30th International Conference on Principles and Practice of Constraint Programming

CP 2024, September 2–6, 2024, Girona, Spain

Edited by

Paul Shaw

LIPICS

Editors

Paul Shaw
IBM, Biot, France
paul.shaw@fr.ibm.com

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

ISBN 978-3-95977-336-2

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-336-2.

Publication date
August, 2024

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

Digital Object Identifier: 10.4230/LIPIcs.CP.2024.0

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.

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Invited Talks

Regular Papers

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Short Papers

Preface

The year 2024 marks the 30th edition of the International Conference on Principles and Practice of Constraint Programming. Whilst I did not attend the very first couple of conferences (1995 in Cassis, and 1996 in Cambridge), I did go to Linz in 1997 and managed to get a paper in the program in 1998 in Pisa: my CP journey had begun. This year, the conference takes place in Girona, Spain, from the 2nd to the 6th of September. More details can be found at `https://cp2024.a4cp.org/index.html`.

This year in Girona, the conference has 38 accepted papers (from 95 submissions), 2 invited talks and 3 tutorials over 5 days. As is customary, day zero of the conference comprises a series of workshops and the doctoral program: a time for young researchers to come together to exchange ideas. This year, four workshops are represented: The 26th International Workshop on Configuration, the 23rd Workshop on Constraint Modelling and Reformulation, the 7th Workshop on Progress Towards the Holy Grail, and the 1st Workshop on Discrete Optimization with Soft Constraints. Special thanks go to Edward Lam for chairing the Doctoral Program, and to Carlos Ansótegui for chairing the Workshops Program.

The conference has three diverse tutorials: Domain-Independent Dynamic Programming by Chris Beck and Ryo Kuroiwa, JuMP and Constraint Programming by Benoît Legat, and Constraint Acquisition by Dimos Tsouros. I want to thank all the authors and presenters for their willingness to contribute to the conference in this way, and to Pierre Schaus for agreeing to be tutorial chair and bringing together these talks.

As usual, we also have a number of special tracks this year: Applications, CP and Machine Learning, and CP and Quantum Computing. I'd like to thank Louis-Martin Rousseau, Quentin Cappart and Philippe Codognet who did an excellent job of chairing these tracks.

I would like to welcome Özgür Akgün to the role of DEI chair this year, a term that he will hold for two years. He joins María Andreína Francisco Rodríguez who is in her second year of her role as joint DEI chair.

A conference like CP relies on many people to make it a success. In terms of the program itself, I would like to thank first of all everybody who contributed papers: submissions are the essence of the conference. Then, of course, I would like to thank the program committee, additional reviewers, and the senior program committee who made the final deliberations. I found all the PC and SPC discussions to be very fair and professional. I came away from the process with a really positive view of our community. Thank you for that.

I am very much looking forward to the two invited talks this year, both from previous program chairs of CP. Ian Gent will speak about "Solving Patience and Solitaire Games with Good Old-Fashioned AI", and Francesca Rossi will discuss "Thinking Fast and Slow in AI: A cognitive architecture to augment both AI and human reasoning". Another special treat will be a panel discussion between previous program chairs of CP. Many thanks to Eugene Freuder for arranging this special 30th anniversary session.

Of course, the conference organization was fully dependent upon our excellent conference chairs, Miquel Bofill and Mateu Villaret and our publicity chair, Jordi Coll. Their hard work on the organization made my job so much simpler and allowed me to focus on the program. I am writing this two months before the conference itself, but I am sure the venue and organization on the week of the conference are going to be fabulous.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Finally, I wish to thank our generous sponsors: The Association for Constraint Programming, the AI Journal, ScheduleOpt, Universitat de Girona, Generalitat de Catalunya, Diputació de Girona, Mitsubishi Electric, IBM, Google, The Optimization Firm, Huawei, Cosling, Potassco Solutions, Ajuntament de Girona, EurAI, Patronat Politècnica (Universitat de Girona), Càtedra d'Informació i Computació (Universitat de Girona), Càtedra Lluís A. Santaló d'Aplicacions de la Matemàtica (Universitat de Girona).

July 2024 Paul Shaw

List of Authors

Özgür Akgün (38)
School of Computer Science,
University of St Andrews, UK

Markus Anders (3)
TU Darmstadt, Germany

J. Christopher Beck (12, 32)
Department of Mechanical and Industrial Engineering, University of Toronto, Canada

Jeremias Berg (4)
Department of Computer Science, HIIT, Helsinki, Finland;
University of Helsinki, Finland

Armin Biere (13)
University of Freiburg, Germany

Ignace Bleukx (29)
DTAI, KU Leuven, Belgium

Bart Bogaerts (4)
Vrije Universiteit Brussel, Belgium

Alexandre Bonlarron (25)
Université Côte d'Azur, Inria, Sophia Antipolis, France;
Université Côte d'Azur, I3S, CNRS, Sophia Antipolis, France

Sofia Brenner (3)
TU Darmstadt, Germany

Laure Brisoux-Devendeville (34)
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Shaowei Cai (5, 19)
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

Quentin Cappart (22, 30)
Polytechnique Montréal, Canada

Anouck Chan (18)
DTIS, ONERA, Université de Toulouse, France

Xiamin Chen (6)
Shanghai University of Finance and Economics, China

Zhihan Chen (5)
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

Sami Cherif (34)
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Samuel Cloutier (7)
Université Laval, Québec, Canada

Amanda Coles (12)
Department of Informatics, King's College London, UK

João Cortes (8)
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Kevin Dalmeijer (16)
H. Milton Stewart School of Ind. and Syst. Engineering, Georgia Institute of Technology, Atlanta, GA, USA

Nguyen Dang (38)
School of Computer Science,
University of St Andrews, UK

Elisabetta De Maria (25)
Université Côte d'Azur, I3S, CNRS, Sophia Antipolis, France

Jorke M. de Vlas (35)
Linköping Universitet, Sweden

Augustin Delecluse (36)
TRAIL, ICTEAM, UCLouvain, Belgium

Emir Demirović (9, 11)
TU Delft, The Netherlands

Jo Devriendt (29)
DTAI, KU Leuven, Belgium

Bistra Dilkina (22)
Center for Artificial Intelligence in Society, University of Southern California, Los Angeles, CA, USA

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Alexandre Dubray (10)
Institute of Information and Communication Technologies, Electonics and Applied Mathematics (ICTEAM), UCLouvain, Belgium

Aaron Ferber (22)
Center for Artificial Intelligence in Society, University of Southern California, Los Angeles, CA, USA

Maarten Flippo (11)
Delft University of Technology, The Netherlands

Adam Francis Green (12)
Department of Informatics,
King's College London, UK;
Tango Hospitality Inc., Toronto, Canada

Cunjing Ge (13)
National Key Laboratory for Novel Software Technology, Nanjing University, China;
School of Artificial Intelligence, Nanjing University, China

Ian P. Gent (1)
School of Computer Science,
University of St Andrews, UK

Diarmuid Grimes (39)
Munster Technological University, Cork, Ireland;
SFI Centre for Research Training in Artificial Intelligence, Cork, Ireland

Tias Guns (20, 29)
DTAI, KU Leuven, Belgium

Hao Hu (5)
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

Panteleimon Iosif (14)
University of Western Macedonia, Kozani, Greece

Peter Jonsson (15)
Department of Computer and Information Science, Linköping University, Sweden

Jihye Jung (16)
H. Milton Stewart School of Ind. and Syst. Engineering, Georgia Institute of Technology, Atlanta, GA, USA

Artem Kaznatcheev (17)
Department of Mathematics, and Department of Information and Computing Sciences, Utrecht University, The Netherlands

Markus Kirchweger (37)
Algorithms and Complexity Group, TU Wien, Austria

Erdem Kuş (38)
School of Computer Science,
University of St Andrews, UK

Victor Lagerkvist (15)
Department of Computer and Information Science, Linköping University, Sweden

Duc Anh Le (18)
DTIS, ONERA, Université de Toulouse, France

Christophe Lecoutre (18)
CRIL, Université d'Artois & CNRS, France

Zhendong Lei (6)
Huawei Taylor Lab, Shanghai, China

Chu-Min Li (34)
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Peng Lin (5, 19)
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

Pinyan Lu (6)
Shanghai University of Finance and Economics, Shanghai, China; Huawei Taylor Lab, Shanghai, China

Corinne Lucet (34)
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Inês Lynce (8)
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Vasco Manquinho (8)
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Imko Marijnissen (11)
Delft University of Technology, The Netherlands

Ciaran McCreesh (9)
University of Glasgow, Scotland

Matthew J. McIlree (9)
University of Glasgow, Scotland

Kostis Michailidis (20)
DTAI, KU Leuven, Belgium

Laurent Michel (28)
Synchrony Chair in Cybersecurity, School of Computing, University of Connecticut, Storrs, CT, USA

Ian Miguel (38)
School of Computer Science,
University of St Andrews, UK

Mohsen Nafar (21)
Bielefeld University, Germany

Siegfried Nijssen (10)
Institute of Information and Communication Technologies, Electonics and Applied Mathematics (ICTEAM), UCLouvain, Belgium

Jakob Nordström (4, 9)
University of Copenhagen, Denmark;
Lund University, Sweden

Barry O'Sullivan (39)
Insight SFI Research Centre for Data Analytics, National University of Ireland Galway, Ireland; SFI Centre for Research Training in Artificial Intelligence, Cork, Ireland; School of Computer Science & IT, University College Cork, Ireland

Andy Oertel (4, 9)
Lund University, Sweden;
University of Copenhagen, Denmark

George Osipov (15)
Department of Computer and Information Science, Linköping University, Sweden

Augustin Parjadis (22)
Polytechnique Montréal, Canada

Tobias Paxian (4)
University of Freiburg, Germany

Gilles Pesant (30)
Polytechnique Montréal, Canada

Nikolaos Ploskas (14)
University of Western Macedonia, Kozani, Greece

Enrico Pontelli (28)
Department of Computer Science, New Mexico State University, Las Cruces, NM, USA

Xavier Pucel (23)
ONERA, ONERA DTIS, Toulouse,
Université de Toulouse, France

Claude-Guy Quimper (7, 30)
Université Laval, Québec, Canada

Ben Rachmut (24)
Ben-Gurion University of the Negev,
Beer-Sheva, Israel

Gaurav Rattan (3)
University of Twente, Enschede,
The Netherlands

Shiraz Regev (33)
Ben-Gurion University of the Negev,
Beer-Sheva, Israel

Francesca Rossi (2)
IBM Research, Yorktown Heights, NY, USA

Louis-Martin Rousseau (22)
Polytechnique Montréal, Canada

Stéphanie Roussel (18, 23)
DTIS, ONERA, Université de Toulouse, France

Florian Régin (25)
Université Côte d'Azur, I3S, CNRS,
Sophia Antipolis, France

Jean-Charles Régin (27)
Université Côte d'Azur, CNRS, I3S,
Sophia Antipolis, France

