update the collection of abstract elements, which we need for in the adaptive algorithm presented in Section 3.3. To circumvent this issue, we use the following heuristic:

1. before doing any sampling, we do a first solving pass until a certain depth;
2. after this pre-processing, we construct a Huffman tree T based on the volumes of the resulting elements;
3. then, during the iterative sampling process, every time we need to split an abstract element e into a collection of smaller elements $(e_1, e_2, \ldots, e_p)$, we construct a Huffman tree T' for $(e_1, e_2, \ldots, e_p)$ and we replace the leaf e in T with T'.

It is worth noting that the replacement of the last sampled leaf can be optimised by keeping a pointer to this last leaf, rather than traversing the tree a second time to find it.

The initial Huffman tree T constructed at step 2 thus potentially evolves away from its optimal shape as we update leaves. However, since the elements that have the highest probability to be selected (and thus split) are the bigger ones, we expected our heuristic to maintain some balance in the tree. Our benchmarks, presented in Section 6, seem to show that this data structure performs well in practice. An interesting algorithmic problem would be to investigate the real performance of this idea, and potentially find a better data structure, which is a work in progress.

3.3 Incremental and Adaptive Generators

From a practical point of view, we cannot really replace an inefficient sampler with a generator based on an extremely expensive constraint solving mechanism. This would simply replace a slow generation speed, by a faster one preceded by a huge overhead due to solving time. This is especially prejudicial when the generator is only used a handful of time.

Iterative solvers are commonly used when dealing with complex constraint satisfaction problems where finding an exact solution is computationally infeasible or impractical. Instead of attempting to solve the entire problem in one step, iterative solvers work incrementally, refining the solution repeatedly until a certain termination criterion is met.

We therefore propose to use constraint resolution mechanisms parsimoniously: first, we will target in priority on certain parts of the problem, identified on the fly, whose filtering can greatly improve the performance of random generation. Second, we try to amortize the cost of the resolution steps (filtering and exploration) during the generation. For example, every n time a generator is called, we can perform a resolution step. Thus, the more a generator is used, the more it improves. To achieve this, the generator embeds an internal solving state that evolves with each step. This state retains information from previous resolution steps, allowing the generator to improve progressively rather than starting from scratch each time.

Algorithm 2 revolves around two primary decisions: selecting which element to refine and determining how to refine it. On the one hand, intuitively, the focus should be on larger elements with higher rejection rates, as they are more critical to the rejection rate and, thus, the efficiency of the whole process. On the other hand, sampling also requires choosing an element, with larger elements being more likely to be selected. This leads to the question of whether sampling's focus on larger elements can be used to help guide the solving process.

For each element in our cover, we track the number of times it was selected and record the number of times it successfully produced a sample. We also track the total number of successes and failures for the whole cover. When our algorithm fails to produce a valid solution within a selected element, we must then decide whether it is necessary to split it or not. To do this, we base our choice on its acceptance rate. If it falls below the global acceptance rate, we proceed to refine that element, meaning we split it and replace it within the cover with the resulting sub-elements. Otherwise, it is left unchanged. This dynamic adjusting of the refinement process can lead to a faster convergence and avoids over-splitting, which can deteriorate the element selection procedure. This is illustrated by Algorithm 5.

Algorithm 5 Sampling-Guided solving.

```
1: function GENERATE(inner,outer,C)
2:     while true do
3:         e ← select(inner,outer)
4:         i ← spawn(e)
5:         if e ∈ inner ∨ sat(i,C) then
6:             return i
7:         else if rate(e) < global(inner,outer) then
8:             refine e
```

4 Graph Representation for Random Sampling

The constraint hypergraph of a constraint satisfaction problem is a hypergraph in which the vertices correspond to the variables, and the hyperedges correspond to the constraints. Hypergraph representations can be integrated with various constraint-solving algorithms, such as backtracking, constraint propagation, and local search. The hypergraph structure guides the search for a solution. For our needs, an interesting idea is the detection of connected components. Indeed, the partition of the constraint hypergraph into connected components correspond to a partition of the problem into statistically independent sub problems. This means that the sampling for each component can be decorrelated from the

sampling of the others. In other words, if the constraint graph consists of two (or more) connected components, it is then possible to generate solutions independently in the different components, and then combine the results. Intuitively, leveraging this independence should speed up the random generation process. We also give a heuristic argument.

The acceptance rate of a set of constraints $\mathcal{C}$ is the probability that every constraint in $\mathcal{C}$ accepts a random sample. We note denote by $P_{\mathcal{C}}$ this probability. The expected number of rejections before a sample satisfies all the constraints in $\mathcal{C}$ is thus $\frac{1}{P_{\mathcal{C}}}$. When there are two (or more) disjoint components in the graph, the set $\mathcal{C}$ can be split into two independent subsets, $\mathcal{C}_1$ and $\mathcal{C}_2$, with respective supports $\mathcal{V}_1$ and $\mathcal{V}_2$ and we have $P_{\mathcal{C}} = P_{\mathcal{C}_1} P_{\mathcal{C}_2}$. Leveraging the independence of the variables in $\mathcal{V}_1$ and $\mathcal{V}_2$ by performing the rejection sampling independently for the two components, yields a expected number of rejections of the order of

$$\frac{1}{P_{\mathcal{C}_1}} + \frac{1}{P_{\mathcal{C}_2}} \qquad \text{instead of} \qquad \frac{1}{P_{\mathcal{C}_1} P_{\mathcal{C}_2}}$$

which is largely smaller when the constraints $\mathcal{C}_i$ are hard to satisfy, *i.e.* when the $P_{\mathcal{C}_i}$ are small. Of course, a more precise analysis would require to take into account the cost of sampling a leaf in the Huffman tree and the cost of generating a uniform sample inside an abstract element. However, we can already foresee that splitting the Huffman tree into the two trees for the two connected components, and reducing the number of variables will only moderately affect the performance in the algorithm. We can conclude with confidence that leveraging the independence of the components, will be beneficial.

In practice, at initialization, the CSP is divided into (disjoint) connected components, and a sampling cache is associated with each one. This cache stores the partial samples generated for each component, allowing the algorithm to reuse previously computed samples and avoid redundant computations.

We benefit from this idea by using the cache to identify and prioritize the solver's efforts on components that exhibit high rejection rates. Empty caches indicate difficulty in finding valid samples, encouraging the solver to select a variable to split in these components. By adapting to the performance of individual components, the solver converges toward a locally optimal configuration.

4.1 The *bridge-first* split

The iterative process of splitting and filtering gradually ensures that certain constraints are locally satisfied, allowing them to be removed from the graph. Whenever a constraint is removed, the solver checks whether its removal disconnects the graph. When this is the case, the graph is split into connected components, enabling their independent handling.

Constraints become redundant when their validity is guaranteed by current domain assignments. Our implementation recomputes connected components whenever a constraint is removed by the solver so as to be able to exploit the independence of the components.

To achieve this, we develop a propagation scheme and an exploration heuristic that help reduce the connectivity of the constraint graph. For each edge, we maintain information on whether it is a bridge or not. A bridge is an edge whose removal increases the number of connected components in the graph. We identify bridges using Tarjan's algorithm [25]. Tracking which edges are bridges is useful for guiding exploration, since variables that are part of a bridge are particularly interesting for splitting. When a constraint is removed, if it was a bridge, the connected components (CCs) are recomputed. Otherwise, the removal is checked to determine whether it introduced new bridges, without recalculating the connected

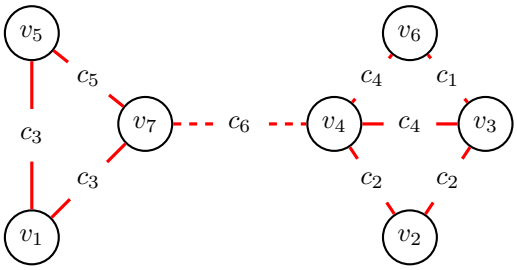


Figure 2 Deletion of the constraint c_6 leads to two independent CSPs.

components. If a bridge is detected, we attempt to eliminate it, as its removal would disconnect the graph and thus improve the sampler's performance. This is done by prioritizing the variables in the bridge's support during splitting steps.

5 Constraint Based Testing with GeGen

We have implemented the techniques presented in this paper in a prototype called *GeGen*, which stands for ***Ge****nerator* ***Gen****erator*. PBT frameworks like QCheck [4], which is the OCaml port of QuickCheck, provide generator for atomic types (bool, ints, floats, ...) and combinator for composite type (pairs, tuples, ...). In presence of recursive types, the traditional approach is to provide the user with a generator of *sized values*, that is user has to provide a size, and the generator builds values with that specific size. Most implementations lack robust support for testing scenarios that require constraints over the generated inputs. *GeGen* bridges this gap by introducing variable generators and constraints, enabling the easy integration of constraint-solving capabilities in a PBT framework. For instance, QCheck provides the function find_example whose signature is given below. Given a generator of a values of type t, it builds a generator of values of t that satisfy a certain predicate

```
find_example : ('a -> bool) -> 'a Gen.t -> 'a Gen.t
```

If a value satisfying the predicate f is found (within a certain number of tries), it is returned. *GeGen* mimics this approach by providing an API that users can manipulate similarly, however the inner mechanism differ notably as we build, solve and sample from a CSP during the generation process.

5.1 GeGen's Generators

GeGen's generators differ fundamentally from traditional QuickCheck-style generators as they produce *symbolic variables* rather than concrete values. Traditional property-based testing frameworks generate fixed values like random integers or floats. In contrast, GeGen's generators operate at a symbolic level which makes possible constraint composition and solving. For instance, instead of directly generating a number, *GeGen* creates a symbolic variable that represents the number and associates it with a range of possible values. This symbolic variable acts as a declaration on the solver's side. Additionally, instead of applying a predicate to verify wether a value satisfies or not the property being tested, GeGen's predicates impose constraints on this variable such as "*the number must be even*" or "*the number must be less than another variable*" that are collected to construct a CSP. Finally, we apply the previously discussed solving and sampling techniques to the constructed CSP. For example, to generate circles within a given square in QCheck, we can use the following code:

```
find_example
  (pair (pair int int) int)
  ~f:(fun ((x, y), r) ->
    r > 0 && x >= r && x <= 100 - r && y >= r && y <= 100 - r && r >= 10)
```

The find_example function generates random integer values for x, y, and r, then applies the given predicate to check whether the generated circle satisfies the constraints. This process repeats until a valid example is found or the search limit is exhausted. Note that here, int is an atomic generator that produces random integers and pair is a combinator that constructs a generator of pairs from two base generators.

The interest of our approach is that the same code, when linked against our library, will yield identical results but in a significantly more efficient manner, and using a fundamentally different mechanism. Instead of directly sampling random values for x, y, and r, we constructs a symbolic representation of the problem. From the solver's perspective, a constraint satisfaction problem (CSP) with three floating-point variables v_1, v_2 and v_3 is formulated. However, this approach abstracts certain details about the algebraic structure of the type. Therefore, a reconstruction function is designed alongside the CSP, so that once a sample s is drawn, it can be re-assembled to a value of the correct type. In this particular case, the function is $f(s) = ((s(v_1), s(v_2)), s(v_3))$. It ensures that once the solver produces a valid assignment for the symbolic variables, the corresponding concrete values are reconstructed in a way that respects the intended algebraic structure. Finally the predicate, instead of returning a truth value will actually build the equivalent constraint system. In essence, this allows us to decouple the problem-solving phase from the data generation phase (at the API level) while ensuring that the generated values maintain the correct type.

5.2 GeGen's language

The constraint language used in *GeGen* supports arithmetic and boolean expressions. It includes: arithmetic operation (addition, multiplication, etc.), Boolean logic (comparisons, conjunctions, disjunctions, negations), and variables. We build upon those generators for more complex types such as tuples and lists by composing the different elements of a generator.

GeGen extends standard arithmetic and Boolean operators by overriding them to facilitate the construction of constraints, making it possible to define a CSP in a manner that closely resembles programming its logical predicate counterpart. This intuitive approach aligns the construction of constraints with familiar programming paradigms, simplifying the transition from predicates to constraints.

For instance, consider a predicate that verifies whether a list is sorted. This predicate can naturally be expressed in a functional style as follows:

```
let rec is_sorted = function
| [] | [x] -> true
| x1 :: x2 :: rest -> x1 <= x2 && is_sorted (x2 :: rest)
```

This predicate can be applied to concrete lists of integers to determine whether they are sorted. This is what is done when doing rejection sampling. Using GeGen, the predicate takes another meaning as it operates on lists of variables to dynamically construct a CSP. The overloaded operators automatically translate the predicate logic into corresponding constraints. For example:

```
open GeGen
let rec is_sorted = function
| [] | [x] -> true_
| x1 :: x2 :: rest -> x1 <= x2 && is_sorted (x2 :: rest)
```

Here, the open **GeGen** directive brings in scope the operators <= and && (among others), so that instead of computing a boolean value, the is_sorted function now builds a constraint. Beside that the only difference is the use of the symbolic constraint true_ instead of the builtin OCaml boolean true, which can't be avoided as OCaml does not permit the overriding of litterals. This approach preserves the logical structure and readability of the predicate while generating a constraint representation that can be used to solve CSPs.

6 Experiments

Our implementation is written in OCaml in a functional style. We use the default *pseudo-random number generator* (PRNG) from the OCaml standard library, an instance of the LXM [24] family of PRNGs. *GeGen* relies on the *AbSolute* solver [21] to handle constraints. This solver provides most of the necessary functionalities for uniform random sampling, including volume measurement and efficient space paving with large elements [30]. It guarantees a non-overlapping solution cover, simplifying uniform distribution construction. Our implementation is open-source and available at:
`https://osf.io/u4r5q/files/osfstorage?view_only=84af8ee65a6c495d98cb7b7bfad8c54b`.

6.1 Benchmark

We have written generators using our framework for several realistic examples: the `convex` problem defines the predicate for star-convex polygons, *i.e.* the vectors corresponding to two consecutive edges of the polygons have a negative cross-product. The `diagdom`, `idempotent`, `symmetric` and `orthogonal` problems, define the predicates for some well known classes of matrices. The `distrib`, `itvcover` and `sorted` represent respectively the predicate for valid distribution of probability *i.e.*, list of positive values that sum to one, interval-covering list that is a list that contains a set of intervals that cover a specific range without gaps and finally the set of sorted lists in increasing order. We compare our approach against QCheck's standard rejection sampler, as it serves as a baseline for property-based testing. This comparison is relevant since we have designed an API that allows users to write the same specifications while seamlessly generating samples using our method instead.

Table 1 presents the number of samples generated by *GeGen* and QCheck for various problem instances. Each row corresponds to a specific problem. The first column lists the problem names. The second column indicates the problem size parameters, denoted as $|\mathcal{V}|, |\mathcal{C}|$. The remaining columns report the number of generated samples for different time constraints (0.1, 1, and 2 seconds of generation time), with results shown separately for *GeGen* and QCheck. Best result for each are shown in bold font. The problem sizes were selected to ensure that trends in sample generation could be clearly observed while maintaining a fair evaluation of both methods. The measurements were conducted on a machine with a 12th Gen Intel(R) Core(TM) i5-1235U processor and 15 GiB of RAM.