Michael Römer (21)
Bielefeld University, Germany

Heythem Sattoutah (34)
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Pierre Schaus (10, 36)
Institute of Information and Communication Technologies, Electonics and Applied Mathematics (ICTEAM), UCLouvain, Belgium

André Schidler (26)
Algorithms and Complexity Group, TU Wien, Austria

Margaux Schmied (27)
Université Côte d'Azur, CNRS, I3S,
Sophia Antipolis, France

Konstantin Sidorov (9, 11)
TU Delft, The Netherlands

Jeff Smits (11)
Delft University of Technology, The Netherlands

Filipe Souza (39)
Insight SFI Research Centre for Data Analytics, National University of Ireland Galway, Ireland; SFI Centre for Research Training in Artificial Intelligence, Cork, Ireland; School of Computer Science & IT, University College Cork, Ireland

Kostas Stergiou (14)
University of Western Macedonia, Kozani,
Greece

Stefan Szeider (26, 37)
Algorithms and Complexity Group, TU Wien,
Austria

Fabio Tardivo (28)
Department of Computer Science, New Mexico
State University, Las Cruces, NM, USA

Dimos Tsouros (20, 29)
DTAI, KU Leuven, Belgium

Dimosthenis C. Tsouros (14)
KU Leuven, Belgium

Pascal Van Hentenryck (16)
H. Milton Stewart School of Ind. and Syst.
Engineering, Georgia Institute of Technology,
Atlanta, GA, USA

Melle van Marle (17)
Department of Mathematics, and Department of
Information and Computing Sciences, Utrecht
University, The Netherlands

Dieter Vandesande (4)
Vrije Universiteit Brussel, Belgium

Wout Vanroose (29)
DTAI, KU Leuven, Belgium

Hélène Verhaeghe (29, 30)
DTAI, KU Leuven, Belgium

William Yeoh (24, 33)
Washington University in St. Louis, MO, USA

Oleg Zaikin (31)
ISDCT SB RAS, Irkutsk, Russia

Jiachen Zhang (32)
Department of Mechanical and Industrial
Engineering, University of Toronto, Canada

Neng-Fa Zhou (40)
CUNY Brooklyn College and the Graduate
Center, NY, USA

Roie Zivan (24, 33)
Ben-Gurion University of the Negev,
Beer-Sheva, Israel

Mengchuan Zou (19)
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

Senior Program Committee

- Christian Artigues (LAAS-CNRS)
- J. Christopher Beck (University of Toronto)
- Nicolas Beldiceanu (IMT Atlantique (LS2N))
- Armin Biere (University of Freiburg)
- Quentin Cappart (Ecole Polytechnique de Montréal)
- Philippe Codognet (JFLI - CNRS / Sorbonne University / University of Tokyo)
- Pierre Flener (Uppsala University)
- Tias Guns (KU Leuven)
- Serdar Kadioglu (Brown University)
- Michele Lombardi (DISI, University of Bologna)
- Ines Lynce (INESC-ID/IST, Universidade de Lisboa)
- Ciaran McCreesh (University of Glasgow)
- Laurent Michel (University of Connecticut)
- Gilles Pesant (Polytechnique Montréal)
- Louis-Martin Rousseau (Polytechnique)
- Domenico Salvagnin (University of Padova)
- Helmut Simonis (Insight Centre for Data Analytics, School of Computer Science and Information Technology, University College Cork)
- Guido Tack (Monash University)
- Willem-Jan Van Hoeve (Carnegie Mellon University)

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committee

- Özgür Akgün (University of St Andrews)
- Alejandro Arbelaez (University College Cork)
- Gilles Audemard (CRIL)
- Nassim Belmecheri (SIMULA Research Laboratory)
- Russell Bent (Los Alamos national Laboratory)
- Jeremias Berg (University of Helsinki)
- Ken Brown (University College Cork)
- Silvia Butti (University of Oxford)
- Shaowei Cai (Institute of Software, Chinese Academy of Sciences)
- Clément Carbonnel (CNRS)
- Dingding Chen (Chongqing University)
- Berthe Y. Choueiry (University of Nebraska-Lincoln)
- Andre Augusto Cire (University of Toronto)
- Laura Climent (University College Cork (UCC))
- Claudio Contardo (Concordia University)
- Martin Cooper (IRIT - Universite Paul Sabatier)
- Timothy Curry (University of Connecticut)
- Nguyen Dang (St Andrews University)
- Simon de Givry (INRA - MIAT)
- Sophie Demassey (Mines Paris, Université PSL, Centre de Mathématiques Appliquées (CMA))
- Emir Demirović (Delft University of Technology)
- Guillaume Derval (University of Liège)
- Catherine Dubois (ENSIIE-Samovar)
- Guillaume Escamocher (University College Cork, Ireland)
- Joan Espasa Arxer (University of St Andrews)
- Andrea Formisano (Dipartimento di Scienze Matematiche, Informatiche e Fisiche, Università di Udine)
- Vijay Ganesh (Georgia Tech)
- Maria Garcia De La Banda (Monash University)
- Ian Gent (University of St Andrews)
- Priyanka Golia (IIT Delhi)
- Diarmuid Grimes (Munster Technological University)
- Stefano Gualandi (Università degli studi di Pavia)
- Djamal Habet (LIS UMR 7020, University of Aix-Marseille)
- Marijn Heule (Carnegie Mellon University)
- Amel Hidouri (LARODEC, High ISG, University of Tunis)
- Ruth Hoffmann (University of St Andrews, School of Computer Science)
- Alexey Ignatiev (Monash University)
- Said Jabbour (CRIL CNRS - Univ. Artois)
- Mikolas Janota (Czech Technical University in Prague)
- Matti Järvisalo (University of Helsinki)
- Christopher Jefferson (University of St. Andrews)
- Anthony Karahalios (Carnegie Mellon University)
- Daniela Kaufmann (TU Wien)

- Philip Kilby (Data61 and the Australian National University)
- Zeynep Kiziltan (University of Bologna)
- T. K. Satish Kumar (University of Southern California)
- Mikael Lagerkvist (zayenz.se)
- Nadjib Lazaar (University of Montpellier)
- Christophe Lecoutre (CRIL, Univ. Artois)
- Jimmy Lee (The Chinese University of Hong Kong)
- Antoine Legrain (Ecole Polytechnique Montreal)
- Olivier Lhomme (IBM France)
- Zhanshan Li (Jilin University)
- Andrea Lodi (Cornell Tech)
- Feifei Ma (Institute of Software, Chinese Academy of Sciences)
- Arnaud Malapert (Université Côte d'Azur, CNRS, I3S, France)
- Kuldeep Meel (University of Toronto)
- Ian Miguel (University of St Andrews)
- Nysret Musliu (TU Wien)
- Peter Nightingale (University of York)
- Barry O'Sullivan (University College Cork, Ireland)
- Sebastian Ordyniak (The University of Leeds)
- Cemalettin Ozturk (Raytheon Technologies, United Technologies Research Center Ireland)
- Anastasia Paparrizou (CRIL-CNRS, University of Artois)
- Justin Pearson (Uppsala University)
- Guillaume Perez (University of Nice-Sophia Antipolis / I3S)
- Laurent Perron (Google France)
- Andreas Podelski (University of Freiburg)
- Cédric Pralet (ONERA Toulouse)
- Steve Prestwich (Insight Centre for Data Analytics)
- Patrick Prosser (University of Glasgow)
- Charles Prud'Homme (IMT Atlantique, LS2N)
- Luis Quesada (Insight Centre for Data Analytics, University College Cork)
- Claude-Guy Quimper (Laval University)
- Philippe Refalo (IBM)
- Jean-Charles Regin (University Nice-Sophia Antipolis / I3S / CNRS)
- Emma Rollon (Universitat Politècnica de Catalunya)
- Stefan Ropke (Technical University of Denmark)
- Hana Rudovà (Masaryk University)
- Thomas Schiex (INRAE)
- Andreas Schutt (CSIRO)
- Ilankaikone Senthooran (Monash University)
- Mohamed Siala (INSA Toulouse & LAAS-CNRS)
- Laurent Simon (Labri, Bordeaux Institute of Technology)
- Christine Solnon (INSA Lyon)
- Kostas Stergiou (University of Western Macedonia)
- Peter J. Stuckey (Monash University)
- Stefan Szeider (TU Wien)
- Cyril Terrioux (LIS - UMR CNRS 7020 - Aix-Marseille Université)
- Kevin Tierney (Bielefeld University)
- Michael Trick (Carnegie Mellon University)

- Gilles Trombettoni (LIRMM, University of Montpellier)
- Dimosthenis C. Tsouros (KU Leuven)
- Felix Ulrich-Oltean (University of York)
- Elise Vareilles (ISAE SUPAERO Toulouse, France)
- Hélène Verhaeghe (KU Leuven)
- Petr Vilím (Coenzyme Fr)
- Ruiwei Wang (National University of Singapore)
- Nic Wilson (Insight Centre for Data Analytics, School of Computer Science and IT, University College Cork)
- Felix Winter (TU Wien)
- Armin Wolf (Fraunhofer)
- Lebbah Yahia (University of Oran 1)
- Roland Yap (National University of Singapore)
- Neil Yorke-Smith (Delft University of Technology)
- Tallys Yunes (University of Miami)
- Allen Z. Zhong (Monash University)

Additional Reviews

- Gennaro Auricchio (University of Padova)
- Ignace Bleukx (KU Leuven)
- Timothy van Bremen (National University of Singapore)
- Jeffrey M. Dudek
- Maarten Flippo (Delft University of Technology)
- Marco Foschini (KU Leuven)
- Jan Høula (University of Ostrava)
- Javier Larossa (Universitat Politecnica de Catalunya)
- Yong Lai (Jilin University)
- Jacobus G. M. van der Linden (TU Delft)
- António Morgado (Universidade de Lisboa)
- Macarena Navarro (Carnegie Mellon University)
- Keisuke Okumra (National Institute of Advanced Industrial Science and Technology / University of Cambridge)
- Tomás Peitl (TU Wien)
- Nicolas Prcovic (University of Aix-Marseille)
- Žaneta Semanišinová (TU Dresden)
- Konstantin Sidorov (TU Delft)
- Ramanujan M. Sridharan (University of Warwick)
- Sebastian Vasquez (Carnegie Mellon University)
- Jiong Yang

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Solving Patience and Solitaire Games with Good Old Fashioned AI

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

Abstract

While games like Chess, Checkers and Go have been the subject of extensive research in AI for decades, there has been comparatively little study of single player card games. These games are generally called "Patience" in British English and "Solitaire" in US English, and have been popular for hundreds of years and remain so today. In fact, our ignorance of the winnability percentage of just one such game – "Klondike" – has been described as "one of the embarrassments of applied mathematics" by the distinguished statistician Persi Diaconis.

I will talk about "Solvitaire", a program to solve patience games given a simple JSON description of the rules of the game and the initial layout. We have used Solvitaire to determine the winnability percentage of dozens different single-player card games with a 95% confidence interval of ± 0.1% or better. For example, we now know the winnability of Klondike as 81.945% ± 0.084% (in the "thoughtful" variant where the player knows the rank and suit of all cards), a 30-fold reduction in confidence interval over the best previous result. The vast majority of results we obtained with Solvitaire are either entirely new or represent significant improvements on previous knowledge.

Solvitaire is very much a "Good Old Fashioned AI" approach to solving patience games, without using Machine Learning or Neural networks. It uses exhaustive depth-first search to explore all possible ways that a game could possibly be won, ensuring that games reported unwinnable really are so. This can involve searching extraordinary seach spaces with depths in the millions even including cases where unwinnability is proven. Numerous techniques imported from AI search play an important role in making this search practicable. Particularly important ones are: the use of a transposition tables; the exploitation of symmetry in search; the use of dominances to force certain moves to be made when it is safe to do so; and the use of streamliners. Solvitaire does have some games it performs poorly on, where exhaustive search is unable to prove that no win is possible but an alternative simple proof is in fact available. I will also talk about using constraint models do this, leading to slight improvements in some variants of Klondike but dramatic improvements in others.

This talk will include personal anecdotes, explaining for example why it is dedicated to my mother Margaret Gent (1923-2021) for her patience in teaching me to love the game of patience.

2012 ACM Subject Classification Computing methodologies → Discrete space search

Keywords and phrases AI Search, Solitaire and Patience Games

Digital Object Identifier 10.4230/LIPIcs.CP.2024.1

Category Invited Talk

Related Version *The Winnability of Klondike Solitaire and Many Other Patience Games, by Charlie Blake and Ian P. Gent*: `https://arxiv.org/abs/1906.12314`

Acknowledgements I pay special thanks to my former students Charlie Blake and Jack Waller who did much of the outstanding work this talk is based on. I thank all others who have helped me in my work on patience, including Matt Birrell, Dawn Black, Laura Brewis, Arthur W. Cabral, Gal Cohensius, Nguyen Dang, Shlomi Fish, Jordina Francès de Mas, Alan Frisch, Chris Jefferson, Michael Keller, Donald Knuth, Dana Mackenzie, Mark Masten, Ian Miguel, Peter Nightingale, Theodore Pringle, Bill Roscoe, András Salamon, Felix Ulrich-Oltean, Judith Underwood, and (posthumously) Jan Wolter. I thank Paul Shaw for inviting me to give this talk.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 1; pp. 1:1–1:1
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Thinking Fast and Slow in AI: A Cognitive Architecture to Augment Both AI and Human Reasoning

Francesca Rossi ✉
IBM Research, Yorktown Heights, NY, USA

Abstract

AI systems are very useful in practically every sector, but they also have several limitations, mostly related to the lack of reasoning capabilities. According to the fast and slow thinking theory of human decision making, we can say that data-driven AI, including generative AI, are providing fast thinking capabilities, but they do not have slow thinking ones. Existing cognitive theories of human decision making, such as the thinking fast and slow theory, can provide insights on how to advance AI systems towards some of these capabilities. In this talk I will present a general architecture, called SOFAI, that is based on fast/slow solvers and a meta-cognitive component that provides a centralized governance of the solvers. I will describe two instances of this architecture, for constrained grid navigation and planning, showing experimentally that SOFAI generates better decisions than each of the *individual* solvers. Emerging behavior related to adaptability, skill learning, and cognitive control are also showed in the analysis of SOFAI's behavior. I will also describe how the thinking fast and slow theory can help design a value-based human-machine collaborative decision environment.

2012 ACM Subject Classification Computing methodologies → Cognitive science

Keywords and phrases Artificial Intelligence, Meta-reasoning

Digital Object Identifier 10.4230/LIPIcs.CP.2024.2

Category Invited Talk

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 2; pp. 2:1–2:1
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The Complexity of Symmetry Breaking Beyond Lex-Leader

Markus Anders
TU Darmstadt, Germany

Sofia Brenner
TU Darmstadt, Germany

Gaurav Rattan
University of Twente, Enschede, The Netherlands

Abstract

Symmetry breaking is a widely popular approach to enhance solvers in constraint programming, such as those for SAT or MIP. Symmetry breaking predicates (SBPs) typically impose an order on variables and single out the lexicographic leader (lex-leader) in each orbit of assignments. Although it is NP-hard to find complete lex-leader SBPs, incomplete lex-leader SBPs are widely used in practice.

In this paper, we investigate the complexity of computing complete SBPs, lex-leader or otherwise, for SAT. Our main result proves a natural barrier for efficiently computing SBPs: efficient certification of graph non-isomorphism. Our results explain the difficulty of obtaining short SBPs for important CP problems, such as matrix-models with row-column symmetries and graph generation problems. Our results hold even when SBPs are allowed to introduce additional variables. We show polynomial upper bounds for breaking certain symmetry groups, namely automorphism groups of trees and wreath products of groups with efficient SBPs.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases symmetry breaking, boolean satisfiability, matrix models, graph isomorphism

Digital Object Identifier 10.4230/LIPIcs.CP.2024.3

Related Version *Preprint*: https://arxiv.org/abs/2407.04419

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

1 Introduction

The search space of a constraint program can exhibit a large amount of symmetry. This simple yet far-reaching observation forms the core principle behind the use of *symmetry based* approaches in the realm of constraint programming [23, 44]. Such methods prune the symmetric parts of the search space to save computational costs. Ideally, they ensure that at most one solution exists per equivalence class of candidate solutions. Over the last two decades, numerous methods have been proposed to exploit symmetries of constraint programs. In particular, many approaches have been developed for Boolean satisfiability solvers [14, 1, 13, 17, 27, 30, 16, 43, 38] as well as mixed integer programming [35, 39, 40]. Symmetry-based solving remains an active and fruitful area of interest, especially from a practical perspective: for example, the defining feature of arguably one of the most successful entries in the SAT competition 2023 was symmetry breaking [11, 10].

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 3; pp. 3:1–3:24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

How symmetries should be used best remains unclear. Approaches can be roughly divided into two different categories: in *dynamic* and *static* approaches. In a *dynamic* approach, symmetries are used during the execution of a solver [43, 18, 16, 38]. A typical example is that the solver incorporates a branching rule that makes use of the symmetries directly [43].

The second approach is the *static* use of symmetries, which is the main focus of this paper. Here, additional constraints, so-called *symmetry breaking predicates* (SBPs), are added to a given problem instance. The notion of SBPs was first introduced in the seminal paper of Crawford, Ginsberg, Luks and Roy [14]. Their goal was to generate polynomial-sized SBPs for SAT formulas in conjunctive normal form (CNF). However, since their framework is rooted in group theory, many results neatly generalize to other constraint languages.

The framework of Crawford et al., as well as the majority of the subsequent work in this area, uses so-called *lex-leader* predicates to achieve complete symmetry breaking. Using *incomplete* lex-leader predicates is arguably one of the most successful approaches to symmetry breaking in practice [1, 17]. On a complexity-theoretic level, however, Crawford et al. proved that computing a predicate true of only the lex-leader in each equivalence class of Boolean assignments is NP-hard. Subsequent results showed that this even holds true for restricted classes of groups [34], as well as orders similar to lex-leader [29, 46].

One may wonder whether there are *other kinds* of SBPs that are efficiently computable. Here, *other kinds* of SBPs simply means that they do *not* make use of a lexicographic ordering of the assignments. In principle, choosing any canonical representative among symmetric assignments is permissible, lex-leader or otherwise. This question is motivated, for instance, by the realm of graph isomorphism (GI). There, choosing the lex-leader is also known to be NP-hard [7], and the best theoretical and practical approaches make use of other mechanisms.

Concerning practical symmetry breaking, only a few, though surprisingly different, approaches of generating non-lex-leader SBPs have been explored. In [22], the global cardinality constraint [42] is used in conjunction with lex-leader constraints to efficiently handle (particular) wreath symmetry. In [13], SAT symmetry breaking constraints for graph problems are produced similarly to the canonical labeling algorithm NAUTY [37]. In [25], minimal SAT symmetry breaking constraints are generated for small groups.

In general, however, the complexity of SBPs remains largely unexplored, even for fairly restricted kinds of symmetries. Perhaps the most glaring example is the problem of breaking *row-column symmetries*, which arise in the so-called *matrix models* [20]. These models allow the decision variables to be arranged in a matrix such that interchanging any two rows or any two columns is a symmetry of the model. Matrix models arise in multiple areas of constraint programming such as scheduling, combinatorial problems, and design [21]. Perhaps the most well-known matrix model is the pigeonhole principle problem, for which it is NP-hard to compute the lex-leader or similar assignments [14]. While the problem of devising SBPs for such models has received much attention [20, 29, 23], the known results do not explain the lack of compact SBPs for matrix models.

Our Results

The objective of this paper is to further investigate the exact complexity of computing static symmetry breaking predicates. *Given a group of symmetries on the variables of a formula, how hard is it to generate a complete symmetry breaking predicate?* The ultimate goal of our work is to obtain a classification of symmetry groups, in terms of the complexity of computing SBPs. Such a classification could help inform practitioners as to which cases can be handled easily, and which ones are more challenging.

In order to simplify the exposition, our setting of choice is that of Boolean satisfiability testing (SAT). However, in the same vein as [14], our results are founded in a general group-theoretic setting, so they should easily transfer to many branches of constraint programming: we consider computing symmetry breaking predicates for a given *permutation group*, instead of a particular SAT formula exhibiting such symmetry.