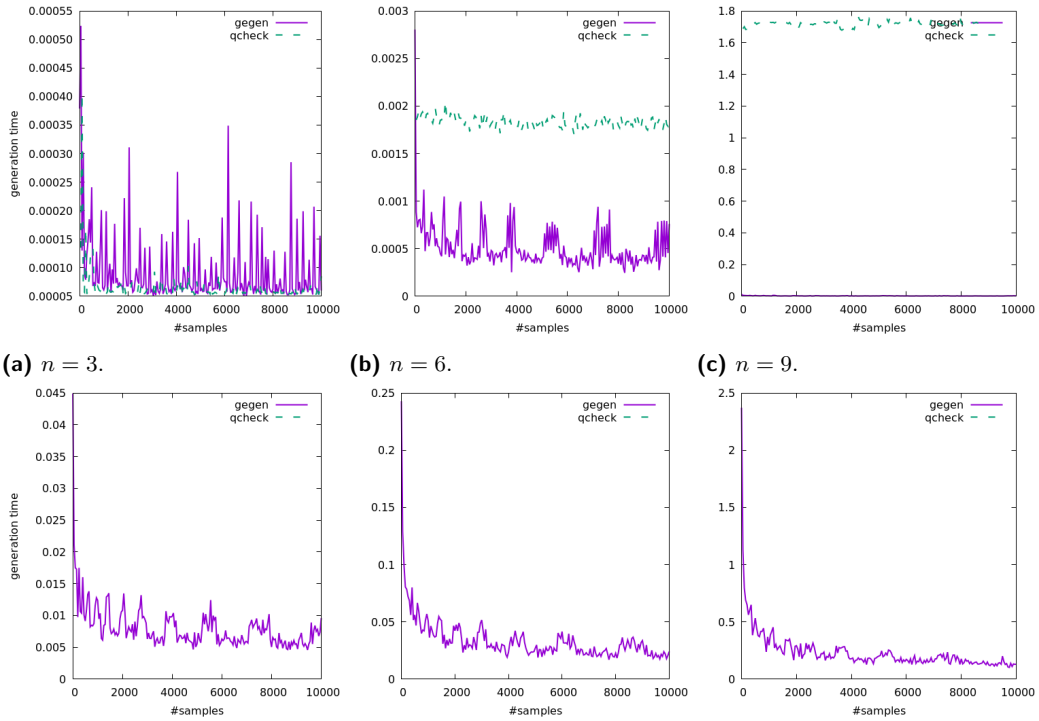
6.2 Results Analysis

Table 1 Number of samples per generation time

		GeGen			QCheck		
problem	$\lvert\mathcal{V}\rvert, \lvert\mathcal{C}\rvert$	0.1	1	2	0.1	1	2
sortedlist4	4,3	7076	292589	621378	**17999**	**527764**	**1107631**
sortedlist8	8,7	**410**	**32985**	**76783**	9	1047	2191
sortedlist12	12,11	**28**	**4891**	**12421**	0	0	0
convex3	6,3	**752**	**54829**	**114298**	5	415	868
convex4	8,4	**203**	**9182**	**19997**	0	0	2
convex5	10,5	**82**	**3526**	**7357**	0	0	0
diagdom3	9,4	823	**39282**	**83120**	**1052**	25354	52677
diagdom4	16,5	**141**	**6454**	**13680**	0	25	54
diagdom5	25,6	**20**	**984**	**2176**	0	0	0
itvcover3	6,5	**5522**	**225252**	**477677**	0	29	62
itvcover4	8,4	**4258**	**132492**	**279720**	0	1	3
itvcover5	10,9	**2724**	**87048**	**174027**	0	0	0
symmetric2	4,1	5617	**428439**	**901930**	**7842**	242921	493929
symmetric3	9,3	**7178**	**233311**	**490004**	0	1	3
symmetric4	16,6	**2176**	**136995**	**280146**	0	0	0
orthogonal2	4,5	**1457**	**59079**	**125073**	0	0	0
orthogonal3	9,8	**132**	**5110**	**10693**	0	0	0
orthogonal4	16,12	**3**	**596**	**1329**	0	0	0
idempotent2	4,10	**1494**	**93988**	**205829**	0	15	33
idempotent3	9,29	**35**	**3590**	**8295**	0	0	0
idempotent4	16,76	0	**41**	**158**	0	0	0

The results indicate that *GeGen* consistently outperforms QCheck, generating significantly more samples across nearly all problem types and sizes. While QCheck performs adequately for smaller problem instances, its performance drops as problem complexity increases, often failing to generate any samples for larger cases. In contrast, *GeGen* scales effectively, maintaining high sample generation rates even for more complex problems. Also, the performance of the rejection sampler remains stable over time while the ones of the generator built using *GeGen* improves as the process progresses. This trend is illustrated in Figure 3, which depicts the time in seconds (y-axis) required to generate the i-th sample (x-axis), for 10000 samples. *GeGen*'s times (solid line), and QCheck's (dashed line) are shown on a single problem, sorted lists of size n, to highlight this feature of our generators, although the same behaviour can be noticed on problems of table 1. Notable spikes in the curves correspond to garbage collection cycles, which are particularly apparent for small values of n.

Figure 3a demonstrates that for small list sizes, QCheck and *GeGen* exhibit similar performance. In fact, QCheck marginally outperforms *GeGen*, as the rate of sorted lists among lists of size 3 is relatively high. However, as n increases, the rejection rate also rises, causing *GeGen* to surpass QCheck. By the time n=9n=9, *GeGen* is already outperforming QCheck by three orders of magnitude, as shown in Figure 3c. Note that, due to the scale, the *GeGen* line appears very close to the x-axis. For larger values of n, rejection sampling via QCheck becomes impractical, and thus only the results for *GeGen* are presented. This figure shows that the incremental nature of our generators allows them to scale effectively, handling larger n more efficiently.



(a) $n = 3$. **(b)** $n = 6$. **(c)** $n = 9$.

(d) $n = 12$. **(e)** $n = 15$. **(f)** $n = 20$.

Figure 3 Evolution of the generation time (in seconds) for sorted list of size n, with both QCheck and *GeGen*.

Uniformity Validation. We have validated the uniformity of our generators by comparing their distributions with those generated by a rejection sampler. The variation between the distributions is measured using a chi-squared test, which quantifies how far apart the distributions are. Our results confirm that our generators produce distributions similar to the rejection sampler, indicating they are uniform.

7 Related Works

Random sampling of solutions of constrained systems is a well studied subject and the literature is full of techniques for SAT [28, 23, 10] models, and some results for CP models exists, but mainly for finite domains CSPs [7, 11, 13, 26, 15, 22].

Closer to our work are [2, 17, 29, 5]. In [2], the author present a technique for automatically deriving test data generators from a predicate expressed as a Boolean function. In order to speed up random generation they use the lazy behaviour of the predicate to know its result on sets of values, rather than individual values. Once they have computed a set of values for which the predicate is going to return false, they remove all of these values from the original set. They implement this by relying on Haskell's call-by-need semantics which applies the predicate to a partially-defined value. This can be seen as an *ad-hoc* use of consistency on a partial assignment. Our work goes further by not only integrating a constraint resolution engine into the random generator, but in addition, this engine is dedicated to the manufacturing of relevant representations for random sampling.

In [29], the authors propose a constraint-based generation framework where constraint solving, which can be costly, is handled at compilation time via preprocessing, while uniform sampling occurs at runtime. While their approach is similar to ours, we find ours more practical as it is incremental and does not require preprocessing. Additionally, they do not focus on tuning the solving method for uniform sampling, instead fixing the solver's depth and size in advance, whereas we adapt constraint solving based on the sampler's output. More importantly, their approach cannot handle recursive types, a limitation our method overcomes effectively. In [5], a Constraint Logic Programming approach for Property based testing for Erlang programs, the authors generate random tests for complex properties involving Modified Condition/Decision Coverage, pattern matching, and higher-order functions. This is equally useful but orthogonal to our wors as their focus is put on code coverage and not of uniformity. A similar idea, in the field of computational statistics, is Adaptive Rejection Sampling [19, 18] (ARS). The idea is to use a piecewise linear density function instead of a single uniform envelope density function. Each time a sample has to be rejected, the rejected value can be used to improve the piecewise approximation of the targeted distribution. This therefore reduces the chance that the next attempt will be rejected. This can however only be applied for sampling from for specific families of densities.

Further from our field, other random sampling frameworks include the so-called recursive method from Nijenhuis and Wilf [20] and the framework of Boltzmann sampling [8]. These frameworks provide a generic and efficient way to sample structured data such as trees, algebraic data types, etc. but are not suitable for numerical data. Finally, Monte-Carlo Markov Chains (MCMC) are another well-known tool for sampling discrete structures [14]. However, here again, tuning a Markov Chain to make it efficient requires some specific knowledge on the objects to sample, which make it unsuitable for generically sampling into an arbitrary CSP solution space.

8 Conclusion

In this work, we have developped a method for building generators that are unifom, efficient, and incremental. Our approach can be summarized as the gradual transition from rejection sampling to a form of constraint-based one. Users have a generator that adapts to their needs: If the generator is rarely used, there is no need to spend time solving complex problems, and the underlying search space remains largely unprocessed by the solver. As a result, the generator behaves more like a rejection sampler. Conversely, if the generator is heavily used, the solver will more aggressively filter the search space, aiming to amortize the resolution cost through improved generation speed. The techniques we propose holds for both finite and continuous domains as the number of solutions in a given search-space, whether it is finite or not, is abstracted by the notion of weight. Also, our method for splitting the CSP into independant pieces, which greatly improves our samplers, can be reused in other context. For example, it makes the parallelization of the solving process straightforward since each component can be treated independently.

Our approach has several areas for improvement. We do not backtrack, assuming that our exploration choices always enhance the sampler, which is not guaranteed. Our generator synthesis is general and may be less efficient than constraint-specific methods. Constrained data structures often exhibit symmetries, suggesting that symmetry-breaking techniques could be a natural extension. Additionally, non-uniform methods may offer a better balance between bug-finding effectiveness and computational cost. While uniformity can be expensive, surjective generators may provide sufficient coverage. Finally, incorporating corner-case analysis into solution sampling could further improve results in future work.

References

1 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable and nearly uniform generator of SAT witnesses. *CoRR*, abs/1304.1584, 2013. arXiv:1304.1584.

2 Koen Claessen, Jonas Duregård, and Michał H. Pałka. Generating constrained random data with uniform distribution. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 18–34, Cham, 2014. Springer International Publishing.

3 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000. doi:10.1145/357766.351266.

4 Simon Cruanes. Qcheck: Property-based testing for ocaml. https://github.com/c-cube/qcheck. Accessed: 2025-04-02.

5 Emanuele De Angelis, Fabio Fioravanti, Adrián Palacios, Alberto Pettorossi, and Maurizio Proietti. Property-based test case generators for free. In Dirk Beyer and Chantal Keller, editors, *Tests and Proofs*, pages 186–206, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-31157-5_12.

6 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

7 Rina Dechter, Kalev Kask, Eyal Bin, and Roy Emek. Generating random solutions for constraint satisfaction problems. In *AAAI/IAAI*, 2002.

8 Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability & Computing*, 13(4-5):577–625, 2004. doi:10.1017/S0963548304006315.

9 Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Random generators for dependent types. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004*, pages 341–355, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

10 Stefano Ermon, Carla P. Gomes, and Bart Selman. Uniform solution sampling using a constraint solver as an oracle. *CoRR*, abs/1210.4861, 2012. arXiv:1210.4861.

11 Vibhav Gogate and Rina Dechter. A new algorithm for sampling csp solutions uniformly at random. In *International Conference on Principles and Practice of Constraint Programming*, pages 711–715, September 2006. doi:10.1007/11889205_56.

12 Carla Gomes and Meinolf Sellmann. Streamlined constraint reasoning. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 274–289, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-30201-8_22.

13 Arnaud Gotlieb and Matthieu Petit. A uniform random test data generator for path testing. *Journal of Systems and Software*, 83(12):2618–2626, 2010. TAIC PART 2009 - Testing: Academic & Industrial Conference - Practice And Research Techniques. doi:10.1016/j.jss.2010.08.021.

14 W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970. URL: http://www.jstor.org/stable/2334940.

15 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, 2022. doi:10.1109/FOCS54457.2022.00021.

16 David Albert Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

17 Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158133.

18 Luca Martino. Parsimonious adaptive rejection sampling. *Electronics Letters*, 53, June 2017. doi:10.1049/el.2017.1711.

19 Luca Martino and Joaquín Míguez. A generalization of the adaptive rejection sampling algorithm. *Statistics and Computing*, 21:633–647, 2010. doi:10.1007/S11222-010-9197-9.

20 Albert Nijenhuis and Herbert Wilf. *Combinatorial Algorithms: For Computers and Hard Calculators*. Academic Press, Inc., USA, 2nd edition, 1978. doi:10.1016/C2013-0-11243-3.

21 Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A constraint solver based on abstract domains. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 434–454, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-35873-9_26.

22 Gilles Pesant, Claude-Guy Quimper, and Hélène Verhaeghe. Practically uniform solution sampling in constraint programming. In Pierre Schaus, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 335–344, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-08011-1_22.

23 Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. Uniform sampling of sat solutions for configurable systems: Are we there yet? In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 240–251, 2019. doi:10.1109/ICST.2019.00032.

24 Guy L. Steele Jr. and Sebastiano Vigna. Lxm: Better splittable pseudorandom number generators (and almost as fast). *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485525.

25 R.Endre Tarjan. A note on finding the bridges of a graph. *Information Processing Letters*, 2(6):160–161, 1974. doi:10.1016/0020-0190(74)90003-9.

26 Mathieu Vavrille, Charlotte Truchet, and Charles Prud'homme. Solution Sampling with Random Table Constraints. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 56:1–56:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2021.56.

27 Ke Xu, Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171(8):514–534, 2007. doi:10.1016/j.artint.2007.04.001.

28 J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling design constraints and biasing in simulation using bdds. In *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*, pages 584–589, 1999. doi:10.1109/ICCAD.1999.810715.

29 Ghiles Ziat, Matthieu Dien, and Vincent Botbol. Automated Random Testing of Numerical Constrained Types. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 59:1–59:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2021.59.

30 Ghiles Ziat, Marie Pelleau, Charlotte Truchet, and Antoine Miné. Finding solutions by finding inconsistencies. In *CP 2018 - 24th International Conference on Principles and Practice of Constraint Programming*, pages 1–16, Lille, France, August 2018. URL: https://hal.science/hal-01885769.

A Proof of correctness of Algorithm 3

Proof for Claim 3

Proof. Assume `spawn(e)` spawns points uniformly within an element e, i.e., for any $i \in e$, $P(i \mid e) = \frac{1}{v(e)}$, where $P(i \mid e)$ is the probability density function describing the likelihood of sampling a point i within the element e, and $v(e)$ is the volume of e. Let $\mathcal{S}$ be the set of all solutions satisfying constraints $\mathcal{C}$. The probability of sampling $i \in (\mathcal{I} \cup \mathcal{O})$ involves selecting an element $e \ni i$ and spawning i uniformly within e. The selection probability of e

is proportional to its volume:

$$P(e) = \frac{v(e)}{\sum_{e' \in \mathcal{I} \cup \mathcal{O}} v(e')}.$$

Thus, the joint probability of selecting e and spawning i is:

$$P(i \text{ is spawned}) = P(e) \cdot P(i \mid e) = \frac{v(e)}{\sum_{e' \in \mathcal{I} \cup \mathcal{O}} v(e')} \cdot \frac{1}{v(e)} = \frac{1}{\sum_{e' \in \mathcal{I} \cup \mathcal{O}} v(e')}.$$

Now, a point sampled with this process, without proper rejection, may not belong to $\mathcal{S}$. If $e \in \mathcal{I}$, all points $i \in e$ satisfy $\mathcal{C}$, so i is always accepted. If $e \in \mathcal{O}$, i is accepted only if $i \in \mathcal{S}$, and otherwise we repeat the sampling process. It follows that Algorithm 3 returns a particular solution $i \in \mathcal{S}$ either if:

- it samples i at the first attempt,
- or if the first attempt fails to produce a solution and i is sampled in a subsequent attempt.