Our results can be divided into *hardness results* and *upper bounds*. The high-level idea for the hardness results can be summarized as follows: We show that if symmetry breaking is feasible for certain expressive groups, such as matrix groups or Johnson actions, then graph isomorphism is in coNP. The containment GI $\in$ co-NP is a major unresolved problem [31], even for the restricted case of group isomorphism [5]. While GI $\in$ co-NP seems to have no other major complexity theoretic consequences and is seemingly not “implausible”, it still poses a barrier to compact SBPs.

The idea of our reductions is to encode the input graphs as binary strings in a suitable way, then guess a canonizing permutation, and use the symmetry-breaking constraint to verify that the result is indeed the canonical form: By definition, the symmetry-breaking constraint is true of precisely the canonical forms. The graphs are non-isomorphic exactly when the canonical forms are different. As a strengthening, we show that this holds even when the symmetry breaking constraint uses additional variables as their values can be guessed as well.

We now explain our hardness results in greater detail.

Matrix Models. Our first result tackles the difficulty of breaking row-column symmetries in matrix models. As mentioned above, this problem has received much attention in symmetry breaking literature.

▶ **Theorem 1.** *Suppose there exists a polynomial time algorithm for generating complete symmetry breaking predicates for row-column symmetries. Then* GI $\in$ co-NP *holds, i.e., graph non-isomorphism admits a non-deterministic polynomial time algorithm.*

Our theorem explains the difficulty of obtaining compact symmetry breaking predicates for matrix models, in the sense that it would imply polynomial time algorithms for certifying graph non-isomorphism. Section 3.1 contains a detailed description of our result.

Johnson Actions. We identify yet another class of groups for which symmetry breaking is hard, namely the (k,t)-*Johnson groups*. These are symmetric groups $\mathrm{Sym}(k)$ acting on t-subsets of $[k]$ for fixed $t < k$. It is well-known that these actions form an important sub-case of Babai’s quasi-polynomial algorithm for graph isomorphism [5].

▶ **Theorem 2.** *Let $t > 1$ be a fixed positive integer. Suppose that we can generate complete symmetry breaking predicates for all (k,t)-Johnson groups in polynomial time (in terms of the domain size). Then* GI $\in$ coNP *holds.*

Section 4 contains a formal description of this result. In fact, it follows from Theorem 16, which proves a stronger statement.

Certificates for Canonization. We strengthen our hardness results of Theorem 1 and Theorem 2 as follows. We allow an algorithm to produce more expressive SBPs:

1. The SBP can be given as a Boolean circuit.
2. The SBP is allowed to *introduce additional variables*. Essentially, this gives the predicate access to additional non-determinism. The SBP may introduce an arbitrary number of additional variables, as long as the overall size is polynomial.

Despite allowing more powerful SBPs, we conclude a stronger hardness implication: an efficient algorithm for such predicates implies an *efficiently verifiable graph canonical form* (see Theorem 13 and Theorem 15). Note that an efficient verifier for graph canonical forms implies an efficient verifier for graph non-isomorphism (Lemma 12), but the converse is unknown. For this result, we observe that SBPs for a permutation group G on the domain $[n]$ essentially solve a particular *decision version* of the *string canonization* problem w.r.t. G on strings of length n. String canonization is a fundamental problem of interest in the graph isomorphism community [7, 5, 6]. Section 3.2 contains a detailed description. Moreover, we prove that the hardness results also hold for all subgroups of polynomial index (see Lemma 18).

Quasi-Polynomial Upper Bound. Realizing that symmetry breaking reduces to string canonization allows us to express an upper-bound on the size of *circuit* SBPs for general permutation groups. The result is mainly of theoretical interest, but we believe that this could have useful consequences in SBP heuristic design. The theorem immediately follows from the quasi-polynomial time algorithm of Babai [6], see Section 3.2 for more details.

▶ **Theorem 3.** *Given a permutation group $G \leq \mathrm{Sym}(n)$, there is a quasi-polynomial time (in n) algorithm producing a complete symmetry breaking circuit of quasi-polynomial size.*

We complement these results concerning *hard* families by focusing on polynomial upper bounds, i.e., the question for which families of groups symmetry breaking is *easy*:

Polynomial Upper Bounds. In Section 5, we examine how group-theoretic structure can help to design SBPs. Our results show how we can assemble SBPs for a group from the SBPs of its constituents, in context of natural operations such as disjoint direct products and wreath products. This extends the results of [24], where the existence of *lex-leader* constraints for constituents is assumed to assemble constraints for direct products and wreath products. (The paper also treats cyclic, dihedral and alternating groups.)

The following theorem is the main consequence of our results in this section.

▶ **Theorem 4.** *Assume that $G \leq \mathrm{Sym}(n)$ is the automorphism group of a tree T. Then G admits a complete symmetry breaking predicate of linear size. Given the tree T, it can be computed in polynomial time.*

Automorphism groups of trees are special cases of so-called wreath products. Such groups naturally occur, for example, whenever models exhibit hierarchical structure. Intuitively, the structure can be split into parts with the same symmetry group (the *base group*), which are permuted by the so-called *top group*. Essentially, we combine symmetry breaking constraints for the base group and the top group to a symmetry breaking constraint for the wreath product by using the predicate for base group to make every part canonical, and the constraint of the top group to fix an ordering of the parts. For the general case of wreath products the problem is far more technical, but we obtain the following result (see Section 5 for details).

▶ **Theorem 5.** *Let $G \leq \mathrm{Sym}(n)$ and $H \leq \mathrm{Sym}(m)$ be permutation groups. Assume that a complete symmetry breaking circuit for G can be computed in polynomial time. Moreover, suppose that for every partition P of $[m]$, the partition stabilizer S of P in H and a complete symmetry breaking circuit for S can be computed in polynomial time. Then there is a complete symmetry breaking circuit for the wreath product $W := G \wr H$ that can be computed in polynomial time.*

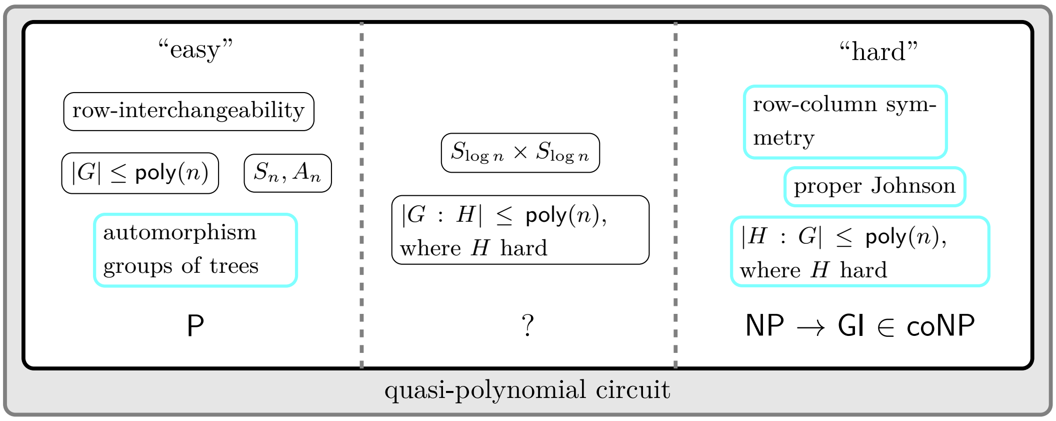

"easy"
row-interchangeability
$|G| \leq \mathsf{poly}(n)$
S_n, A_n
automorphism groups of trees
P
$S_{\log n} \times S_{\log n}$
$|G : H| \leq \mathsf{poly}(n)$, where H hard
?
"hard"
row-column symmetry
proper Johnson
$|H : G| \leq \mathsf{poly}(n)$, where H hard
NP → GI ∈ coNP
quasi-polynomial circuit

Figure 1 Complexity of computing symmetry breaking predicates for the stated families of groups in SAT. All groups can be handled in quasi-polynomial time using a circuit. The symbol G refers to the permutation group of consideration. The parameter n refers to the domain size of the permutation group, or, the number of variables of the formula. For "easy" families of groups, a CNF predicate can be computed in polynomial time. For "hard" families of groups, the existence of polynomial time symmetry breaking, even allowing the use of additional variables, implies that GI is in coNP. Blue outlines indicate novel results proven in this paper.

In summary, Figure 1 provides a concise description of our progress towards a complexity classification for the problem of generating SBPs for permutation groups.

2 Preliminaries

2.1 Boolean Circuits and Satisfiability

Boolean Circuits. A Boolean circuit ψ is a circuit consisting of input gates, one output gate, and $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$-gates connecting them in the usual way. We refer to the input gates as the *variables* $\mathrm{Var}(\psi)$. The *size* of a circuit refers to the number of gates.

An *assignment* of ψ is a function $\theta\colon V \to \{0,1\}$ where $V \subseteq \mathrm{Var}(\psi)$. The assignment is *complete* whenever $V = \mathrm{Var}(\psi)$ and *partial* otherwise. A circuit is evaluated using an assignment $\theta\colon V \to \{0,1\}$, by replacing each input gate $v \in V$ with $\theta(v)$, with the usual meaning. The resulting circuit is $\psi[\theta]$. Whenever θ is complete, the value of the output gate can be determined in linear time, and hence either $\psi[\theta] = 0$ or $\psi[\theta] = 1$ holds.

If $\psi[\theta] = 1$ we call θ a *satisfying assignment*, whereas if $\psi[\theta] = 0$ we call θ a *conflicting assignment*. A circuit ψ is *satisfiable* if and only if there exists a satisfying assignment to ψ.

Conjunctive Normal Form. In practice, a SAT instance ψ is typically given in *conjunctive normal form* (CNF), which we denote with $\psi = \{\{l_{1,1} \vee \dots \vee l_{1,k_1}\} \wedge \dots \wedge \{l_{m,1} \vee \dots \vee l_{m,k_m}\}\}$. Each element $C \in \psi$ is called a *clause*, whereas a clause itself consists of a set of *literals*. A literal is either a variable v or its negation $\overline{v}$.

A symmetry, or *automorphism*, of ψ is a permutation of the variables $\varphi\colon \mathrm{Var}(\psi) \to \mathrm{Var}(\psi)$ which maps ψ back to itself, i.e., $\psi^\varphi \equiv \psi$, where φ is applied element-wise to the variables in each clause. The permutation group of all symmetries of ψ is $\mathrm{Aut}(\psi) \leq \mathrm{Sym}(\mathrm{Var}(\psi))$.

Another common way to define symmetries is to define them on the *literals* of the formula, allowing the use of so-called *negation symmetries* (see [44]). In any case, symmetries can be efficiently computed in practice using state-of-the-art symmetry detection tools [37, 28, 15, 2].

2.2 Permutation Groups

We briefly introduce some notation and results for permutation groups. For further background material on permutation groups, we refer to [19]. Throughout, we use the notation $[n] \coloneqq \{1,\dots,n\}$ for $n \in \mathbb{Z}_{>0}$ and set $[0] \coloneqq \emptyset$.

Let Ω be a nonempty finite set. Let $\mathrm{Sym}(\Omega)$ denote the *symmetric group* on Ω, i.e., the group of permutations of Ω. A *permutation group* is a subgroup G of $\mathrm{Sym}(\Omega)$, denoted by $G \leq \mathrm{Sym}(\Omega)$. We also say that G *acts on* Ω. A permutation group is always specified by the abstract isomorphism type of G (for instance, G could be cyclic of order 10), together with the action of G on Ω. For $g \in G$ and $\omega \in \Omega$, we write ω^g for the image of ω under g and $\omega^G = \{\omega^g \colon g \in G\}$ for the *orbit* of ω under G. The *support* of G consists of those elements in Ω that are moved (i.e., not fixed) by some element of G. For a partition $P = (\Omega_1,\dots,\Omega_r)$ of Ω (i.e., $\Omega = \Omega_1 \dot\cup \cdots \dot\cup \Omega_r$), the *partition stabilizer* of P in G consists of all elements $g \in G$ that setwise stabilize $\Omega_1,\dots,\Omega_r$, i.e. for all $i \in [r]$, $\{\omega^g : \omega \in \Omega_i\} = \Omega_i$. The *index* of a subgroup H of G is $|G : H| \coloneqq |G|/|H|$.

Two permutation groups $G \leq \mathrm{Sym}(\Omega)$ and $H \leq \mathrm{Sym}(\Delta)$ are *permutation isomorphic* if there exists a bijection $\lambda \colon \Omega \to \Delta$ and a group isomorphism $\alpha \colon G \to H$ such that $\lambda(\omega^g) = \lambda(\omega)^{\alpha(g)}$ for all $\omega \in \Omega$ and $g \in G$. Note that this notion is stronger than G and H being isomorphic (as abstract groups) as the same abstract group can give rise to different group actions. For instance, $\mathrm{Sym}(k)$ admits so-called Johnson actions on different domains:

Johnson Groups. Let k be a positive integer and $t \in [k-1]$. A permutation $\pi \in \mathrm{Sym}(k)$ induces a permutation $\hat\pi$ on the domain $\binom{[k]}{t}$ of t-subsets of $[k]$, mapping a t-subset A to $A^{\hat\pi} = \{a^\pi \colon a \in A\}$. This way, $\mathrm{Sym}(k)$ becomes a permutation group $S_k^{(t)}$ on a domain of size $\binom{k}{t}$. The groups $S_k^{(t)}$ are called *Johnson groups* and the action is called a *Johnson action.* We call a Johnson group *proper* if $t \notin \{1, k-1\}$ holds.

Usually, the analogous action of the so-called alternating groups is also called a Johnson action. Due to our results in Section 4.3, it suffices to only consider the symmetric groups.

Wreath products. Let $G \leq \mathrm{Sym}(\Omega)$ and $H \leq \mathrm{Sym}(\Delta)$ be permutation groups. The *wreath product* $G \wr H$ consists of the set $G^\Delta \times H$, endowed with the multiplication rule $\big((g_\delta)_{\delta\in\Delta}, h\big)\big((g'_\delta)_{\delta\in\Delta}, h'\big) = \big((g_\delta g'_{\delta^{h^{-1}}})_{\delta\in\Delta}, hh'\big)$. We call G the *base group* and H the *top group.* The group $G \wr H$ acts on $\Omega \times \Delta$ by $(\omega,\delta)^{((g_\delta)_{\delta\in\Delta},h)} = (\omega^{g_{\delta^h}}, \delta^h)$. This action is called the *imprimitive action* of the wreath product.

2.3 Graph Isomorphism and String Canonization

Graphs. A finite, undirected graph $\Gamma = (V, E)$ consists of a set of vertices $V \subseteq \mathbb{N}$ and an edge relation $E \subseteq \binom{V}{2}$. Unless stated otherwise, the set of vertices V is $\{1,\dots,n\}$ and $m \coloneqq |E|$ denotes the number of edges. We may refer to the set of vertices of Γ with $V(\Gamma)$, and to the set of edges with $E(\Gamma)$. The *adjacency matrix* of Γ is the $n \times n$-matrix $A = (a_{ij})$ with $a_{ij} = 1$ if $\{i,j\} \in E(\Gamma)$, and $a_{ij} = 0$ otherwise. Unless stated otherwise, we assume our graphs are given as adjacency matrices.

A graph Γ is *bipartite* if $V(\Gamma) = A \dot\cup B$ can be partitioned into two independent sets $A = \{a_1,\dots,a_k\}$ and $B = \{b_1,\dots,b_\ell\}$. In this case, we may obtain an *bipartite adjacency matrix* $M = (m_{ij})$ by setting $m_{ij} = 1$ if a_i and b_j are adjacent, and $m_{ij} = 0$ otherwise.

Lexicographic ordering. For $\{0,1\}$-strings θ, θ' of the same length, we write $\theta \preceq_{\text{lex}} \theta'$ if θ is smaller or equal to θ' with respect to the lexicographic ordering. Likewise, we define a lexicographic ordering of matrices with entries in $\{0,1\}$ of a fixed size by interpreting them as strings, reading them row by row.

Relational Structures. As a generalization of graphs, we define a *t-ary relational structure* $R = (U, A)$, where U is a universe and A is a t-ary relation on U. A t-ary relational structure is *symmetric* if for every t-tuple $(u_1, \ldots, u_t) \in A$ and for every $\sigma \in \mathrm{Sym}(t)$, it holds that $(u_{\sigma(1)}, \ldots, u_{\sigma(t)}) \in A$.

Graph Isomorphism. Two graphs $\Gamma_1 = (V_1, E_1), \Gamma_2 = (V_2, E_2)$ are said to be *isomorphic*, whenever there exists a bijection $\varphi \colon V_1 \to V_2$ such that $\varphi(\Gamma_1) = (V_1^\varphi, E_1^\varphi) = (V_2, E_2) = \Gamma_2$ holds. Here, V_1^φ and E_1^φ means applying φ element-wise to each element in V_1, and each element of each tuple in E_1, respectively. We call φ an *isomorphism* between Γ_1 and Γ_2. We may write $\Gamma_1 \cong \Gamma_2$ to denote isomorphism. A corresponding computational problem follows:

▶ **Problem 6** (GI). *Given two graphs Γ_1, Γ_2, does $\Gamma_1 \cong \Gamma_2$ hold?*

Regarding certification, it is easy to see that GI is in NP. On the other hand, graph isomorphism is known to be in coAM, i.e., there are efficient randomized proofs for non-isomorphism [9]. As mentioned in the introduction, whether graph isomorphism is in coNP is a long-standing open problem [31].

Analogously, we may define isomorphism for t-ary relational structures $R_1 = (U_1, A_1)$ and $R_2 = (U_2, A_2)$: R_1 and R_2 are *isomorphic* if there exists a bijection $\pi \colon U_1 \to U_2$ such that for every $(u_1, \ldots, u_t) \in A_1$, it holds that $(u_1^\pi, \ldots, u_t^\pi) \in A_2$ and vice-versa.

String Canonization. We next define the *string canonization* problem [7, 32]. The string canonization problem asks, given a permutation group $G \leq \mathrm{Sym}(\Omega)$ and a string $\sigma \colon \Omega \to \Sigma$ on a finite alphabet Σ, for a canonical representative of σ^G. In particular, it computes a function $F \colon \mathcal{G} \times \Sigma^\Omega \to \Sigma^\Omega$ where $\mathcal{G}$ denotes the set of all permutation groups $G \leq \mathrm{Sym}(\Omega)$, and for all $\sigma_1, \sigma_2 \in \Sigma^\Omega$ it holds that (1) $F(G, \sigma_1) \cong_G \sigma_1$ and (2) if $\sigma_1 \cong_G \sigma_2$ then $F(G, \sigma_1) = F(G, \sigma_2)$. Here, $\cong_G$ means that σ_1 can be permuted to σ_2 using an element of G. A corresponding computational problem follows:

▶ **Problem 7** (s-SCANON$_F$). *Given a permutation group $G \leq \mathrm{Sym}(\Omega)$, a finite alphabet Σ and a string $\sigma \in \Sigma^\Omega$, compute the canonical representative $F(G, \sigma)$.*

The graph isomorphism problem polynomial time reduces to s-SCANON, but the converse is unknown. However, there is an F such that there is a quasi-polynomial time algorithm which solves the string canonization problem [6]. It turns out that the string canonization problem is intimately related to symmetry breaking, which we discuss thoroughly in Section 3.2.

A crucial special case of string canonization is *graph canonization.* As the name suggests, it computes canonical forms for graphs. Let f be a graph canonization function, i.e., for graphs Γ, Δ, it holds that (1) $\Gamma \cong \Delta$ iff $f(\Gamma) = f(\Delta)$, and, (2) $f(\Gamma) \cong \Gamma$. Here, the symbol $\cong$ denotes the graph isomorphism relation. The corresponding computational problem follows:

▶ **Problem 8** (s-GCANON$_f$). *Given a graph Γ, compute the canonical representative $f(\Gamma)$ within the isomorphism class of Γ.*

Indeed, this problem is a special case of string canonization: G can be chosen as a Johnson group of appropriate order and the strings encode the given graphs (see [6]).

2.4 Notions of Symmetry Breaking

Next, we define our notions of symmetry breaking. Let ψ be a CNF formula. Typically, symmetry breaking is defined specifically for the automorphism group $\mathrm{Aut}(\psi)$ of ψ. However, it turns out that often, our symmetry breaking predicates only depend on the structure of $\mathrm{Aut}(\psi)$ and its action on the set of variables $\mathrm{Var}(\psi)$. In particular, they do not depend on the specific shape of the formula ψ. Hence, we define symmetry breaking only using an arbitrary permutation group $G \leq \mathrm{Sym}(\Omega)$ and without referring to a precise formula ψ.

Symmetry Breaking Constraints. We begin with a discussion of complete symmetry breaking. Indeed, we find that in the literature two different notions are in use.

The first of these notions is what we will refer to simply as *complete symmetry breaking*. The idea is that a complete symmetry breaking constraint must ensure that in each orbit of *complete* assignments, all but one canonical representative is conflicting [14].