The probability of sampling a particular solution $i \in \mathcal{S}$ is thus

$$\begin{aligned} P(i) &= \frac{1}{\sum_{e' \in \mathcal{I} \cup \mathcal{O}} v(e')} + P(\text{fail})P(i) \\ &= \frac{1}{(1 - P(\text{fail})) \sum_{e' \in \mathcal{I} \cup \mathcal{O}} v(e')} \end{aligned}$$

where $P(\text{fail})$ is the probability that a uniform element in $\mathcal{I} \cup \mathcal{O}$ does not satisfy $\mathcal{C}$. This is constant for all $i \in \mathcal{S}$, proving uniformity.

We wrote this proof in a discrete setting but a similar proof can be written in the continuous. ◁

Modeling and Solving a Composite Structure Design Problem with Constraint Programming

Miguel Antoons
UCLouvain, Belgium

Augustin Delecluse
TRAIL, ICTEAM, UCLouvain, Belgium

Samih Zein
High Performance Composite team, Cenaero, Gosselies, Belgium

Pierre Schaus
ICTEAM, UCLouvain, Belgium

Abstract

Composite structures are composed of plies (layers) of carbon fibers. For each ply, one must decide its orientation from the set of possible angles: -45°, 0°, 45°, and 90°. The stack of plies must follow strict constraints on the chosen orientations to achieve mechanical properties of the composite, such as sufficient buckling load. The design problem becomes more complex when determining the stack of plies for a complete surface material, that does not require the same number of plies in every region of the surface. Not only must the orientations be selected in each region, but it is also necessary to decide which plies are discontinued between adjacent regions. Thanks to its declarative nature, Constraint Programming (CP) offers an elegant modeling of the constraints, making it easy for designers to activate or deactivate them as needed. We propose a CP model, implemented in MiniZinc. The performance of this model on synthetic yet realistic instances when solved by different exact solvers, including Mixed Integer Programming (MIP) solvers, demonstrates the superiority of CP over MIP on our MiniZinc model, and over a commercial solution implemented by an industrial partner. It opens up the adoption of CP as an efficient building block of Computer-Aided Design tools for composite structures. By making the model and instances publicly available, we also hope to facilitate the inclusion of this problem in CP solver competitions and stimulate further research in this area.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Backtracking; Mathematics of computing → Combinatoric problems; Computing methodologies → Combinatorial algorithms; Computing methodologies → Model development and analysis; Applied computing → Aerospace

Keywords and phrases Constraint Programming, Composite Structures, Design Rules, MiniZinc

Digital Object Identifier 10.4230/LIPIcs.CP.2025.41

Category Short Paper

Supplementary Material *Software (Source code)*: `https://github.com/augustindelecluse/composite_structure` [1], archived at `swh:1:dir:6a1da259835bc6788634d1ba52bda3a11d8b03b1`

Funding *Augustin Delecluse*: This work was supported by Service Public de Wallonie Recherche under grant n°2010235 – ARIAC by DIGITALWALLONIA4.A.

1 Introduction

Composite structures are widely used in the industry to design components such as aeronautical or automotive pieces, due to their high strength-to-weight ratio. A composite structure is typically subdivided into several panels, and each panel is made up of multiple thin layers, or plies, arranged in a specific stacking sequence. The fibers in each ply are commonly oriented

© Miguel Antoons, Augustin Delecluse, Samih Zein, and Pierre Schaus;
licensed under Creative Commons License CC-BY 4.0

at one of four discrete angles in $\{0°, \pm 45°, 90°\}$. An example of a composite structure is a plane wing, as shown in Figure 1, where we can see the stacking sequence of a panel on the right. Due to manufacturing limitations, as well as to ensure consistent load transfer and meet required stiffness and strength, design constraints restrict how a ply may be oriented, depending on its neighbors and on adjacent panels.

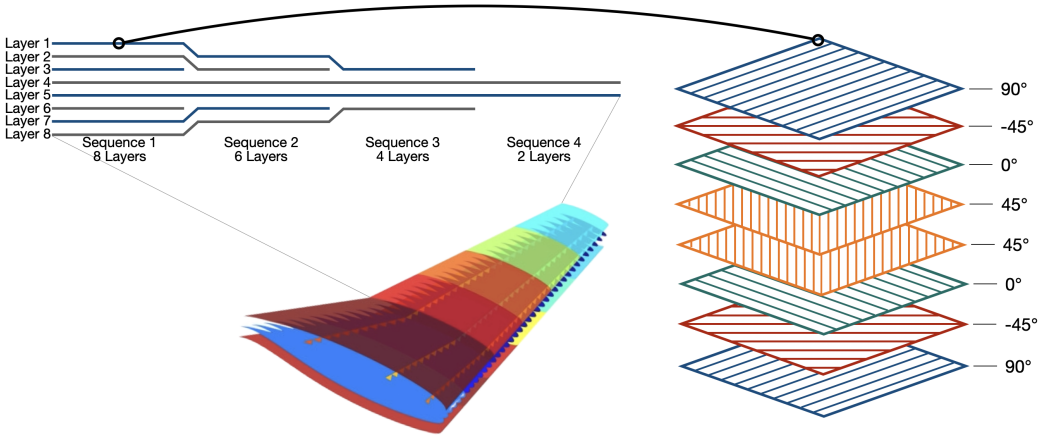


Figure 1 Example of the use of composite structures, here with a plane wing.

As can be seen in a review on the optimization of composite structures [15], this type of problem has so far been mainly addressed using genetic algorithms. A notable exception is the exact algorithm proposed in [21], implemented in the commercial tool of our industrial partner, and which we compare with. Our contributions are as follows:

1. The introduction of a pure declarative constraint programming model that can be used to generate a consistent composite structure.
2. An empirical validation of the model through Minizinc, improving the generation of admissible sequences proposed in [21].

The paper is organized as follows. Section 2 first presents the problem definition for deriving stacking sequences of a composite structure. A Constraint Programming (CP) model is introduced in Section 3. Section 4 is dedicated to the related work and mostly explains the main ideas of the exact custom algorithm of [21]. The CP model, implemented in MiniZinc, is experimentally validated in Section 5 and compared with the approach of [21]. Finally, Section 6 concludes and highlights possible future research directions.

2 Design Constraints for Composite Structures

The problem is to decide the orientation for each ply in each stacking sequence while satisfying rules specific to each stacking sequence and connecting those rules between adjacent sequences.

A first set of rules are the structural ones for each individual stacking sequence S_i.

- **D1**: The number of plies for each orientation is imposed and denoted by $n_i^{-45}, n_i^0, n_i^{45}, n_i^{90}$. We also define n_i as the sum of these: $n_i = n_i^{-45} + n_i^0 + n_i^{45} + n_i^{90}$.
- **D2**: The angle difference between two consecutive plies cannot be equal to 90°. This means that a -45° ply cannot be followed by a 45° ply, a 0° ply cannot be followed by a 90° ply and the inverse of those two cases is also prohibited.
- **D3**: A succession of four or more plies having the same angle is disallowed.

- **D4**: The sequence must be symmetric vertically (e.g. $(0°, 45°, 45°, 0°), ...$). However, some exceptions are accepted at the center of the sequence in addition to symmetrical patterns, namely $(-45°, 45°)$, $(-45°, 0°, 45°)$, $(-45°, 90°, 45°)$ and their inverses.
- **D5**: The top and bottom plies are only allowed to take values in $\{-45°, 45°\}$.

A second set of constraints consists of the blending rules, which restrict how neighboring stacking sequences relate to each other. For every pair of neighboring stacking sequences S_i, S_j, an implicit directed edge points from the sequence with the larger number of plies toward the one with fewer plies ($n_i > n_j$). We call i a parent of j. A directed graph represents the neighboring relationships. This graph must be acyclic for the structure to be feasible. This set of edges is denoted E.

- **B1**: The subsequence S_j (the thinner one) must be a subsequence of S_i (the thicker one). In the event S_j has multiple parents, its plies must be contained in all of them.
- **B2**: A ply can either be shared between two neighboring stacking sequences or be discontinued, but two continuous plies cannot cross each other. This can be seen as having the same ordering for the common plies between neighboring sequences: given a parent sequence $S_i = (A, B, C)$, a child sequence could be $S_j = (A, C)$ but cannot be $S_j = (C, A)$ as the ordering is not preserved.
- **B3**: The top ply of S_i must be the same as the top ply of S_j, and the same applies to their bottom plies. This ensures the continuity of the top and bottom surface plies throughout the entire structure.
- **B4**: A maximum of three consecutive layers can be dropped at once from S_i to S_j.

Depending on the structure requirements in practical applications, some rules may be ignored. The blending rules B1-B4 are always present, to ensure a global coherence between adjacent sequences. In the same manner, the design rule D1 is always enforced to generate sequences of required size. However, the design rules D2-D5 may be absent in some settings. This further motivates the use of declarative paradigms such as CP to tackle this problem.

▶ **Example 1.** Given the input graph from Figure 2, a solution satisfying most of the constraints is shown in Figure 3. In this solution, the constraints satisfied are B1, B2, B3, B4 and D1, which are always activated as well as the optional constraints D3, D4 and D5. There are some places where an angle difference of 90° is present, which is prohibited by constraint D2. This is for instance the case for $S_1[2] = 90°$ and $S_1[3] = 0°$.

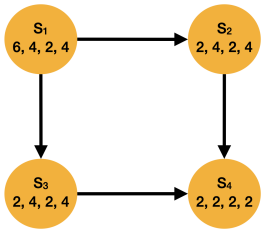


Figure 2 An example of how we represent composite structures for our CP model.

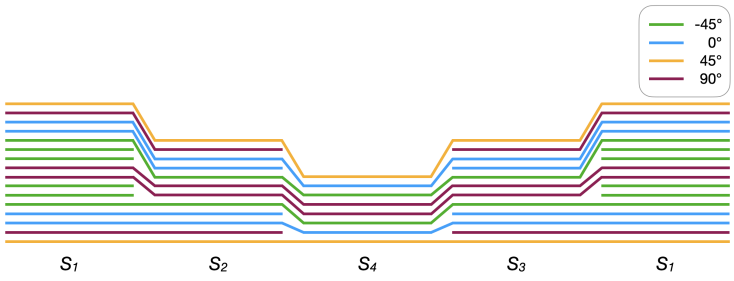


Figure 3 An example solution of our model with a limited set of constraints activated.

3 CP Model

3.1 Decision Variables

The main decision variables of our problem are the orientations that each layer k can take in every stacking sequence i:

$$S_i[k] \in \{Id_{-45},\ Id_0,\ Id_{45},\ Id_{90}\} \qquad \forall\ 1 \leq k \leq n_i \qquad \forall i \in \mathcal{I} \tag{1}$$

Where $Id_o, o \in \{0°, \pm 45°, 90°\}$ denotes the plies orientation and $\mathcal{I} = \{1 \dots |S|\}$ is the set of indices for all stacking sequences S. A second set of variables $Y_{i,j}[k] \in \{1, \dots, n_i\}, \forall 1 \leq k \leq n_j$, for all $(i,j) \in E$ is used to ensure the continuity of layers between adjacent regions. More precisely $Y_{i,j}[k]$ indicates that the ply k of the stacking sequence j is the continuation of the ply $Y_{i,j}[k]$ in the neighboring stacking sequence i. For example, referring to Figure 2 and 3, $Y_{1,2}[6] = 8$ since the sixth ply from S_2 is the same as the eighth ply from S_1.

3.2 Constraints

To ease the constraint notations, we first define some set of indices and constants. $\mathcal{I}_{even} = \{i \in \mathcal{I} \mid n_i \text{ is even}\}$, $\mathcal{I}_{odd} = \{i \in \mathcal{I} \mid n_i \text{ is odd}\}$ are the sets of indices for sequences of even and odd length, respectively. Moreover, $m_i := \lfloor n_i/2 \rfloor$ denotes the midpoint of a sequence $S_i \in S$.

The structural constraints D1-D5 on each stacking sequence are expressed on $S_i[k]$ variables while the blending constraints B1-B4 are expressed on $y_{i,j}$ variables. The formal definition of the constraints is given next.

$$\text{cardinality}(S_i, [Id_{-45}, Id_0, Id_{45}, Id_{90}], [n_i^{-45}, n_i^0, n_i^{45}, n_i^{90}]) \qquad \forall i \in \mathcal{I} \tag{2}$$

$$(S_i[k], S_i[k+1], S_i[k+2], S_i[k+3]) \in \text{allowed_tuples} \qquad \forall 1 \leq k \leq n_i - 3, \forall i \in \mathcal{I} \tag{3}$$

$$S_i[k] = S_i[n_i - k + 1] \qquad \forall 1 \leq k < \lfloor n_i/2 \rfloor, \forall i \in \mathcal{I} \tag{4}$$

$$(S_i[m_i], S_i[m_i+1]) \in \text{allowed_even} \qquad \forall i \in \mathcal{I}_{even} \tag{5}$$

$$(S_i[m_i], S_i[m_i+1], S_i[m_i+2]) \in \text{allowed_odd} \qquad \forall i \in \mathcal{I}_{odd} \tag{6}$$

$$S_i[1] \in \{Id_{-45}, Id_{45}\} \qquad \forall i \in \mathcal{I} \tag{7}$$

$$S_i[n_i] \in \{Id_{-45}, Id_{45}\} \qquad \forall i \in \mathcal{I} \tag{8}$$

$$S_j[k] = S_i[Y_{i,j}[k]] \qquad \forall 1 \leq k \leq n_j, \forall (i,j) \in E \tag{9}$$

$$Y_{i,j}[k] < Y_{i,j}[k+1] \qquad \forall 1 \leq k < n_j, \forall (i,j) \in E \tag{10}$$

$$Y_{i,j}[1] = 1 \qquad \forall (i,j) \in E \tag{11}$$

$$Y_{i,j}[n_j] = n_i \qquad \forall (i,j) \in E \tag{12}$$

$$Y_{i,j}[k+1] - Y_{i,j}[k] \leq 4 \qquad \forall 1 \leq k < n_j, \forall (i,j) \in E \tag{13}$$

The design rules D1-D5 correspond to constraints (2)-(8). The D1 constraint is enforced in (2) with a global cardinality constraint [10]. It ensures that the requested plies of every angle, requested by the user, are present in every sequence. This constraint is applied on every stacking sequence S_i and takes three arguments:

1. The sequence of variables forming the stacking sequence.
2. The possible values, one for each angle,
3. The cardinalities, one for every possible angle.

Then, the D2 and D3 rules are implemented in (3) by a single table constraint [2, 16, 5, 11] that covers both prohibiting 90° angles and disallowing four or more consecutive plies that have the same angle. It does so by applying a table on every set of four consecutive plies.

This table, represented by allowed_tuples, contains all tuples that have at least one different ply from the three others and that does not contain a 90° gap anywhere. This constraint is also enforced on all stacking sequences. In cases where only one constraint between D2 and D3 is activated, the tuples are adapted.

Next is the D4 rule. The equality constraint (4) enforces every ply to be the same as their opposite ply, resulting in the symmetry required by the D4 rule. So, for instance, the first ply must be the same as the last, the second must be the same as the penultimate and so on. This goes on until the middle two plies in case of an even thickness sequence, or until the middle three plies in case of an odd thickness sequence. Indeed, this allows for the exception to the D4 rule, allowing a dissymmetry in the middle to mitigate the problem of odd cardinalities since the symmetry constraint only works when all cardinalities are even. This exception is formulated in (5) in the case of an even thickness and in (6) in the case of an odd thickness. We can see that a table constraint is applied on the middle two or three plies of every stacking sequence. In addition to allowing for a symmetric suite of plies (in case all cardinalities are even), it also allows for the $(-45, 45)$, $(-45, 0, 45)$, $(-45, 90, 45)$ suites and their inverses. For this constraint to work properly, 90° gaps must be possible in the middle and thus the constraint implemented in (3) is relaxed in the middle only. Moreover, we expect CP solvers to represent constraint (4) by reusing the same variables for $S_i[k]$ and $S_i[n_i - k + 1]$ instead of using equality constraints, which reduces the number of variables[1].