Formally, we let $\theta_{\text{full}}(\Omega) := \{\theta \mid \theta \colon \Omega \to \{0,1\}\}$ denote the set of all complete assignments to Ω. We let $G \leq \mathrm{Sym}(\Omega)$ act on $\theta_{\text{full}}(\Omega)$ in the natural way. A Boolean circuit ψ with $\mathrm{Var}(\psi) \subseteq \Omega$ is called a *complete* symmetry breaking circuit for G, whenever for each orbit $O \subseteq \theta_{\text{full}}(\Omega)$ under G, there is

- a $\tau \in O$ such that $\psi[\tau]$ is satisfying,
- for all $\tau' \in O$ with $\tau \neq \tau'$ the formula $\psi[\tau']$ is conflicting.

If ψ is restricted to be a CNF formula, we refer to ψ as a symmetry breaking *predicate*. This notion is typically used in the context of general-purpose symmetry breaking, such as for example in [14, 1, 17, 25]. We remark that in [25], this notion is referred to as an *isolator*.

The second notion in use in the literature is *isomorph-free generation*. It is usually considered in the realm of *dynamic* symmetry breaking. However, a notion for predicates can be defined: a predicate is supposed to ensure that in each orbit of *partial* assignments, all but one canonical representative is conflicting. Intuitively, isomorph-free generation ensures that no isomorphic branches are *ever* considered in the search. Isomorph-free generation immediately also ensures complete symmetry breaking. It is typically used in the context of generation tasks, such as in [36, 30], but it has also been considered for general-purpose symmetry breaking [27].

The focus of this paper is on complete symmetry breaking and not on isomorph-free generation.

Symmetry Breaking as a Computational Problem. We define a corresponding computational problem for symmetry breaking.

▶ **Problem 9** (Symmetry Breaking). *Given a permutation group* $G \leq \mathrm{Sym}(\Omega)$*, compute a complete symmetry breaking circuit for* G*.*

There are two variations of this problem that we discuss throughout the paper: the first of which concerns the group G. Usually, G is the automorphism group of a given CNF formula ψ, i.e., $G = \mathrm{Aut}(\psi)$. In this case, the problem might become easier, since automorphism groups and a given formula may admit further structural arguments. However, considering symmetry breaking for arbitrary permutation groups G opens up the possibility of using symmetries *beyond* syntactic ones, even though it might be unclear how they could be obtained. Furthermore, results are independent of the specific structure of SAT instances.

The second variation concerns the output: we may expect a CNF predicate, or a Boolean circuit. Computing a CNF predicate may be harder, since circuits are more expressive. We believe that all variations of the problem are of potential interest. Therefore, it seems best to attempt to use the problem definition which yields the strongest possible statement.

3 Row-Column Symmetries

In this section, we analyze the complexity of computing symmetry breaking predicates for row-column symmetry. Section 3.1 describes the hardness of obtaining SBPs for breaking row-column symmetries. In particular, we provide a proof of Theorem 1. Section 3.2 establishes the connection between symmetry breaking and decision string canonization. Lastly, in Section 3.3, we strengthen our results to work for circuit SBPs and SBPs with extra variables.

3.1 Hardness of Breaking Row-Column Symmetries

We begin with a formal definition of row-column symmetry.

Row-Column Symmetry. Let m, n be two positive integers, and $\Omega := [n] \times [m]$. The *row-column symmetry* group G is defined to be the group $\mathrm{Sym}([n]) \times \mathrm{Sym}([m])$, where $\mathrm{Sym}([n])$ naturally acts on the first component of Ω, and $\mathrm{Sym}([m])$ on the second component. Informally, we can view Ω as a *matrix* with n rows and m columns. The group $G \leq \mathrm{Sym}(\Omega)$ then consists of all the possible row transpositions and all possible column transpositions, along with their arbitrary compositions.

A *matrix model* is a constraint program whose decision variables can be arranged as a matrix above such that its automorphism group is the row-column symmetry group for this matrix arrangement. For such programs, it is typical to index their variable set by $\Omega = \{x_{ij} \mid i \in [n], j \in [m]\}$.

The following lemma states a one-to-one correspondence between assignments to a matrix model and bipartite graphs. Let $\Gamma(U, V)$ denote a bipartite graph with a designated left-partition U and a right-partition V, where U and V are non-interchangeable.

▶ **Lemma 10.** *There exists a one-to-one correspondence between the set of all Boolean assignments to the variables $\{x_{11}, \ldots, x_{nm}\}$ of a matrix model and the set of all bipartite graphs $\Gamma([n], [m])$ with designated left and right partitions.*

Proof. Interpret the truth-value of x_{ij} as the indicator for whether there exists an edge between $i \in [n]$ and $j \in [m]$. ◀

We proceed with the proof of Theorem 1.

Proof of Theorem 1. We devise a polynomial time verifier for checking purported certificates for non-isomorphism, assuming that we can compute a row-column symmetry breaking predicate in polynomial time.

Bipartite Graphs Suffice. It will be more convenient for us to work with bipartite graphs instead of general graphs, in the spirit of standard reductions in isomorphism literature [47]. To every graph Γ, we can always associate a bipartite graph $\mathrm{bip}(\Gamma)$, namely the vertex-edge incidence graph as follows. The graph $\mathrm{bip}(\Gamma)$ has a designated *left* partition consisting of $V(\Gamma)$, a designated *right* partition consisting of $E(\Gamma)$, and the edges of $\mathrm{bip}(\Gamma)$ are defined by vertex-edge incidence. Moreover, $\mathrm{bip}(\Gamma)$ is a vertex-ordered graph: the left partition inherits the ordering from the graph Γ, and the right partition $E(\Gamma)$ is ordered according to the ordering induced by the ordering of $V(\Gamma)$. Observe that the mapping $\Gamma \mapsto \mathrm{bip}(\Gamma)$ is injective. Moreover, it is easy to verify that two graphs Γ and Δ are isomorphic if and only if the bipartite graphs $\mathrm{bip}(\Gamma)$ and $\mathrm{bip}(\Delta)$ are isomorphic via a bijection which maps the left-partition (right-partition) of $\mathrm{bip}(\Gamma)$ to the left-partition (right-partition) of $\mathrm{bip}(\Delta)$.

Therefore, it suffices to verify non-isomorphism certificates for bipartite graphs.

Certificate. Given two bipartite graphs Γ and Δ, our chosen certificate of non-isomorphism is a pair of bijections (σ, π), where $\sigma \colon V(\Gamma) \to V(\Gamma)$ and $\pi \colon V(\Delta) \to V(\Delta)$.

Verifier. Given such a certificate (σ, π), our polynomial time verifier proceeds as follows:

1. Compute a symmetry breaking predicate $\delta_{n,m}(x_{11}, \ldots, x_{nm})$ in time $\mathsf{poly}(n, m)$.
2. Check if both Γ^σ and Δ^π satisfy $\delta_{n,m}(x_{11}, \cdots, x_{nm})$, when viewed as Boolean assignments. If both of them satisfy $\delta_{n,m}$, continue; otherwise reject.
3. Check whether $\Gamma^\sigma \neq \Delta^\pi$, otherwise reject.
4. Declare Γ and Δ to be non-isomorphic.

It is easy to verify that all of the steps above are polynomial time computations.

Correctness of Verifier. It remains to be shown that (1) for every pair of non-isomorphic graphs, there exists a polynomial sized certificate accepted by the verifier above, and (2) for every pair of isomorphic graphs, the verifier always rejects any certificate.

For (1), let Γ and Δ be two non-isomorphic graphs. Let Γ^* be the unique satisfying assignment of $\delta_{n,m}$ in the orbit of Γ (similarly define Δ^*) under row-column symmetries. Let σ be an isomorphism from Γ to Γ^* (similarly define π). Since $\Gamma \ncong \Delta$, it must hold that Γ^* and Δ^* lie in different orbits, and hence $\Gamma^* \neq \Delta^*$. Since the certificate satisfies all conditions of the verifier, the verifier correctly certifies Γ and Δ to be non-isomorphic.

For (2), suppose Γ and Δ are isomorphic. Then they lie in the same orbit under the action of row-column symmetry on the Boolean assignments to the matrix model. Given any certificate (σ, π), the requirement of Γ^σ and Δ^π having to satisfy $\delta_{n,m}$ implies that $\Gamma^\sigma = \Delta^\pi$. But then such a certificate is rejected by the verifier in the third step. Hence, the verifier correctly refuses to certify that Γ and Δ are non-isomorphic. ◀

It is not clear whether the converse of Theorem 1 holds. In fact, even a P-time algorithm for graph isomorphism may not be sufficient to yield symmetry breaking algorithms for row-column symmetries. In what follows, we address this situation with a closer examination of the complexity of symmetry breaking.

3.2 A Decision Version of String Canonization

We now introduce a *decision variant* of the string canonization problem, which only decides whether a given string *is* the canonical string:

▶ **Problem 11** (d-SCANON$_F$). *Given a group $G \leq \mathrm{Sym}(\Omega)$, a finite alphabet Σ and a string $\sigma \in \Sigma^\Omega$, decide whether $\sigma = F(G, \sigma)$ holds, i.e., whether σ is the canonical representative within its isomorphism class σ^G.*

Let us consider a CNF formula ψ. We consider the case of the string canonization problem where $\Sigma = \{0, 1\}$ and the group $G \leq \mathrm{Sym}(\mathrm{Var}(\psi))$ consists of symmetries of ψ. Note that any two given assignments σ_1 and σ_2 of ψ can be interpreted as strings, and $\sigma_1 \cong_G \sigma_2$ holds if and only if they are in the same orbit of G.

We observe that an algorithm for d-SCANON accepts precisely one assignment per orbit of G. But this just means that if we translate such an algorithm into a Boolean circuit, the resulting circuit *is* a symmetry breaking circuit.

Clearly, d-SCANON$_F$ polynomial time reduces to s-SCANON$_F$. Since s-SCANON can be solved using a quasi-polynomial time algorithm [6], Theorem 3 follows immediately.

Analogously, we may define a decision version of the graph canonization problem, denoted as d-GCANON$_f$. Recall that graph canonization is a special case of string canonization. In the following, we prove that the decision canonization problem is tightly related to graph isomorphism in terms of its non-deterministic complexity.

▶ **Lemma 12.** *Let f be a canonical form such that d-GCANON$_f$ is in* NP*. Then,* GI $\in$ coNP*.*

Proof. Assuming d-GCANON is in NP gives us access to a class of polynomial-sized certificates and a polynomial time verifier for these certificates, such that the following hold. If a given graph Γ is the canonical representative of its isomorphism class, then there must be a certificate σ such that the verifier accepts (Γ, σ). If Γ is not the canonical representative, then for all certificates σ the verifier rejects (Γ, σ).

Based on this, we provide a non-deterministic polynomial time algorithm for graph non-isomorphism of two graphs Γ and Δ.

Certificate. The certificate consists of two permutations $\varphi_1 \in \mathrm{Sym}(V(\Gamma)), \varphi_2 \in \mathrm{Sym}(V(\Delta))$, a certificate σ_1 for decision canonization of Γ, as well as σ_2 for decision canonization of Δ.

Verifier. Given two graphs Γ, Δ and certificate $(\varphi_1, \varphi_2, \sigma_1, \sigma_2)$, the verifier proceeds as follows. (Step 1) Run the decision canonization verifier for $(\Gamma^{\varphi_1}, \sigma_1)$ and $(\Delta^{\varphi_2}, \sigma_2)$. If both are accepted, proceed, otherwise reject. (Step 2) Accept if $\Gamma^{\varphi_1} \neq \Delta^{\varphi_2}$, otherwise reject.

Correctness of Verifier. Note that whenever we reach Step 2 of the verifier, the procedure guarantees that Γ^{φ_1} is a canonical form of Γ and Δ^{φ_2} of Δ. Hence, $\Gamma \cong \Delta$ holds if and only if $\Gamma^{\varphi_1} = \Delta^{\varphi_2}$ holds. It immediately follows that the algorithm accepts if and only if Γ and Δ are non-isomorphic. ◀

3.3 Hardness of Symmetry Breaking with Additional Variables

Consider the situation where one is allowed to use additional variables from a set Ω' to write down symmetry breaking constraints. In principle, this expands our domain Ω of variables used to $\Omega \,\dot\cup\, \Omega'$. Since the introduction of new variables Ω' changes the set of assignments, we need to adjust our definition of complete symmetry breaking.

Symmetry Breaking with Additional Variables. A Boolean circuit ψ is called a complete symmetry breaking circuit *with additional variables* for $G \leq \mathrm{Sym}(\Omega)$, whenever for each orbit $\tau \subseteq \sigma_{\mathrm{full}}(\Omega)$ under G, there is

- a $\tau' \in \tau$ such that $\psi[\tau']$ is satisfiable,
- for all $\tau'' \in \tau$ with $\tau' \neq \tau''$ the circuit $\psi[\tau'']$ is unsatisfiable.

A point of contention in the above definition might be whether $\psi[\tau']$ should actually have *exactly one* satisfying assignment. This would ensure that there is precisely one satisfying assignment per orbit, while our definition only suffices to ensure a unique satisfying assignment *when restricted* to the variables of ψ. In this paper, we stick to the above definition.

Using additional variables is typically not considered in the literature, most likely because this might substantially alter the difficulty of the underlying instance. Introducing additional variables is however intriguing: it gives the symmetry breaking predicates access to non-determinism, and hence might enable substantially more powerful constraints.

Hardness with Additional Variables. We show hardness results for symmetry breaking *even if* we are allowed to introduce new variables.

▶ **Theorem 13.** *Suppose there exists a polynomial time algorithm for generating complete symmetry breaking circuits* with additional variables *for row-column symmetries. Then, it holds that d-GCANON $\in$* NP *and hence* GI $\in$ coNP*.*

Proof sketch. It suffices to show that there exists a canonical form f such that d-GCANON$_f$ $\in$ NP (see Lemma 12). Again, we encode the input graph as a bipartite graph as in the proof of Theorem 1. The main argument follows by an inspection of the proof of Theorem 1: we observe that a certificate can also guess an assignment to the additional variables introduced by the SBP. We then simply verify that the adjacency matrix of the input graph *and* the assignment to the additional variables is accepted by the symmetry breaking circuit. ◀

The formal details of the proof can be found in Appendix A.

4 Johnson Actions

Next, we consider the so-called Johnson groups. Johnson groups are groups which naturally occur in problems encoding graph generation tasks [30]. We begin this section by describing a correspondence between Johnson groups and symmetric relational structures. Then, we provide a formal proof of Theorem 2. Lastly, we show how to derive SBPs for a group G, given SBPs for a small index subgroup $H \leq G$ (Lemma 18).

4.1 Johnson Groups and Relational Structures

Johnson Families. Let k be a positive integer. For $t \in [k-1]$, let X_k^t be the set of variables indexed by t-element subsets of $[k]$. In particular, we have $|X_k^t| = \binom{k}{t}$. For fixed $t \geq 1$, we call the group family $S_k^{(t)} \leq \mathrm{Sym}(X_k^t)$ the *Johnson family of arity t*.

Johnson groups form a subclass of the so-called groups of Cameron type. These groups as well as their natural action can be recognized in polynomial time (see [8]).

Relational Structures and Johnson Groups. To a symmetric t-ary relational structure R, we associate an assignment $f_R\colon X_k^t \to \{0,1\}$ with $f(x_S) = 1$ for a t-subset S of $[k]$ if S is a hyperedge in R, and $f(x_S) = 0$ otherwise. Conversely, given $f\colon X_k^t \to \{0,1\}$, we define a symmetric t-ary relational structure R_f on the universe $[k]$ whose relation is the set of all tuples $(a_1, \dots, a_t)$ with $f(\{a_1, \dots, a_t\}) = 1$.

This defines a one-to-one correspondence between assignments of X_k^t and symmetric t-ary relational structures. The following result formalizes the correspondence (see also [33]).

▶ **Lemma 14.** *Let R and R' be two symmetric t-ary relational structures on the universe $[k]$. Then R and R' are isomorphic if and only if the assignments f_R and $f_{R'}$ of the set X_k^t lie in the same orbit under the action of the Johnson group $S_k^{(t)} \leq \mathrm{Sym}(X_k^t)$.*

Proof. Suppose that R and R' are isomorphic via a bijection $\pi\colon [k] \to [k]$. Then, the induced action $\hat{\pi}$ on t-subsets of $[k]$ defines an element of $S_k^{(t)}$ with $f_{R'} = f_R^{\hat{\pi}}$. Conversely, suppose that $f_{R'} = f_R^{\hat{\pi}}$ for some $\hat{\pi}$ corresponding to the induced action of $\pi\colon [k] \to [k]$. It is easy to check that π is an isomorphism between R and R'. ◀

4.2 Johnson Families of Fixed Arity are Hard

In this section, we show that polynomial time symmetry breaking for Johnson families of fixed arity $t \geq 2$ implies GI $\in$ coNP.

▶ **Theorem 15.** *Suppose there exists a polynomial time algorithm for generating complete symmetry breaking circuits with additional variables for the Johnson family of arity 2. Then, d-GCANON $\in$ NP and hence GI $\in$ coNP.*

Proof. We again make use of Lemma 12, proving that polynomial time symmetry breaking circuits with additional variables for Johnson groups give rise to a non-deterministic polynomial time algorithm for decision graph canonization. Using similar arguments to Theorem 13, this follows from Lemma 14 and the fact that for two relational structures R, R' it holds that $R = R'$ if and only if $f_R = f_{R'}$. ◀

We generalize Theorem 15 to arbitrary arity.

▶ **Theorem 16.** *Let $t \geq 2$ be a fixed arity. Suppose there exists a polynomial time algorithm for generating complete symmetry breaking circuits with additional variables for the Johnson family of arity t. Then, d-GCANON $\in$ NP and hence GI $\in$ coNP.*

Proof sketch. Lemma 12 ensures that it suffices to prove d-GCANON $\in$ NP. By Lemma 14, it suffices to solve d-GCANON in non-deterministic polynomial time using a non-deterministic polynomial time oracle for decision canonization for uniform, symmetric t-ary relational structures. This is achieved by defining an isomorphism-invariant encoding of graphs into t-ary symmetric relational structures, essentially extending every graph edge to a t-ary relation by adding $t - 2$ bogus vertices. ◀

The remaining reduction is standard and can be found in Appendix C.

▶ Remark 17. In contrast, observe that the Johnson family for $t = 1$ consists of the symmetric groups $\mathrm{Sym}(n)$ in their natural action on n points. For these groups, complete symmetry breaking can be achieved with a CNF predicate of linear size (see Section 5).

4.3 Subgroups of Small Index and Large Primitive Groups

In this section, we consider symmetry breaking for a permutation group $G \leq \mathrm{Sym}(n)$ and a subgroup H of G. Mostly, we are interested in the case that H has polynomial index in G. We first show that a symmetry breaking constraint for H gives rise to symmetry breaking constraint for G:

▶ **Lemma 18.** *There exists a polynomial p such that the following holds: if there is a complete symmetry breaking circuit for a group $H \leq \mathrm{Sym}(n)$ which can be computed in time t, then complete symmetry breaking circuit with additional variables for $G \leq \mathrm{Sym}(n)$ with $G \geq H$ can be computed in time $t \cdot p(n|G : H|)$.*

Proof. Let ψ be a symmetry breaking circuit for H. We now devise a symmetry breaking circuit for G. For simplicity, we fix a system of representatives R of the right cosets of H in G, which can be computed in time polynomial in $|G : H|$ (see [26]).

Certificate. The certificate $\sigma = \{(\theta_r, h_r) \colon r \in R\}$ consists of assignments $\theta_r \colon \mathrm{Var}(\psi) \to \{0, 1\}$ and an element $h_r \in H$ for every $r \in R$.

Verifier. Given an assignment $\theta \colon \mathrm{Var}(\psi) \to \{0, 1\}$ and a certificate $\sigma = \{(\theta_r, h_r) \colon r \in R\}$, we proceed as follows:

1. For all $r \in R$, verify that $\theta_r^{h_r} = \theta^r$ holds.
2. For all $r \in R$, verify that $\psi[\theta_r]$ is satisfying. Verify that $\psi[\theta]$ is satisfying.
3. For all $r \in R$, check whether $\theta \preceq_{\mathrm{lex}} \theta_r$ holds. If this is the case, accept θ, otherwise reject.

Clearly, the runtime of this procedure is polynomial in t and $|G : H|$.