The last design rule, D5, is implemented by (7) and (8) by redefining the domain of the top and bottom plies of every stacking sequence (i.e. $S_i[1]$ and $S_i[n_i]$, respectively), so that only values representing the -45° angle or the 45° angle are allowed.

The blending rules are enforced by (9)-(13), and are mandatory in every variant of the problem. The B1 rule is implemented in (9) using element constraints [8]. It ensures that every ply of a subsequence S_j must be present in all its parents denoted by S_i. In our model, this is enforced using an element constraint between S_i and S_j. This means that every ply from a thinner sequence must come from its thicker neighbors. This constraint is applied on every edge of the input graph.

The B2 rule is then implemented in (10). This rule prohibits plies from crossing each other when going from one sequence to another, which is ensured by forcing each index variable ($Y_{i,j}[k]$) to be strictly lower than the next ($Y_{i,j}[k+1]$).

The B3 rule is implemented in (11) and (12) by fixing the domains of the $Y_{i,j}[1]$ and $Y_{i,j}[n_j]$ to 1 and n_i respectively. Fixing these values will cause the top and bottom plies of every subsequence to be the same as their neighboring subsequences, and thus cause the top and bottom plies to be continuous across the whole structure.

Finally, the B4 rule is implemented in (13). In order to prevent drop-offs of four or more consecutive plies in between sequences, we can define a constraint that forces the absolute difference of two consecutive index variables (i.e. $Y_{i,j}[k]$ and $Y_{i,j}[k+1]$) to be less than or equal to four. In other words, this constraint allows a maximum of three consecutive plies to stop at once between two sequences, satisfying the B4 rule.

4 Related Works

A first exact approach was proposed in [22]. It iteratively generates a set of admissible structures (i.e., stacking sequences that meet all coherence constraints), performs numerical simulations, and selects the sequence(s) yielding the highest buckling load. However, it does

[1] Using the same variables for (4) instead of equality constraints is performed by MiniZinc[14], used in our experiments

not include the blending constraints. The approach of Zein et al. (2014) [21] solves the same problem as ours. Despite the title, the problem is not formalized as a declarative CSP. Instead, the authors introduce a custom depth-first search algorithm. During the search, stacking sequences are considered one by one in decreasing order of their number of layers. For each stacking sequence, branching decisions related to blending constraints are made before those related to design constraints. The approach does not rely on a solver with variables, domains, and a fixed-point computation of generic constraints. The algorithm verifies the consistency of the current layer in the stacking sequence with respect to the prefix of already fixed stacking sequences from previous decisions. If an infeasibility is detected for a layer, the search backtracks. The rules are thus enforced as checkers that verify the validity of a prefix, rather than through a filtering algorithm acting on all the variables in its scope as in a standard CP approach. This means that in a stack S of n plies, the ply orientation $S[k]$ is only deemed invalid if it conflicts with plies $S[1..k-1]$ but no inference is made regarding subsequent plies $S[k+1..n]$. This algorithm shares similarities with Generative Constraint Programming [17], which lazily prunes only the domain of the next variable to instantiate taking into account the prefix of already fixed variables. Our approach, on the other hand, relies on a pure declarative model where the search is decoupled from the model. While it is understandable that the custom approach uses a search algorithm that makes decisions in the order of stacking sequences to facilitate the verification of blending constraints, it is unclear whether this is the best search strategy, given that all layers are interconnected through blending constraints. Our approach can utilize efficient black-box searches that could detect or learn that, for instance, branching on the thinnest stack first might be more effective.

5 Experimental results

To assess the performance of our model, we generated in agreement with our industrial partner a set of instances representative of the ones that are generally processed by their software. Four datasets were generated using grid sizes of 3x3, 4x4, 5x5 and 6x6[2], stacking sequences, each containing 40 instances. Each node in the grid is connected to a minimum of two and up to four of its neighbors (top, left, bottom, right) in the grid by a directed edge, whose direction is chosen randomly. The cardinality constraints on the number of plies per orientation in each layer were selected uniformly with up to 60 plies per stacking layer. The retained instances are such that none is trivially detected as infeasible by the industrial solver when adding all constraints (B1-B4, D1-D5).

The model was implemented using MiniZinc [14], allowing us to easily experiment with the solving of our problem using different solvers. In particular, three different CP and Lazy Clause Generation solvers are compared: Gecode [18], OR-Tools [4] and Chuffed [3]. We also tried two Mixed Integer Linear Programming (MILP) [20] solvers using their MiniZinc backend: Gurobi [7], HiGHS [9]. Lastly, the performance of the solution deployed by our industrial partner that implements the approach of [21] with all constraints activated is also reported. Instances and models are available at `https://github.com/augustindelecluse/composite_structure`.

Figure 4 reports the number of instances solved over time with the 3 tested models, using a timeout of 15 minutes. Each model includes constraints B1-B4 and D1. Depending on the application, not all optional constraints must be considered, which was also the case in [21]. The first model (top) does not include any optional constraints, the second (middle) adds constraints D3 and D4, and the third (bottom) includes all constraints.

[2] Figure 2 corresponds to a grid size of 2x2

OR-Tools appears as the most stable solver, and consistently solves more than 80% of the instances within 15 minutes in all datasets and all models. Additionally, its performance does not degrade much when the size of the instances increases, which is not the case with Gecode, Chuffed and the industrial solver. Although not visible in this aggregated plot, Gecode is generally faster on feasible instances, while OR-Tools performs better at proving infeasibility. MILP solvers are behind CP solvers, and Gurobi performs better than Highs. The industrial solver achieves performance competitive with Gecode but is outperformed by the lazy-clause generation solvers OR-Tools and Chuffed. This indicates that clause learning significantly enhances the ability to solve this problem robustly.

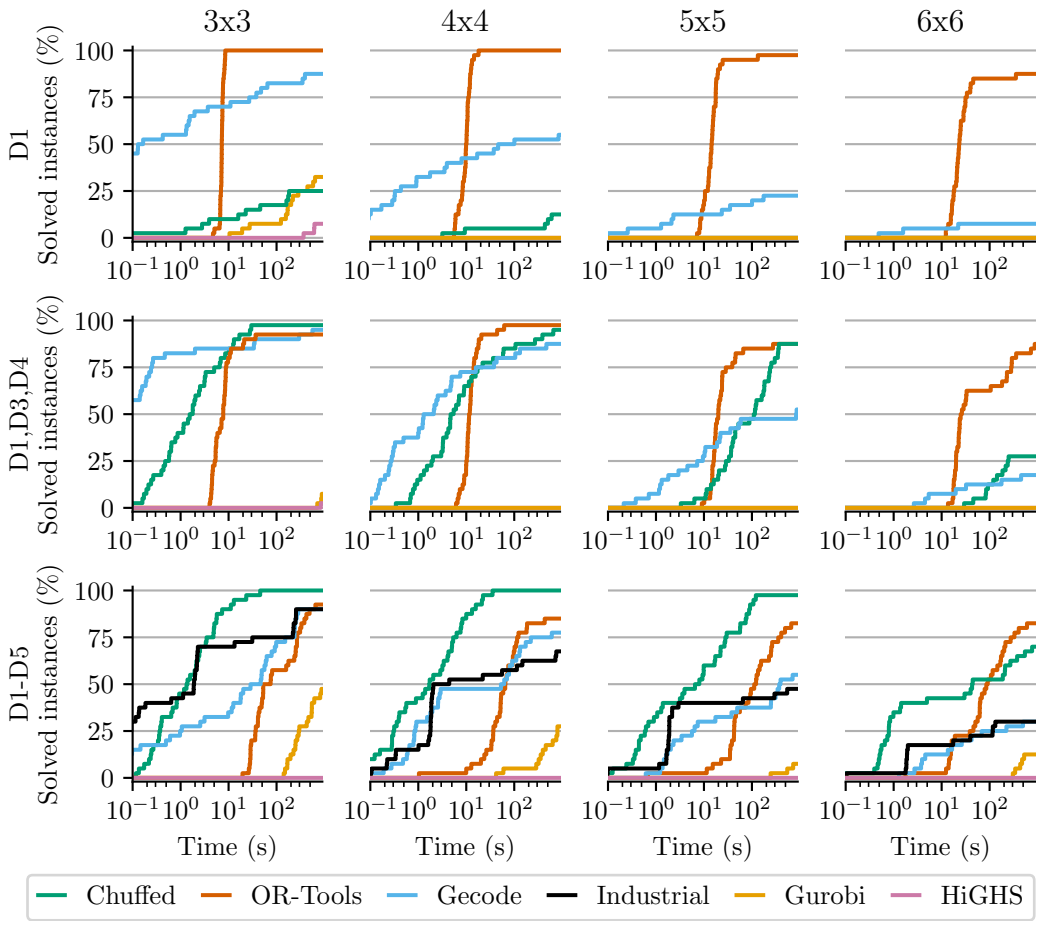


Figure 4 Solver performances with different sets of design constraints (rows). Each column denotes a given dataset.

6 Conclusion and Future Work

We have introduced a CP model for the Composite Structure Design Problem, including the blending constraints between neighboring stacking sequences. This model was implemented in MiniZinc and tested with various CP-based solvers and MIP solvers. The results clearly show that CP outperforms MIP on the MiniZinc models, with lazy clause generation solvers enabling the solution of even more instances. This approach also outperforms a state-of-the-art industrial solver that employs a dedicated algorithm for this problem.

In future work, we plan to improve the filtering and search. For filtering, the standard MiniZinc library does not include the AtMostSeqCard constraint from [19]. This constraint limits the number of consecutive variables that can take a specific value v and restricts the total number of variables instantiated to v. We believe that this constraint could help reduce the solving time by acting as a redundant constraint. In addition, our goal is to experiment with different search strategies to further accelerate the solving process. More specifically, one could try to use custom search strategies, possibly combined with conflict-based searches [6] and restart-based strategies with no-good learning [12]. Finally, we would like to include some possible objective functions or hybridize the model with a finite element simulator that would allow for a more fine-grained estimation of the mechanical properties. An interesting approach we could try in that direction is the so-called *Empirical Decision Model Learning* framework [13].

References

1 Miguel Antoons, Augustin Delecluse, Samih Zein, and Pierre Schaus. composite_structure. Software, swhId: `swh:1:dir:6a1da259835bc6788634d1ba52bda3a11d8b03b1` (visited on 2025-07-23). URL: `https://github.com/augustindelecluse/composite_structure`, `doi:10.4230/artifacts.24098`.

2 Kenil CK Cheng and Roland HC Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010. `doi:10.1007/S10601-009-9087-Y`.

3 Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed - a lazy clause solver, 2016. URL: `https://github.com/chuffed/chuffed`.

4 Thibaut Cuvelier, Frederic Didier, Vincent Furnon, Steven Gay, Sarah Mohajeri, and Laurent Perron. OR-Tools' vehicle routing solver: a generic constraint-programming solver with heuristic search for routing problems. In *24e congrès annuel de la société française de recherche opérationnelle et d'aide à la décision*, 2023.

5 Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings 22*, pages 207–223. Springer, 2016. `doi:10.1007/978-3-319-44953-1_14`.

6 Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search for scheduling problems. In *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*, pages 140–148. Springer, 2015. `doi:10.1007/978-3-319-23219-5_10`.

7 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024. URL: `https://www.gurobi.com`.

8 Pascal Van Hentenryck and Jean-Philippe Carillon. Generality versus specificity: An experience with AI and OR techniques. In *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence*, pages 660–664, 1988. URL: `http://www.aaai.org/Library/AAAI/1988/aaai88-117.php`.

9 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`.

10 Régin Jean-Charles. Generalized arc consistency for global cardinality constraint. *American Association for Artificial Intelligence (AAAI 1996)*, pages 209–215, 1996. URL: `http://www.aaai.org/Library/AAAI/1996/aaai96-031.php`.

11 Christophe Lecoutre. STR2: optimized simple tabular reduction for table constraints. *Constraints*, 16:341–371, 2011. `doi:10.1007/S10601-011-9107-6`.

12 Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009. doi:10.1016/J.ARTINT.2009.09.002.

13 Michele Lombardi, Michela Milano, and Andrea Bartolini. Empirical decision model learning. *Artificial Intelligence*, 244:343–367, 2017. doi:10.1016/J.ARTINT.2016.01.005.

14 Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

15 SKMSS Nikbakt, Saeed Kamarian, and Mahdieh Shakeri. A review on optimization of composite structures part i: Laminated composites. *Composite Structures*, 195:158–185, 2018.

16 Guillaume Perez and Jean-Charles Régin. Improving GAC-4 for table and MDD constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 606–621. Springer, 2014. doi:10.1007/978-3-319-10428-7_44.

17 Florian Régin and Elisabetta De Maria. Generative constraint programming revisited. In *2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 18–26. IEEE, 2024. doi:10.1109/ICTAI62512.2024.00012.

18 Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. Modeling and programming with gecode. *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael*, 1, 2010.

19 Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. An optimal arc consistency algorithm for a particular case of sequence constraint. *Constraints*, 19:30–56, 2014. doi:10.1007/S10601-013-9150-6.

20 Juan Pablo Vielma. Mixed integer linear programming formulation techniques. *Siam Review*, 57(1):3–57, 2015. doi:10.1137/130915303.

21 S Zein, P Basso, and S Grihon. A constraint satisfaction programming approach for computing manufacturable stacking sequences. *Computers & Structures*, 136:56–63, 2014.

22 Samih Zein, Benoît Colson, and Stéphane Grihon. A primal-dual backtracking optimization method for blended composite structures. *Structural and Multidisciplinary Optimization*, 45(5):669–680, 2012.

Towards Modern and Modular SAT for LCG

Jip J. Dekker
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Alexey Ignatiev
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia

Peter J. Stuckey
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Allen Z. Zhong
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Abstract

Lazy Clause Generation (LCG) is an architecture for building Constraint Programming (CP) solvers using an underlying Boolean Satisfiability (SAT) engine. The CP propagation engine lazily creates clauses that define the integer variables and impose problem restrictions. The SAT engine uses the clausal model to reason and search, including, crucially, the generation of nogoods. However, while SAT solving has made significant advances recently, the underlying SAT technology in most LCG solvers has largely remained the same. Using a new interface to SAT engines, IPASIR-UP, we can construct an LCG solver which can swap out the underlying SAT engine with any that supports the interface. This new approach means we need to revisit many of the design and engineering decisions for LCG solvers, to take maximum advantage of a better underlying SAT engine while adhering to the restrictions of the interface. In this paper, we explore the possibilities and challenges of using IPASIR-UP for LCG, showing that it can be used to create a highly competitive solver.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization

Keywords and phrases Lazy Clause Generation, Boolean Satisfiability, IPASIR-UP

Digital Object Identifier 10.4230/LIPIcs.CP.2025.42

Category Short Paper

Supplementary Material *Software (Source Code)*: `https://github.com/huub-solver/huub` [11] archived at `swh:1:dir:b28854946ae60e86a37051ea89465cce8b84b7ed`

Funding This research was partially funded by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Project ID IC200100009.

1 Introduction

Lazy Clause Generation [31] is an architecture for building Constraint Programming solvers by making use of an underlying SAT engine [27]. LCG solvers such as Chuffed [10] and CP-SAT [32] define the state-of-the-art for constraint programming and have consistently scored the most points in the Free search category in the MiniZinc challenge ever since 2010 when one was first entered.

© Jip J. Dekker, Alexey Ignatiev, Peter J. Stuckey, and Allen Z. Zhong;
licensed under Creative Commons License CC-BY 4.0

However, the SAT infrastructure used in LCG solvers is mainly based on SAT technology from at least a decade ago. Meanwhile, SAT solvers have made significant advances, including inprocessing [24], chronological backtracking [29], as well as improvement to search strategies by including local search components [9].

With these developments, it is worthwhile to investigate how a more modern conflict-driven clause learning (CDCL) SAT solver impacts the design choices of Lazy Clause Generation solvers. In this paper, we make use of the IPASIR-UP interface [15, 16] for extending SAT solvers and developing a new LCG solver. While the interface limits the design decisions, it also directly benefits from all the improvements of modern SAT solvers. As a result, it becomes necessary to revisit and reassess key decisions in LCG to determine how they are affected by these improvements.

2 Background and Related Work

In this short paper, we assume that the readers have a reasonable understanding of how both CP and SAT solvers work. For more details, see e.g. CP [33] and SAT [19].

LCG solvers aim to combine the high-level reasoning of CP solvers, allowing, for example, integer variables and global constraints, with the powerful conflict analysis of SAT solvers. The integration is possible by creating CP propagators that make their inference available to the SAT solver in clausal form. A key challenge is mapping the representation of finite domain/integer variables from the CP model to Booleans. The usual approach is to represent variables domains using both equality $[\![x = d]\!]$ and bounds $[\![x \geq d]\!]$ Booleans. Each time the LCG solver propagates, which is equivalent to setting one of these Booleans to true or false, it must be able to explain the propagation, by computing a clause which is a consequence of the CP model which will cause the propagation in the current solver state.

The first LCG solver [30] was built on top of MiniSAT 2.0β. Representation of integer variables was eager, creating both the bounds $[\![x \geq d]\!]$ and equality $[\![x = d]\!]$ literals for each integer x before commencing search. The SAT solver controlled the search completely, and the CP engine was woken up once unit propagation ceased. This initial solver used *clausal explanation*, that is, when a propagator makes a new inference, it generated a clause that defined the propagation, and adds it to the SAT engine *permanently*. The new clauses then started more unit propagation in the SAT engine. This continued until failure was detected, and the SAT engine backjumped as usual, or a solution was found.

The next LCG architecture [17] reversed the control. Here the constraint programming engine was in control, performing search decisions. The unit propagation of clauses is treated as a (highest priority) propagator in the CP engine. The CP trail is extended with reason clauses for each propagation or failure, and conflict resolution works on the CP trail (in the usual CDCL manner). This architecture introduced *lazy encoding*, where equality literals were only created on demand, while bounds literals were created up front for integers x with small initial domains, or lazily during search for integers with large initial domains. In this architecture, the explanation strategy was *forward explanation*, where each propagator built an explanation clause for each new propagation, at the time of propagation, but these explanation clauses only lived on the trail, and are removed when backjumping.

Chuffed [10] was built around a customized version of MiniSAT 2.2. It is extended to use a *spaghetti stack*, so it can insert into the stack for choicepoints earlier than the current choicepoint. It is much more similar to a SAT engine, with a propagation loop which runs when unit propagation quiesces. Again here explanation clauses only live in the trail, but propagators could implement forward explanation, or *backwards explanation* where

the explanation clause is computed during conflict analysis, and not at propagation time. Chuffed did not implement propagators for many constraints, and instead encodes a number of constraints, such as `element` and `table`, into clauses.

After Chuffed was made open source in 2016, there has been a surge in LCG solvers. This might have been championed by the high-profile CP-SAT solver [32], developed as part of Google's OR-Tools. This solver includes novel CP and SAT engines that implement some more modern SAT features, such as Glucose's Literal Block Distance [3]. Other notable enhancements include the integration of a simplex based linear programming propagation [2], and a parallel portfolio approach of different variations of the solver, which makes the solver robust to different problem classes. At the same time, alternative solvers, such as Geas [22] and Pumpkin [12], have been developed by academics and have introduced exciting new features, such as core-guided search [21] and proof logging [20] for LCG.

A common trend among these new solvers is their tight integration and customization of the SAT solver employed. In the remainder of this paper, we will evaluate the opposite approach, where we maintain a clear boundary between the SAT and CP components, and the SAT solving is used without any customization.

3 LCG Architecture with a Modular SAT Engine

The recent IPASIR-UP interface [15, 16] augments a conflict-driven clause learning (CDCL) SAT solver [27] with the capability to invoke and interact with an external propagation engine to improve the CDCL solver's performance. The interface was originally implemented in the CaDiCaL solver [8] and successfully showcased in the setting of Satisfiability Modulo Theories (SMT) solving [4].

Although IPASIR-UP can be used to construct an LCG solver, it requires a change of perspective from the solver designer. Most LCG solvers use a SAT solver as a component to perform specific actions, such as clausal propagation and conflict resolution, within their search process. However, when using the IPASIR-UP interface, it is the SAT solver that runs the solving process, and it will make different calls to the external propagation engine at certain points in the process, much like the original LCG solver [30]. In our case, the external propagation engine will behave similarly to the CP component of most LCG solvers. It tracks the state of non-Boolean decision variables, such as integer variables, and oversees the running of propagators, which will communicate using literals and clauses.

Figure 1 shows the expected interactions between the SAT solver and our LCG engine. Hereinafter, when explaining the interplay between the SAT solver and the LCG backend, the names of the IPASIR-UP methods corresponding to the functionality being described are given in `verbatim`. The most important functionality of the IPASIR-UP interface is requesting external propagation. When this is required, the SAT solver will first notify the LCG backend of all literals that have been assigned since the last propagation (`notify_assignments`), and then ask it to propagate. To perform propagation, we maintain a priority queue of propagators that, given the new assignments, perform further inference [33]. We then activate the propagators in order until one performs any inference or the queue becomes empty. Propagators are required to inform the SAT engine about the outcome of their latest inference by providing literals that the SAT engine should assert (`propagate`) and clauses that can be added to the solver (`add_external_clause`). If no propagator performs any inference, then the LCG backend will report to be at fix point.

If the SAT solver performs conflict analysis that involves a literal inferred by the LCG backend, then the backend is expected to explain it by providing the corresponding antecedent clause (`add_reason_clause`). In the case of *backward explanation*, the backend reacts by

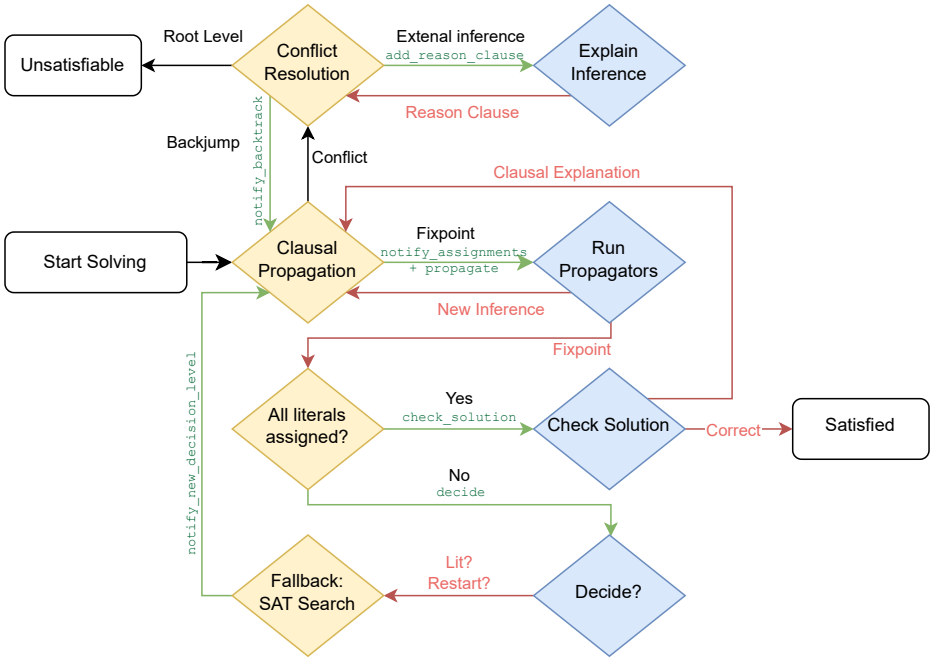


Figure 1 Depiction of the interaction between a SAT solver (yellow) and the LCG engine (blue) using IPASIR-UP interface. During green transitions, the SAT solver will make a call to the LCG. The red transitions represent the types of responses to these calls.

computing such a clause and passing it to the solver. Otherwise, in the case of *forward explanation*, it constructs and stores all antecedent clauses during propagation and then emits them to the SAT solver upon such a request.

Explanations passed by the LCG backend to the SAT engine through IPASIR-UP are allowed to contain only literals known to the solver during search. This represents an obstacle if propagators' literals are to be introduced lazily *during* backward explanation. This can, for example, prevent the strengthening of the explanation clauses through *lifting*, the process of generalizing an explanation. For example, given the constraint $2x + 2y + z < 5$, we might infer that $x \leq 1$ when $y \geq 1 \wedge z \geq 1$; however, we can infer the same when $z \geq 0$. So instead, the lifted explanation could be $(y \geq 1 \wedge z \geq 0) \rightarrow x \leq 1$. The above requirement prevents us from introducing (and defining) new literals at this point. So if $z \geq 0$ was an already existing literal, then this lifting optimization would be possible. However, if it is a *lazy* literal to be introduced for the explanation, then it is not.

For an LCG backend, `check_solution` behaves largely the same as `propagate`, since its task is to check that no constraints are violated to validate the solution. A complication is that any higher level decision variables that introduce literals lazily, such as integer variables, might not yet be fixed to a single value. The most straightforward solution to this problem is to assume that the variables take a value in their domain (e.g. the lower bound), and check the constraints using these values. If no conflicts are found under these assumed values, then a solution is found (with the assumed values). Otherwise, the conflict detected will result in a new clause that can be used by the SAT solver for further propagation, which will ensure that the combination of assumed values that caused the conflict cannot be taken.

It is well known that the search strategy determined by the branching heuristic used can have an enormous impact on solving performance. Using the IPASIR-UP's `decide` callback, the LCG backend can influence the search strategy. Most prominently, this method allows the backend to choose a literal to branch on. It can also be used to request the solver to restart, returning to a state where no decisions had been made. If no decision is made by the LCG backend, then the search strategy is left up to the SAT solver. Generally, it will make its decisions based on the variable state independent decaying sum (VSIDS) heuristic [28], and will use its own policy for restarts unless disabled.

Finally, the SAT solver is expected to inform the LCG backend of any changes to the decision level. It will inform the backend both when a new search decision level has been made (`notify_new_decision_level`) and when it backtracks to an earlier level (`notify_backtrack`). These notifications allow the LCG backend to maintain its own state that is reverted when backtracking. This can, for example, be used to maintain a state of which literals have been assigned and the current lower and upper bounds of integer variables.

4 Experimental Evaluation

We now introduce **Huub** [11], an open-source LCG solver implemented using the IPASIR-UP interface. We will use it to evaluate the performance of using modern SAT features and explore several design and engineering decisions for LCG solvers. The core engine of Huub is implemented in Rust 1.81.0 using CaDiCal 2.1.3 [8] as the back-end SAT solver, although in theory any SAT solver supporting the IPASIR-UP interface could be used. In the following experiments, the solver exercises through its FlatZinc interface. This allows us to use the MiniZinc Challenge [35] 2024 benchmarks, which consist of 20 problems with 5 instances each. Huub has propagators for all FlatZinc primitive constraints, but only supports the `all_different` and `disjunctive` global constraints. All other constraints are decomposed into these constraints or clauses using MiniZinc's decompositions. All experiments were run on a single core of an Intel Xeon Gold 6338 processor with 16 GB and a 20-minute time limit.

4.1 Comparison with State-of-the-art Solvers

Table 1 Comparison using instances from MiniZinc Challenge 2024.

Search	Solver	PAR2	OPTIMAL	UNSAT	SAT	UNK
SAT-based	CP-SAT	163832.54	54 (165.35)	3 (9.09)	41	2
	Chuffed	184494.97	50 (231.49)	2 (7.55)	31	17
	Huub	157613.18	55 (173.98)	4 (125.89)	35	6
Fixed	CP-SAT	241410.42	31 (119.30)	3 (9.08)	53	13
	Chuffed	252075.25	28 (125.60)	3 (12.20)	56	13
	Huub	250824.30	29 (192.61)	3 (78.75)	56	12

We first compare the baseline of our solver against Chuffed and CP-SAT. We compare the solvers in two different scenarios: (1) fixed search, where solvers must follow the search strategy in a model, and (2) SAT-based search, where solvers use the SAT search heuristic.

Table 1 presents the PAR2 scores obtained by the competitors calculated as the sum of the runtime plus twice the timeout for unsolved instances, i.e., the smaller PAR2 the better. For each configuration, we report the number of instances that are proven optimal

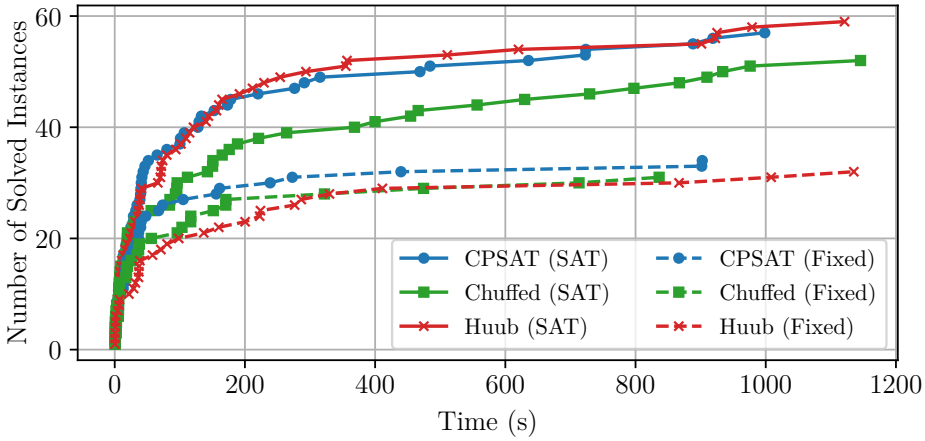


Figure 2 Cactus plot using instances from MiniZinc Challenge 2024.

(OPTIMAL), are proven unsatisfiable (UNSAT), where a solution without optimality proof is found (SAT), where no solutions are found in the time/memory limits (UNK). For the instance that are solved to completion, we also include the average solving time.

As shown in Table 1, Huub performs best when it uses the SAT search heuristic. When the fixed search strategy is used, Huub is outperformed by CP-SAT, but is better than Chuffed in terms of the PAR2 score. The cactus plot in Figure 2 also shows that Huub with the SAT-based search strategy outperforms the other solvers in terms of completed instances. Importantly, given that Huub currently implements only a limited set of propagators for global constraints, its performance has the potential for significant further improvement. A breakdown of the PAR2 scores for each problem, included as Section A, supports this observation. Specifically, CP-SAT and Chuffed outperform Huub substantially in the *accap*, *community-detection*, *network_50_cstr*, *tincy-cvrp*, *train-scheduling* and *yumi-dynamic* problems. For these problems, efficient propagators for global constraints, such as `diffn` [6], `cumulative` [34], and `global difference` [18], play crucial roles in solving performance. On the other hand, Huub demonstrates superior performance in problem categories such as *cable-tree-wiring*, *compression*, *fox-geese-corn*, *harmony*, and *word-equations*, where Huub has similar propagators to CP-SAT and Chuffed. This highlights the advantages of leveraging a modern and efficient SAT engine to enhance the performance of LCG solvers. We expect the overall performance of Huub can be improved with the integration of more efficient propagators for global constraints.

4.2 Inprocessing in Modern SAT Solvers

Using a SAT solver as modern as CaDiCaL enables us to use features that are not available in the SAT solvers used by other LCG solvers. Through the IPASIR-UP interface, Huub can easily leverage the latest advancements in modern SAT solvers. In particular, we will evaluate *inprocessing* techniques, which perform runtime simplification on the clause database, are widely recognized as essential for efficient SAT solving [14, 24]. Note that using the IPASIR-UP interface implicitly disables some pre/in-processing, which are not safe if the entire model is not known [16]. Additionally, our solver must sometimes reintroduce ternary clauses to ensure the consistency of integer literals.

Table 2 presents the results of an ablation study evaluating the impact of different inprocessing techniques including clause subsumption [5], variable elimination [13], failed literal probing [26], and globally blocked clause elimination [25]. The “Baseline” configuration does

Table 2 Comparison of different combinations of simplification techniques: clause subsumption (S), variable elimination (E), failed literal probing (P), and globally blocked clause elimination (C).

Configuration	PAR2	OPTIMAL	UNSAT	SAT	UNK
Baseline	157613.18	55 (173.98)	4 (125.89)	35	6
With All	154264.10	56 (174.73)	4 (117.36)	36	4
Without S	154259.11	56 (177.94)	4 (120.24)	35	5
Without E	157415.81	55 (172.44)	4 (124.77)	37	4
Without P	154514.60	56 (182.23)	4 (127.37)	36	4
Without C	154904.39	56 (189.04)	4 (128.99)	36	4

not perform any inprocessing, while "With All" enables all studied inprocessing techniques. The results indicate that enabling inprocessing techniques provides a slight, but consistent, improvement in solver performance compared to the baseline. Among the techniques studied, variable elimination has the most significant impact. Without it, Huub solves one fewer instance completely and has an increased PAR2 score. Removing any of the other inprocessing techniques results in an increased solving time for completed instances. This suggests that the performance benefits gained from inprocessing outweigh its computational overhead on these benchmarks. In the following, we enable all inprocessing techniques by default.

4.3 Revisit Explanation Mechanisms

In this section, we revisit a key decision in the construction LCG solvers, the way in which propagation is explained to the SAT engine. Chuffed uses both forward and backward explanation, and Huub implements the same strategy by default. However, with modern SAT solvers becoming increasingly efficient at handling clauses, pushing more clauses into the SAT engine can potentially enhance clause simplification and improve variable selection during search. To leverage these advantages, we implement a hybrid approach that combines clausal and backward explanation.

As a starting point, we experiment with using clausal explanations instead of forward explanation, and where the reasons were already created eagerly. We then introduce an additional threshold parameter L. Any propagation, using backward explanation, that is explained using a clause of size L or smaller will also use clausal explanation. This affects propagators handling global constraints with an undetermined number of variables, such as `linear`, `maximum` and `element`. Reason clauses are generated lazily when the number of involved variables exceeds the threshold; otherwise, they are generated eagerly.

Table 3 Comparison using different thresholds L for clausal explanation.

Configuration	PAR2	OPTIMAL	UNSAT	SAT	UNK
Forward	154264.10	56 (174.73)	4 (117.36)	36	4
$L = 0$	151590.61	57 (194.07)	4 (113.47)	36	3
$L = 3$	149572.99	58 (222.72)	4 (109.54)	34	4
$L = 5$	149860.45	58 (227.55)	4 (108.07)	36	2
$L = \infty$	157834.48	55 (174.20)	4 (116.44)	36	5

Table 3 presents the results of using different values of L. We note that when $L = 0$ only forward explanation is replaced by clausal propagation, and conversely when $L = \infty$ all propagation is through clausal explanation. The results indicate that increasing the number

of clauses pushed to the SAT solver generally improves the number of optimally solved instances compared to the baseline. Meanwhile, pushing more clauses into the SAT engine can introduce overhead, as we observe that there are more unknown instances with $L = 3$ compared to $L = 0$. $L = 5$ achieves the highest number of optimal instances and also reduces the number of unknown instances. However, the $L = \infty$ does not improve performance overall. Its computational overhead leads to a lower number of (completely) solved instances. This suggests that a moderate threshold allows the SAT solver to benefit from additional clauses, without introducing excessive overhead.

4.4 Disabling High-Level Propagation

Table 4 Comparison of different mechanisms for adaptive propagations.

Configuration	PAR2	OPTIMAL	UNSAT	SAT	UNK
Always Enabled	154264.10	56 (174.73)	4 (117.36)	36	4
Adaptive Engine	280496.22	20 (308.80)	4 (122.18)	72	4
$T = 10^{-2}$	154064.75	56 (174.82)	4 (118.25)	37	3
$T = 10^{-4}$	153367.16	56 (163.04)	4 (116.16)	35	5
$T = 10^{-6}$	151331.5	57 (188.60)	4 (118.82)	35	4

Our final experiment explores adaptive propagation. Previous work using IPASIR-UP [23] introduces an *adaptive* propagation engine for pseudo-Boolean constraints. This engine conditionally disables itself based on the number of calls to the propagation engine, the number of literals it propagates, and the proportion of assignments that satisfies its constraints. We reproduce the experiments in Huub using the same set of instances, where the goal is to enumerate all solutions. When the LCG engine is always enabled, all instances are solved in 2095 seconds. While disabling the LCG engine using the same conditions reduces the solving time to 1759 seconds. This reaffirms the claims in [23] and suggests that disabling higher-level propagation can lead to improvements in solver performance.

Since Huub supports a wider variety of constraint types, each of which may have different impacts on solving, we refined the adaptive mechanism to be more fine-grained. We introduce an adaptive scoring mechanism, similar to NVSIDS [7], that conditionally disables individual propagators based on their activity. The score for each propagator is initialized as 1. The score s is updated to $s' = f \cdot s$ if the propagator is called but neither propagates literals nor detects a conflict; otherwise, the score is updated as $s' = f \cdot s + (1 - f)$. The key difference with NVSIDS is that f takes the value $(1 - 0.5^c)$, where c is the number of conflicts encountered. The intuition behind this change is that, initially, the SAT-based search heuristic is random, and CP propagation may not yield useful information. As the search progresses, however, the propagator should become more active for proving optimality. If the score falls below a predefined threshold, the execution of that propagator is postponed until a solution is found, effectively transforming it into a checker that verifies whether assigned values remain consistent when all variables are fixed, which is done in `check_solution`.

Table 4 presents the results of applying the adaptive engine and adaptive propagators with different thresholds T to instances from the MiniZinc Challenge 2024. As shown, directly applying the adaptive engine mechanism reduces the number of instances solved to optimality, whereas adaptive propagators with an appropriate threshold slightly improve performance. Although the overall performance remains similar to the baseline, the adaptive

propagators approach demonstrates positive results for specific problems. In the *cable-tree-wiring*, *community-detection*, and *train-scheduling* problems, it either reduced solving times or improved solutions. Further details can be found in Appendix B. These results suggest that adaptive propagation, while promising, requires further investigation.

5 Conclusion

We present Huub, an LCG solver built based on the IPASIR-UP interface to SAT engines. This modular design allows us to easily swap in any SAT solver that supports the interface, enabling the solver to benefit from the latest advancements in SAT solving technology. Our implementation demonstrates competitive performance compared to state-of-the-art LCG solvers, and we expect further improvements with more global constraint propagators.

As SAT solvers become more and more efficient in handling clauses, it is timely to revisit the design and engineering of LCG solvers. In this work, we investigate various explanation strategies and adaptive propagation techniques. In the future, we plan to explore other options, such as dynamically decomposing constraints and encoding them into the SAT solver on the fly [1].

References

1 Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Peter J Stuckey. To encode or to propagate? the best choice for each constraint in SAT. In *International Conference on Principles and Practice of Constraint Programming*, pages 97–106. Springer, 2013. doi:10.1007/978-3-642-40627-0_10.

2 T Achterberg. Constraint integer programming. *Ph. D. Thesis, Technische Universitat Berlin*, 2007.

3 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.

4 Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021. doi:10.3233/FAIA201017.

5 Roberto J Bayardo and Biswanath Panda. Fast algorithms for finding extremal sets. In *Proceedings of the 2011 SIAM International Conference on Data Mining*, pages 25–34. SIAM, 2011. doi:10.1137/1.9781611972818.3.

6 Nicolas Beldiceanu, Qi Guo, and Sven Thiel. Non-overlapping constraints between convex polytopes. In *International Conference on Principles and Practice of Constraint Programming*, pages 392–407. Springer, 2001. doi:10.1007/3-540-45578-7_27.

7 Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer, 2008. doi:10.1007/978-3-540-79719-7_4.

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

9 Shaowei Cai and Xindi Zhang. Deep cooperation of CDCL and local search for sat. In *Theory and Applications of Satisfiability Testing–SAT 2021: 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings 24*, pages 64–81. Springer, 2021. doi:10.1007/978-3-030-80223-3_6.

10 Geoffrey Chu. *Improving combinatorial optimization.* PhD thesis, Department of Computing and Information Systems, University of Melbourne, 2011.

11 Jip J. Dekker, Alexey Ignatiev, Peter J. Stuckey, and Allen Z. Zhong. Huub: Lazy Clause Generation Solver. Software, swhId: `swh:1:dir:b28854946ae60e86a37051ea89465cce8b84b7ed` (visited on 2025-07-24). URL: `https://github.com/huub-solver/huub`, `doi:10.4230/artifacts.24103`.

12 Emir Demirović, Maarten Flippo, Imko Marijnissen, Jeff Smits, and Konstantin Sidorov. Pumpkin Solver. URL: `https://github.com/ConSol-Lab/Pumpkin`.

13 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *International conference on theory and applications of satisfiability testing*, pages 61–75. Springer, 2005. `doi:10.1007/11499107_5`.

14 Katalin Fazekas, Armin Biere, and Christoph Scholl. Incremental inprocessing in SAT solving. In *Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22*, pages 136–154. Springer, 2019. `doi:10.1007/978-3-030-24258-9_9`.

15 Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. IPASIR-UP: user propagators for CDCL. In *SAT*, volume 271 of *LIPIcs*, pages 8:1–8:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. `doi:10.4230/LIPICS.SAT.2023.8`.

16 Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. Satisfiability modulo user propagators. *Journal of Artificial Intelligence Research*, 81:989–1017, 2024. `doi:10.1613/JAIR.1.16163`.

17 T. Feydy and P.J. Stuckey. Lazy clause generation reengineered. In I. Gent, editor, *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *LNCS*, pages 352–366. Springer-Verlag, 2009. `doi:10.1007/978-3-642-04244-7_29`.

18 Thibaut Feydy, Andreas Schutt, and Peter J Stuckey. Global difference constraint propagation for finite domain solvers. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 226–235, 2008. `doi:10.1145/1389449.1389478`.

19 Johannes K. Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. The silent (r)evolution of sat. *Communication of the ACM*, 66(6):64–72, May 2023. `doi:10.1145/3560469`.

20 Maarten Flippo, Konstantin Sidorov, Imko Marijnissen, Jeff Smits, and Emir Demirovic. A multi-stage proof logging framework to certify the correctness of CP solvers. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain*, volume 307 of *LIPIcs*, pages 11:1–11:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. `doi:10.4230/LIPICS.CP.2024.11`.

21 Graeme Gange, Jeremias Berg, Emir Demirović, and Peter J Stuckey. Core-guided and core-boosted search for CP. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21–24, 2020, Proceedings 17*, pages 205–221. Springer, 2020. `doi:10.1007/978-3-030-58942-4_14`.

22 Graeme Gange, Daniel Harabor, and Peter J. Stuckey. Lazy CBS: Implicit conflict-based search using lazy clause generation. In Nir Lipovetzky, Eva Onaindia, and David Smith, editors, *Proceedings of the 29th International Conference on Automated Planning and Scheduling*, pages 155–162. AAAI Press, 2019. URL: `https://www.aaai.org/ojs/index.php/ICAPS/article/view/3471`.

23 Alexey Ignatiev, Zi Li Tan, and Christos Karamanos. Towards universally accessible sat technology. In *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, pages 16–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

24 Matti Järvisalo, Marijn JH Heule, and Armin Biere. Inprocessing rules. In *International Joint Conference on Automated Reasoning*, pages 355–370. Springer, 2012.

25 Benjamin Kiesl, Marijn J. H. Heule, and Armin Biere. Truth assignments as conditional autarkies. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*, volume 11781 of *Lecture Notes in Computer Science*, pages 48–64. Springer, 2019. doi:10.1007/978-3-030-31784-3_3.

26 Inês Lynce and Joao Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *Proceedings. 15th IEEE International Conference on Tools with Artificial Intelligence*, pages 105–110. IEEE, 2003.

27 Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021. doi:10.3233/FAIA200987.

28 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM, 2001. doi:10.1145/378239.379017.

29 Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 111–121, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-94144-8_7.

30 O. Ohrimenko, P.J. Stuckey, and M. Codish. Propagation = lazy clause generation. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 544–558. Springer-Verlag, 2007. doi:10.1007/978-3-540-74970-7_39.

31 O. Ohrimenko, P.J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009. doi:10.1007/S10601-008-9064-X.

32 Laurent Perron and Frédéric Didier. CP-SAT. URL: https://developers.google.com/optimization/cp/cp_solver/.

33 Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Trans. Program. Lang. Syst.*, 31(1):2:1–2:43, 2008. doi:10.1145/1452044.1452046.

34 Andreas Schutt, Thibaut Feydy, Peter J Stuckey, and Mark G Wallace. Explaining the cumulative propagator. *Constraints*, 16:250–282, 2011. doi:10.1007/S10601-010-9103-2.

35 Peter J Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15:307–316, 2010. doi:10.1007/S10601-010-9093-0.

A Additional Results for Solver Comparison

Table 5 compares the PAR2 scores for different solver configurations on individual problems from the MiniZinc Challenge 2024.

B Additional Results for Adaptive Propagation

Table 6 presents a comparison of solving times for different adaptive propagation mechanisms across instances from the *cable-tree-wiring*, *community-detection*, and *train-scheduling* problems. Only instances solved by at least one configuration are included. The entry "t.o." denotes cases where the instance exceeded the time limit.

Table 5 Breakdown of PAR2 scores, with the smallest PAR2 score for each row highlighted.

Problem	SAT			Fixed		
	CP-SAT	Chuffed	Huub	CP-SAT	Chuffed	Huub
accap	**4382.25**	8452.66	7559.29	18002.19	18005.07	18003.38
aircraft-disassembly	10830.06	14419.04	11061.13	**10827.55**	10837.57	11050.72
cable-tree-wiring	8103.01	4317.21	**858.01**	18010.37	18011.34	18009.67
community-detection	**346.26**	1920.21	10950.04	7350.97	7827.96	11702.54
compression	4328.04	7213.36	**3719.28**	4042.27	7202.23	4741.58
concert-hall-cap	**14487.89**	14504.51	14761.80	18006.69	18009.45	18008.94
fox-geese-corn	14694.75	11625.72	**3843.77**	18001.89	18002.43	18001.61
graph-clear	7349.13	7841.57	7316.19	7250.30	**7224.61**	7337.01
harmony	7265.72	3796.52	**1048.57**	11143.52	7497.17	7795.56
hoist-benchmark	7619.59	4435.85	**4054.75**	18013.18	18017.04	18014.37
monitor-placement	5601.80	**5582.54**	5649.02	5601.50	11014.31	11003.24
neighbours	11802.65	**11068.51**	11781.09	14402.04	11638.80	14403.29
network_50_cstr	182.20	18017.16	424.07	**177.71**	7379.87	577.85
peacable_queens	14507.04	14412.62	14229.03	14476.76	**14411.15**	14484.60
portal	**7542.36**	8384.32	7585.00	14427.55	8089.08	8623.93
tiny-cvrp	**11024.93**	11259.59	14406.99	14406.67	14725.86	18003.81
train-scheduling	4713.67	**4241.32**	5377.87	14412.48	10910.69	14414.13
triangular	**14431.54**	14487.15	14614.11	14438.36	14440.20	14603.16
word-equations	3642.49	3639.17	**274.30**	3771.83	10813.68	3943.58
yumi-dynamic	**10977.15**	14875.93	18098.88	14646.59	18016.73	18101.33

Table 6 Solving times for instances in *cable-tree-wiring*, *community-detection*, and *train-scheduling*.

Problem	Instance	With All	Adaptive Engine	$T = 10^{-2}$	$T = 10^{-4}$	$T = 10^{-6}$
cable-tree-wiring	A022	100.59	t.o.	65.83	53.17	**47.88**
	A033	181.37	t.o.	188.59	240.81	**150.98**
	A041	11.56	t.o.	13.25	**7.21**	7.98
	A046	60.05	t.o.	56.81	51.47	**35.90**
	R193	451.48	t.o.	385.63	**258.37**	579.40
community-detection	adjourn.s200.k3	65.24	t.o.	55.67	**50.53**	51.24
	Polblogs.s2480.k2	77.45	t.o.	41.91	**37.56**	37.78
	Zakhary.s12.k3	**806.09**	t.o.	1071.65	1036.39	903.26
train-scheduling	trains09	932.35	t.o.	865.73	**547.81**	893.91
	trains12	29.83	21.37	24.31	**13.34**	20.44
	trains15	**7.02**	57.23	9.79	7.33	7.39
	trains18	665.75	t.o.	**541.34**	622.26	569.59

Scheduling Telescope Observations for the European Southern Observatory

Michael Prümm
European Southern Observatory, Garching, Germany

Peter Nightingale
Department of Computer Science, University of York, UK

Felix Ulrich-Oltean
Department of Computer Science, University of York, UK

Abstract

The European Southern Observatory (ESO) provides state-of-the-art large telescope facilities at three sites in Chile, supported by 16 European member states. Astronomers submit proposals for sets of observations which are reviewed and ranked based on scientific merit, then a schedule is constructed respecting the ranking and aiming to make the fullest use of the various telescopes and numerous instruments. Currently a schedule covers six months, but in the near future ESO will switch to annual schedules.

Here we examine the most challenging scheduling problem encountered by ESO: scheduling the operations of the Very Large Telescope Interferometer (VLTI) on Paranal, Chile. Tasks to be scheduled include observations performed by ESO staff, "visitor mode" periods where astronomers visit the site to use the telescopes, various maintenance tasks, and reconfiguration tasks taking multiple days. Typically a VLTI six-month schedule would contain approximately 450 activities. We explore global constraint models and a SAT encoding of the problem.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Planning and scheduling

Keywords and phrases Modelling, Constraint Programming, Scheduling, SAT, Global Constraints

Digital Object Identifier 10.4230/LIPIcs.CP.2025.43

Category Short Paper

Supplementary Material
Software: `https://doi.org/10.15124/30f68e66-acf8-44a2-ae49-54b79381c498` [10]

Funding *Peter Nightingale*: EPSRC grant EP/W001977/1.
Felix Ulrich-Oltean: EPSRC grant EP/W001977/1.

1 Introduction

The European Southern Observatory (ESO) provides a state-of-the-art facility at its site on Paranal, Chile, hosting a variety of instruments (see Figure 1). Astronomers submit proposals for sets of observations on instruments – these proposals are then ranked based on their scientific merit. A schedule is constructed to accommodate the observations from as many proposals as possible, respecting the ranking. This scheduling task is particularly challenging for the Very Large Telescope Interferometer (VLTI). The VLTI combines light from four 8.2 metre Unit Telescopes (UTs) and four movable 1.8 metre Auxiliary Telescopes (ATs) to create a virtual telescope equivalent to the distance between the individual telescopes (called the *baseline* distance). When scheduling observations on the VLTI, the time taken to reconfigure the baseline must be considered, as well as the need to run various maintenance tasks on individual telescopes making up the array.

© Michael Prümm, Peter Nightingale, and Felix Ulrich-Oltean;
licensed under Creative Commons License CC-BY 4.0

Figure 1 The instruments on Paranal, Chile. The four larger structures are the Unit Telescopes (UTs) housing various instruments. The four smaller structures are the movable Auxiliary Telescopes (ATs). The image shows light from each AT being directed through tunnels to the central VLTI laboratory, which is housed underneath the central building. Light may also be directed from the UTs to the central VLTI equipment. Credit: ESO/L. Calçada (eso.org).

ESO uses Savile Row [8] and several of its backends to create a prototype schedule for half a year of observing time using a ranked list of science proposals as input. For each proposal, we try to allocate time on the used telescope taking into account the observability of the targets and any time constraints from the astronomer and also trying to take into account probabilistic observing conditions (e.g. weather) and deterministic conditions, e.g. moon illumination. Only proposals which can fit into the schedule will get observing time. From around 900 proposals per half-year semester, roughly a quarter are accepted and scheduled.

2 Observation Scheduling Problems

ESO has two sites in Chile hosting various optical telescopes, and jointly operates a radio telescope array at a third site. Of the optical telescope sites, the older site is La Silla (in operation since the late sixties) with the 3.6m telescope and the NTT. The newer site is Paranal with the Very Large Telescope (VLT) consisting of four 8.2m Unit Telescopes (UTs) and four 1.8m Auxiliary Telescopes (ATs). The UTs can work either independently or work together to form an interferometric array (VLTI). When the UTs operate independently, the ATs can be used as VLTI, too. A telescope has several instruments. Each instrument has specific, unique capabilities and cannot easily be moved to another telescope.

Science Proposals. ESO has two Calls for Proposals per year where astronomers are invited to submit papers describing their science case and detailing the observations they wish to make. For each instrument, they list the targets they want to observe and the necessary observing conditions in *observing runs*. Observing time is allocated to individual observing runs. The proposals are reviewed for their scientific quality. Each run is assigned a *grade* and the grades are then normalised over the different instruments. The graded runs are the input for our scheduling activities. A run has the following attributes:

- A mode: Visitor Mode (VM), designated Visitor Mode (dVM), Service Mode (SM).
- Instrument and instrument setup
- Deterministic observing constraints: moon illumination ($0 \ldots 100\%$), airmass (how much atmosphere the light passes through before reaching the telescope)

- Probabilistic observing constraints: sky transparency (cloudiness: photometric, clear, thin clouds, thick clouds), turbulence category (tolerable disturbance through the air; 7 different percentages), precipitable water vapour (PWV; 0. . . 30mm)
- Optionally some time constraints, and a run grouping tag

The *observing mode* defines who makes the observations: in VM the astronomer travels to the observatory and operates the instrument herself; in dVM the remote astronomer communicates with a support astronomer at the observatory while the observations are made; and in SM, the observer prepares detailed observations in advance (*before* the observing period) and operators at the observatory carry out the observations based on current conditions. VM/dVM is *scheduled* on the calendar with a granularity of one tenth of a night, but SM is *allocated* if sufficient time under the requested observing conditions is available.

The specified airmass limits the observing window of the target. Moon illumination can be computed for each night. Sky transparency, turbulence category, and PWV have known statistical distributions. The astronomers can additionally specify time constraints such as: `between(2025-06-01, 2025-06-16)`; or `1n:0.3en (5..10) 1n:1n` (one night, followed by 0.3 nights towards the dawn, then 5 to 10 nights pause then two consecutive nights).

All runs are passed through some astronomical pre-processing that deals with all observability questions. It computes bins for the observing constraints, computes sidereal time ranges for observations, combines VM observations into compact sets of blocks for each possible start date and assigns an initial run grouping tag.

Technical Time Runs. Technical Time Runs (TTRs) are used for maintenance of the telescopes or instruments (mirror re-coating, repairs, etc) or commissioning of new instruments. TTRs have a grade like science runs and may require some telescope or instruments. They can happen simultaneously if they do not need the same facilities. The time of TTRs cannot be used for science and TTRs are usually scheduled on the calendar like VM runs.

2.1 General Approach

Currently, we try to produce a schedule for each telescope plus the VLTI using different recipes involving one or more Essence Prime [7] models. The key requirement is to schedule runs according to their ranking. It is impractical to include all runs in one optimization problem: the resulting concrete model would be too large. Instead, we solve a sequence of satisfaction problems, adding runs one by one in rank order. When an instance is unsatisfiable, we discard the most recently added run and continue with the next one in rank order.

The La Silla telescopes (3P6 and NTT) are offered only in visitor mode. Time resolution is a 1/10 of a night (a “deci night”). There can be at most 2 visitors per night. Runs with the same grouping tag must be in a few sets of contiguous nights (e.g. if an instrument needs to be mounted which is a time consuming procedure; observations should happen in appropriate blocks; these blocks are an output of the model). Once all the runs that fit are known, we enable a few more constraints and try to optimise groups and runs for compactness.

On the UTs on Paranal, we have both VM and SM runs. The VM runs are scheduled similarly to the La Silla runs with one model. The SM runs are handled differently: all their observations take time from the resource pool bin of each requested observing constraint. The run is “ in” only if sufficient time in the requested bin or bins with “ better” conditions of *all* observing constraints is available for *all* observations in the run. This requires computing the available times in all resource bins based on which VM runs/TTRs are already scheduled

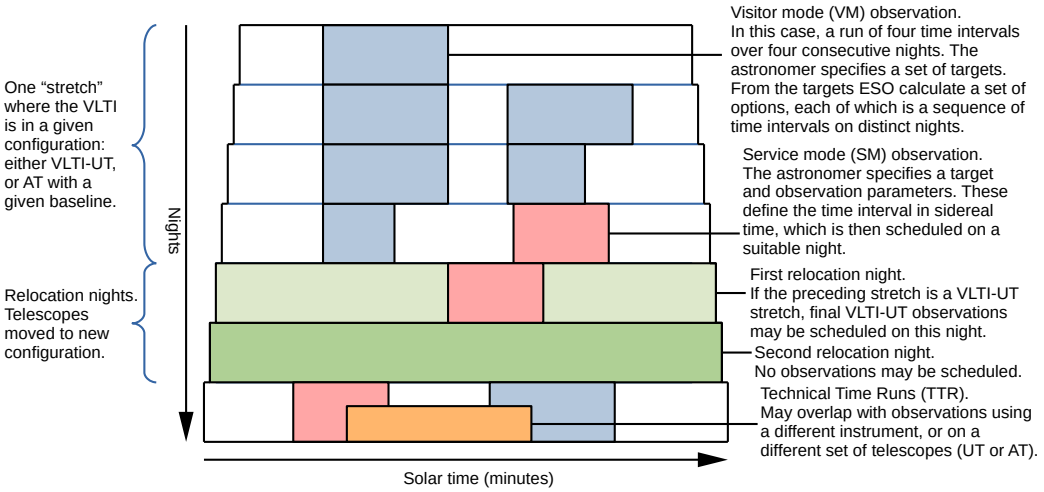


Figure 2 Illustration of VLTI scheduling showing the types of tasks and how they may overlap in time. Nights vary in length throughout the year, we have shown the nights lengthening.

(since they also use time from the bins). VM and SM runs use different models, so we need to re-run the SM model (with updated resource budgets) after a new VM run is found to fit. The VM run is considered scheduled only if the SM model is still satisfiable.

2.2 Related Work

Handley et al. [4] use Integer Linear Programming (ILP) to obtain an “optimal ordering of a set of targets over a night subject to timing constraints and time-dependent slew overheads”. The problem is solved over 10 minute intervals with up to 100 targets. Naghib et al. [6] use a Markovian approach to automate schedule generation for observations lasting around 30 seconds and lens changes around 2 minutes. In a problem closer to ours, Lampoudi et al. also implement a solution based on ILP for Las Cumbres Observatory Global Telescope (LCOGT) [5] where requests have different priorities, durations, and time windows and can be fulfilled by one of a number of instruments. They report being able to select 2055 of 3864 submissions over a half-year semester. The time granularity is a hyperparameter and the exact time unit is unclear. Spohn [12] uses OR-Tools’ CP-SAT solver to schedule observations of desired targets; the *Optional Interval Variable* type is used for the no-overlap constraints.

The context for our work is described in [11], detailing the entire long-term scheduling challenge for ESO at La Silla and Paranal. Our work here specifically addresses scheduling the VLTI which presents its own unique challenges. The VLT and VLTI have a broad mix of service mode and visitor mode observations, where other observatories are mostly either all service mode or all visitor mode. In addition, there are many dependencies between observations on the VLT and VLTI that must be taken into account making the problem harder than having fewer telescopes without such dependencies.

3 VLTI Scheduling Problem

VLTI scheduling is the most difficult case (illustrated in Figure 2). Here, all observations carry a grouping tag, specifying the size of the *baseline*: small, medium, large, extended (all of these use only the ATs; VLTI-AT) or UTs (only the UTs; VLTI-UT). The baseline is the set of physical locations of the ATs. Changing the baseline takes 1 or 2 nights (depending on

the two baselines) so observations are grouped accordingly and there must be sufficient time between them. The VLTI schedule must also take all the TTRs into account that need one of the UTs, because during these times one cannot have VLTI-UT observations.

Once the VLTI has been set up with a baseline, it remains in that configuration for a set of nights (named a *stretch*). Stretches have a minimum length because it is not practical to spend one or two nights setting up a new baseline then using it for a single night. Stretches also have an upper limit designed to avoid making the UTs unavailable for long periods of time and thus blocking other observations (that are not in the VLTI schedule) on the UTs.

There are two periods called *imaging slots* (given as input) when no VLTI-UT observations occur, and which contain at least one set of nights of each of the VLTI-AT baselines. To find the baseline stretches, we attempt to put all observations somewhere on the calendar.

4 Modelling the VLTI Scheduling Problem

In this section we describe the most important aspects of each of the constraint models. Full models and all benchmark instances are provided in the experimental repository [10].

4.1 Basic Constraint Model

The decision variables are as follows:

- The night sn_i, start time ss_i (sidereal time in minutes), and end time se_i of each SM task i (where a task is part of a run).
- Equivalent variables for VM and TTR tasks: vn_i, vs_i, ve_i, tn_i, ts_i, te_i.
- The first and last nights of each TTR r: $twhen_r$ and $tlast_r$.
- The first and last nights of each VM run r: $when_r$ and $nlast_r$.
- The first and last days of each daytime TTR r: $dwhen_r$ and $dlast_r$.
- The first night of each stretch: $gbeg_{g,s}$, and the first night after the stretch ends: $gend_{g,s}$ for a group g and stretch s.

The domain of the variables representing sidereal time in minutes (e.g. vs_i) is an interval from the earliest starting time to the latest end time of a night in the schedule. For VM and TTR tasks, the value 0 is added as a dummy value to allow for optional tasks: some runs do not have a fixed number of tasks, and any excess ones are given a start and end time of 0.

Baseline Stretch Constraints. Each group g represents a baseline (described in Section 3), and a stretch s of g is a set of nights when the telescopes are in the baseline of g. Stretches have a minimum and maximum length (constant for each baseline) which constrain the difference between $gend_{g,s}$ and $gbeg_{g,s}$. Within one group, any two stretches are separated by at least 4 nights to allow another stretch in between. Stretches cannot overlap, and are separated by relocation nights (illustrated in Figure 2) when ATs are moved:

$$(gend_{g_1,i} + r_{g_1,g_2} \leq gbeg_{g_2,j}) \vee (gend_{g_2,j} + r_{g_2,g_1} \leq gbeg_{g_1,i})$$

for each pair of distinct groups g_1, g_2 and stretches i, j, with r being the number of relocation nights. Within a group, stretches are constrained to be in order. Finally, a set of predefined stretches (specified by group, first night, and last night) may be given as input. These may be assigned to any *gbeg* and *gend* pair in the group as needed to satisfy the ordering constraints.

Tasks. For each VM run (encompassing a set of VM tasks), a set of options is precomputed and passed in as a parameter. Each option specifies the exact night, start time and end time of all the tasks in the run. A table constraint is used to ensure the relevant task variables (vn_i, vs_i, and ve_i for all i in the run, and $nlast_r$ for run r) take values corresponding to one of the options. VM runs are associated with a group g, and vn_i for each task in the run is constrained to be within a stretch of g (i.e. there exists a stretch that contains vn_i). TTRs are handled similarly but with a different stretch constraint: TTRs are excluded from the night before each stretch begins. Daytime TTRs are handled similarly to TTRs but only the start and end dates are needed in this case and the relevant variables are $dwhen_r$ and $dlast_r$.

SM runs are handled differently. For each run r, the parameters are the number of tasks in the run, the task duration, a time interval in sidereal time within which the tasks must be scheduled (on any suitable night), and a group. For each task i within the SM run r, the start time ss_i is constrained such that the task will start and finish within the given time interval; se_i is simply ss_i plus the given duration; and sn_i must be within any stretch of the given group (with the same constraint as for VMs).

Each task must fit within the night on which it is scheduled. The times of the start and end of each night are given as parameters *nstart* and *nend*, and for an SM task we have the constraints $nstart[sn_i] \leq ss_i \wedge se_i \leq nend[sn_i]$. The options for times of VM runs and TTRs are precalculated so their times will be within the relevant night.

Non-Overlap Constraints. A key constraint is the non-overlap of tasks. For a pair of SM tasks i, j, the constraint is as follows:

$$sn_i \neq sn_j \ \vee \ se_i \leq ss_j \ \vee \ se_j \leq ss_i$$

For each pair of (SM or VM) tasks i, j, non-overlap constraints are posted if: i is an SM task and j is a VM task; i and j are SM tasks; or i and j belong to different VM runs. Within an SM run, some observations may be repeated. Tasks corresponding to repeated observations are also constrained to be in order. Within a VM run, the precalculated options for nights and times are non-overlapping so no constraint is required. Non-overlap constraints are also posted between TTRs when they are using the same telescope setup (any UT, all UTs, or all ATs) or the same instrument. UT TTRs are not allowed to overlap with any VLTI UT observations (because VLTI uses all four UTs), and the same is true for ATs. Other overlaps of TTRs with SM or VM observations are allowed.

A few instances have reserved time intervals on a specific night where no SM or VM tasks may run. For example, at the beginning of a 6-month scheduling period, some time may be reserved for installing software updates on one of the UTs. Non-overlap constraints are posted to exclude VM and SM tasks from reserved times.

Further Constraints. There are a number of other constraints arising from operational policy or astronomical reasons. At most two VM observations are allowed on one night (with the exception of some VM observations that are exempted from this rule). The reason is that the control room on Paranal would become too busy with more than two visiting astronomers. The rule is implemented with an *atmost* constraint on the vn variables.

Another group of constraints concern imaging slots (described in Section 3; a fixed sequence of nights typically 3-5 weeks long). No VM tasks are allowed within imaging slots (a policy of the observatory); implemented as a unary constraint on each vn variable. For each imaging slot i, and for each AT baseline group g, there must exist a stretch of the baseline somewhere within the imaging slot; the constraint is written $\exists j : istart_i \leq gbeg_{g,j} \wedge gend_{g,j} \leq iend_i$

where $istart_i$ and $iend_i$ are the bounds of the imaging slot. Also, AT stretches must be completely outside or completely within an imaging slot, implemented as a logical combination of unary constraints on *gbeg* and *gend* variables.

There are some restrictions on the nights that activities can happen, implemented with unary constraints: the night before each stretch cannot have reserved time intervals; and VLTI UT stretches cannot be within imaging slots (because imaging slots exclusively use the ATs). Relocation nights before AT stretches must not contain TTR tasks on the ATs (implemented with $\neq$ constraints between a constant shifted *gbeg* variable and a *tn* variable).

Finally, a pair of (VM, TTR, or daytime TTR) runs a, b may have an interrun constraint that places them in a fixed order (e.g. a goes before b) and specifies a minimum and maximum separation (in nights) between the runs. Interrun constraints are implemented with $\leq$ or $<$ constraints between *vn*, *tn*, or *dwhen* variables (with some shifted by a constant).

4.2 Global Constraint Models

The VLTI problem has a set of tasks (observations, TTRs, etc) that either cannot overlap or can only overlap in restricted ways. In addition there are complex constraints on the order and timing of tasks. While we cannot capture the whole problem in a single global constraint, it is possible to capture many of the non-overlap constraints in one global constraint, reducing the size (in terms of number of constraints) and complexity of the model.

One approach would be to represent the start time of each task with a single variable (as opposed to two variables for the night and the sidereal time in minutes) and use a Disjunctive scheduling constraint [1] to capture non-overlap of VLTI observations. This would allow many of the pairwise non-overlap constraints to be removed but would create variables with very large domains (time in minutes, spanning the nights of half a year or a full year).

We decided to use a 2D packing constraint as an alternative to standard scheduling constraints. DiffN [2] prevents overlap of rectangles on a plane. Each rectangle is described by four integer variables: the x and y coordinates of one corner, and dimensions dx and dy. DiffN seems particularly appropriate here because it directly captures the fact that tasks cannot be split over two nights, and it avoids the need for variables with very large domains.

DiffN Model 1 is adapted from the basic constraint model with the following changes. Duration variables have been added for VM tasks, and connected to the night, start time and end time variables with a table constraint (recall that the possible nights and times of VM tasks are pre-calculated and provided in a table). A single DiffN constraint has been added, with the x dimension representing the night and the y dimension representing the sidereal time in minutes. All SM and VM tasks are represented as a rectangle of size $(1, dy)$ where dy is the SM or VM duration variable as appropriate. No other tasks are included in the DiffN. Finally, non-overlap constraints between VM and SM tasks are deleted, removing a quadratic number of primitive constraints.

In the first DiffN model the global constraint is quite loose because it does not include any maintenance or relocation tasks. **DiffN Model 2** is an attempt to strengthen propagation by adding more tasks to the DiffN constraint. Recall that for each relocation, the last night (or only night) of the relocation period cannot overlap with any VM or SM tasks. We use the existing variables representing the first night of each stretch (shifted by -1) to define rectangles of size $(1, dy)$ (where dy is the duration of the longest night in the schedule) and these are added to the DiffN constraint. Otherwise the model is identical to DiffN model 1.

4.3 SAT Encoding

The basic model is used when solving with SAT, and it is encoded as described in the Savile Row manual [7]. The order or direct encoding (or both) is chosen for each integer variable according to the constraints posted on the variable; constraints are encoded with the default encodings described in the manual; and some variables (notably all se_i) are removed automatically because they are a linear mapping of another variable.

5 Experimental Evaluation

We carry out experiments to compare performance across different combinations of the models presented and backend solvers. The models are written in the Essence Prime language [7] which are read by the modelling assistant tool Savile Row [8]. Savile Row is able to tailor models to a variety of back-end solvers, including OR-Tools and SAT. We chose OR-Tools CP-SAT [9] 9.11.4210 as it has been the best all-rounder in recent constraint solving competitions [13] and it implements the DiffN global constraint. We also solve the problem with Savile Row's SAT output, using Kissat [3] 3.1.1, because in preliminary experiments this approach was usually the fastest in a single-threaded context. We provide the models, instances and software in an online repository [10].

As described in Section 2.1, each solver call is independent, solving the entire instance (i.e. scheduling a concrete set of runs, or finding that no such schedule exists). The approach may seem inefficient but it has two advantages: we do not need to represent all the proposed runs in one constraint optimisation problem (which would be very large and difficult to handle); and each instance can be optimised prior to solving by Savile Row (with a concrete set of runs, i.e. without complications such as optional tasks or reification of constraints between tasks that might hinder pre-solving). Throughout this section we use *instance* to mean one CSP instance, corresponding to one solver call and one point on the plots below (Figure 3). The instances in the repository represent the progression towards the final schedule, with instances containing incrementally more requested runs to schedule.

The experiments are run on a server with Dual AMD EPYC 7501 2.6GHz processors with 2×64 threads and 2TB RAM, running Ubuntu Linux 22.04.5. To account for randomness in the solvers, each instance is solved 5 times with different random seeds and the median total runtime recorded. We run 231 real half-year instances from ESO with these setups:

- DiffN Model 1 as described in Section 4.2, solved with OR-Tools CP-SAT single-threaded with free search;
- DiffN Model 2 as described in Section 4.2, solved with OR-Tools (as above);
- The basic constraint model, solved with OR-Tools (as above); and
- The basic constraint model encoded by Savile Row into SAT, solved with Kissat.

We also solve 110 synthetic full-year instances (including real science runs and TTRs from two successive half-years, preprocessed as a full year), using the best performing model and solver combinations for the half-year experiment, namely DiffN 1 and SAT. With the full-year instances we found that the relative performance of OR-Tools (single-threaded, free search) was poor for the most challenging instances so we also ran OR-Tools with 16 threads and without free search (because the free search flag causes it to ignore the specified number of threads). In this case we report the total CPU time across all threads.

The experimental results are plotted in Figure 3. We observe that the DiffN 1 model is slightly faster than DiffN 2 on the majority of instances, with an overall geometric mean speedup of 1.38. DiffN 1 also does marginally better than SAT, with an overall speed-up

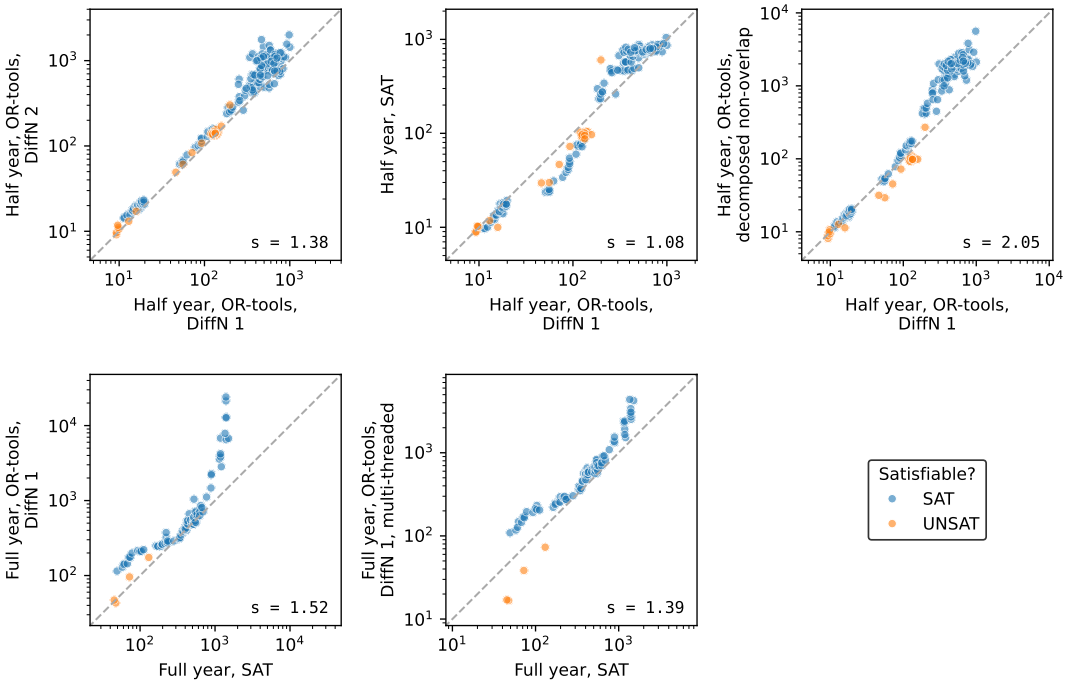


Figure 3 Experimental results. We compare total solving time (Savile Row and backend solver) in seconds for pairs of setups. Each plot displays the geometric mean speed-up s of the x-axis setup compared to the setup on the y-axis. Top: half-year schedules. Bottom: full year schedules.

of 1.08; the advantage is more marked in the harder instances, whereas SAT is solving the easy and medium difficulty instances more quickly. The third plot shows that the global constraint model does outperform the one with decomposed non-overlap constraints, being over twice as fast on average. In terms of the full-year instances, single-threaded OR-Tools is struggling to solve the harder instances, with SAT being 1.52 times faster on average. When running OR-Tools with 16 threads, the trend with the most challenging instances is curving away from the $x = y$ line, so it appears that SAT is scaling better overall.

There is a larger proportion of unsatisfiable instances in the half-year schedule; this is explained in part by the fact that some requested targets are barely visible during the half-year period, but there must be a set of nights when they are visible during the full year.

6 Conclusions

ESO provides state-of-the-art large telescope facilities which astronomers bid to use. They are significantly over-subscribed, leading to scheduling problems where the goal is to make the fullest possible use of the facilities while respecting priority order (based on scientific merit) on the bids. The most challenging scheduling problems arise with the VLTI facility on Paranal, Chile, where there are complex constraints between observation tasks and other tasks, and hundreds of tasks to schedule per half-year. We have presented CP models of the VLTI problem and evaluated their performance with a CP solver and a SAT solver.

The new scheduling approach replaces a laborious and partially manual process with a largely automated one, saving significant time and potentially also allowing more observations to be packed into the schedule. The new system has been in production use since the end of 2023, during which time the CP models have evolved to their current state. Future

work includes refining the CP models and SAT encoding, moving to full-year scheduling, and a feasibility study of scheduling more types of observations in one model (for example, individual UT observations combined with VLTI).

References

1 Philippe Baptiste and Claude Le Pape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *Proceedings of IJCAI-95*, pages 600–606, 1995. URL: http://ijcai.org/Proceedings/95-1/Papers/079.pdf.

2 Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994.

3 Armin Biere and Mathias Fleury. Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*, pages 10–11. University of Helsinki, 2022. URL: https://hdl.handle.net/10138/359079.

4 Luke B. Handley, Erik A. Petigura, and Velibor V. Mišić. Solving the Traveling Telescope Problem with Mixed-integer Linear Programming. *The Astronomical Journal*, 167(1):33, December 2023. doi:10.3847/1538-3881/ad0dfb.

5 Sotiria Lampoudi, Eric Saunders, and Jason Eastman. An Integer Linear Programming Solution to the Telescope Network Scheduling Problem, March 2015. doi:10.48550/arXiv.1503.07170.

6 Elahesadat Naghib, Peter Yoachim, Robert J. Vanderbei, Andrew J. Connolly, and R. Lynne Jones. A Framework for Telescope Schedulers: With Applications to the Large Synoptic Survey Telescope. *The Astronomical Journal*, 157(4):151, March 2019. doi:10.3847/1538-3881/aafece.

7 Peter Nightingale. Savile Row manual, 2021. doi:10.48550/arXiv.2201.03472.

8 Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, 2017. doi:10.1016/j.artint.2017.07.001.

9 Laurent Perron and Frédéric Didier. OR-Tools CP-SAT 9.11.4210, 2024. URL: https://developers.google.com/optimization/cp/cp_solver/.

10 Michael Prümm, Peter Nightingale, and Felix Ulrich-Oltean. Dataset for: Scheduling telescope observations for the European Southern Observatory. doi:10.15124/30f68e66-acf8-44a2-ae49-54b79381c498.

11 M. Rejkuba, O. R. Hainaut, T. Bierwirth, M. Pruemm, and A. Weiss. Time Allocation and Long-Term Scheduling of ESO Telescopes at La Silla Paranal Observatory, July 2024. doi:10.48550/arXiv.2407.15470.

12 Corey Spohn. *Planning Direct Imaging Observations of Exoplanets with Precursor Data*. Phd thesis, Cornell University, August 2023. doi:10.7298/hvz9-7325.

13 The MiniZinc Team. MiniZinc - Challenge 2024 Results. URL: https://www.minizinc.org/challenge/2024/results/.