Correctness of Verifier. Let Δ be a G-orbit of assignments. Note that Δ is a disjoint union of H-orbits $\Delta_1, \dots, \Delta_k$. In each Δ_i, there exists a unique assignment α_i such that $\psi[\alpha_i]$ is satisfying. Let θ be the lexicographically minimal element in $\{\alpha_1, \dots, \alpha_k\}$. Note that we have $\theta^G = \bigcup_{r \in R} (r\theta)^H$ as every element of G can be decomposed as hr for $h \in H$ and $r \in R$. For $r \in R$, there exists $i_r \in [k]$ with $(r\theta)^H = \Delta_{i_r}$. Hence, there exists $h_r \in H$ with $\alpha_{i_r}^{h_r} = \theta^r$. By construction, θ together with the certificate $\sigma = \{(\alpha_{i_r}, h_r) \colon r \in R\}$ is accepted by the verifier.

Now suppose that $\theta, \theta' \in \Delta$ are accepted by the verifier, and let $\sigma_\theta = \{(\theta_r, h_r) \colon r \in R\}$ and $\sigma_{\theta'} = \{(\theta'_r, h'_r) \colon r \in R\}$ denote corresponding certificates. Due to the decomposition of θ^G and since $\psi[\theta_r]$ and $\psi[\theta'_r]$ are satisfying for all $r \in R$, we have $\{\alpha_1, \dots, \alpha_k\} = \{\theta_r \colon r \in R\} = \{\theta'_r \colon r \in R\}$. Since the verifier accepts both θ and θ', they coincide with the lexicographically minimal element in $\{\alpha_1, \dots, \alpha_k\}$, so $\theta = \theta'$ follows. ◀

It should be noted that while the above lemma gives a valid upper bound, the resulting SBP is not practical: The SBP simply uses the additional variables to determine the representative for all cosets, and then determines a minimal one among them. This requires trying out all the symmetric choices, defeating the purpose of the SBP. However, the result can also be read as a hardness result. For example, for the matrix models studied in Section 3, we can restrict the group on each axis of the model as follows, while still being able to retrieve our hardness result (see Theorem 13):

▶ **Corollary 19.** *Consider a family of permutation groups $G_{m,n} = X_m \times Y_n$ with $X_m \leq \mathrm{Sym}(m)$ and $Y_n \leq \mathrm{Sym}(n)$, acting component-wise on $[m] \times [n]$. Assume that $|\mathrm{Sym}(m) : X_m| < \mathsf{poly}(m)$ and $|\mathrm{Sym}(n) : Y_n| < \mathsf{poly}(n)$ holds. Then, efficient complete symmetry breaking with additional variables for $G_{m,n}$ implies $\mathsf{GI} \in \mathsf{coNP}$.*

Our main interest in studying subgroups of small index is sparked by a result on the structure of so-called large primitive groups, which forms an important building block of the quasi-polynomial isomorphism test for general graphs [5]. Roughly speaking, every primitive group $G \leq \mathrm{Sym}(n)$ with $|G| \geq n^{1+\log_2 n}$ contains a normal subgroup N with $|G : N| \leq n$ exhibiting a natural Johnson action. If the converse of Lemma 18 holds, we can thus employ our results on Johnson groups to study the complexity of symmetry breaking for large primitive groups.

5 Upper Bounds

Complementing the results from the previous sections, we show that certain families of groups can be efficiently handled. We begin by recalling three simple cases.

Groups of Polynomial Order. The first case pertains to groups where the order is polynomial in the size of the original formula. For these groups, we can explicitly write a constraint that breaks each element of the group [14]. The resulting constraint is complete and of polynomial size in the formula.

Symmetric Groups. Symmetric groups in their natural action can be handled by imposing an ordering on the assignments. For $\mathrm{Sym}(n)$, this can be achieved by the predicate $\psi_n = x_1 \leq x_2 \leq \cdots \leq x_n$.

A slight extension of symmetric groups are known and used in practice, namely row-interchangeability subgroups [17, 40]. We say that a permutation group $G \leq \mathrm{Sym}(\Omega)$ exhibits *row-interchangeability* if Ω can be arranged in an $n \times m$-matrix $X = (x_{ij})$ such that G consists precisely of the permutations of the *rows* of X. This symmetry can be broken by lexicographically ordering the rows in any assignment $\theta\colon X \to \{0,1\}$ (viewed as an $n\times m$-matrix). Formally, for $i \in [n-1]$, let $\lambda_i^k := (\bigwedge_{r\in[k-1]}(x_{ir} = x_{(i+1)r})) \to (x_{ik} \le x_{(i+1)k})$. Then $\lambda_{n,m} := \bigwedge_{i=1}^{n-1} \bigwedge_{k=1}^{m} \lambda_i^k$ is a symmetry breaking predicate for G.

Disjoint direct decomposition. A direct product $G = G_1 \times \dots \times G_r$ of permutation groups is called a *disjoint direct decomposition* if the subgroups $G_1, \dots, G_r$ have pairwise disjoint supports. Disjoint direct products naturally arise and have been successfully used in practice [17]. The finest disjoint direct decomposition can be computed in polynomial time for general permutation groups [12], and in quasi-linear time for automorphism groups of graphs [3]. For the sake of completeness, we argue that disjoint direct decompositions can be exploited without giving up on complete symmetry breaking.

▶ **Lemma 20.** *Let $G \le \mathrm{Sym}(\Omega)$ be a permutation group with a disjoint direct product decomposition $G = G_1 \times \dots \times G_r$. For $i \in [r]$, let Ω_i denote the support of G_i and assume that a complete symmetry breaking predicate γ_i for G_i, viewed as a permutation group on Ω_i, is given. In particular, we require $\mathrm{Var}(\gamma_i) \subseteq \Omega_i$. Then $\gamma := \gamma_1 \wedge \dots \wedge \gamma_r$ is a complete symmetry breaking predicate for G.*

Proof. Let $F \subseteq \Omega$ be the set of points fixed by G. Then $\Omega = \Omega_1 \dot\cup \dots \dot\cup \Omega_r \dot\cup F$. An assignment $\theta\colon \Omega \to \{0,1\}$ can be viewed as a tuple $(\theta_1, \dots, \theta_r, \theta_F)$ of assignments defined on $\Omega_1, \dots, \Omega_r, F$, respectively, and we have $\theta^G = {\theta_1}^{G_1} \times \dots \times {\theta_r}^{G_r} \times \{\theta_F\}$. Hence θ satisfies γ if and only if θ_i satisfies γ_i for every $i \in [r]$. Thus, γ is a complete symmetry breaking predicate for G. ◀

The size of the constraint γ is linear in the size of the constraints $\gamma_1, \dots, \gamma_r$.

Wreath Products. We now turn our attention to so-called *wreath products*. They naturally occur as the automorphism groups of tree-like structures and can be detected as the induced action on a block system [45]. Tree-like appendages are already detected and exploited by practical symmetry detection algorithms [4], and thus these wreath products seem readily available.

Indeed, certain cases of wreath products can be efficiently handled in symmetry breaking. Specifically, we show that automorphism groups of trees can be taken care of (see Theorem 4).

Intuitively, a wreath symmetry occurs if the domain can be partitioned into equally-sized parts with identical symmetries that can be permuted among each other. The corresponding symmetry group is made of a group describing the possible permutations of the points within a part, and a group describing the permutation of the parts. Formally, let $G \le \mathrm{Sym}(n)$ and $H \le \mathrm{Sym}(m)$, and consider the imprimitive action of $G \wr H$ on $X := \{x_{ij} \colon i \in [n],\, j \in [m]\}$. Explicitly, it is given by $x_{ij}^{((g_1,\dots,g_m),h)} = x_{g_{h(j)}(i)\,h(j)}$. For $\theta\colon X \to \{0,1\}$ and $j \in [m]$, let $\theta_j := \theta|_{\{x_{1j},\dots,x_{nj}\}}$ and write $\theta = (\theta_1, \dots, \theta_m)$.

Wreath Products with CNF. Let us first focus on CNF predicates. Recall the predicate $\lambda_{m,n}$ from the beginning of this section. The following result shows that a symmetry breaking predicate for a permutation group G can be "lifted" to a predicate for $G \wr \mathrm{Sym}(m)$:

▶ **Lemma 21.** *Assume that γ is a complete symmetry breaking predicate for $G \le \mathrm{Sym}(n)$ and set $\gamma_j := \gamma(x_{1j}, \dots, x_{nj})$ for all $j \in [m]$. Then $\nu := \bigwedge_{j\in[m]} \gamma_j \wedge \lambda_{m,n}$ is a complete symmetry breaking predicate for $W := G \wr \mathrm{Sym}(m)$.*

Proof. Let $\theta\colon X \to \{0,1\}$ be an assignment. For every $j \in [m]$, there exists $g_j \in G$ such that $\theta_j^{g_j}$ satisfies γ. Write $\theta' := \theta^{((g_1,\dots,g_m),1)}$. There exists $h \in \mathrm{Sym}(m)$ with $\theta'_{h^{-1}(1)} \preceq_{\mathrm{lex}} \cdots \preceq_{\mathrm{lex}} \theta'_{h^{-1}(m)}$. Hence, the assignment $\theta'^{(1,h)} \in \theta^W$ satisfies ν.

On the other hand, consider assignments $\theta, \theta'\colon X \to \{0,1\}$ satisfying ν, and assume $\theta' = \theta^{((g_1,\dots,g_m),h)}$ for $\big((g_1,\dots,g_m),h\big) \in W$. For all $j \in [m]$, this implies $\theta'_j = \theta_{h^{-1}(j)}^{g_{h^{-1}(j)}} \in \theta_{h^{-1}(j)}{}^G$. As θ'_j and $\theta_{h^{-1}(j)}$ satisfy γ, they coincide, so we may choose $g_1 = \cdots = g_m = 1$. Since θ and θ' satisfy $\lambda_{m,n}$, we have $\theta_1 \preceq_{\mathrm{lex}} \cdots \preceq_{\mathrm{lex}} \theta_m$ and $\theta'_1 \preceq_{\mathrm{lex}} \cdots \preceq_{\mathrm{lex}} \theta'_m$. This yields $\theta'_j = \theta_j$ for all $j \in [m]$, so $\theta = \theta'$ follows. ◀

▶ Remark 22. The size of the predicate ν given in Lemma 21 is in $\mathcal{O}(s(\gamma)m + nm)$, where $s(\gamma)$ denotes the size of γ. Note that if $s(\gamma) \in \mathcal{O}(n)$ holds, then the size of ν is linear in the domain size nm of the wreath product.

▶ **Corollary 23.** *The predicate* $\nu = \bigwedge_{j\in[m]}(x_{1j} \leq \cdots \leq x_{nj}) \wedge \lambda_{m,n}$ *is a complete symmetry breaking predicate for* $\mathrm{Sym}(n) \wr \mathrm{Sym}(m)$*.*

Combining the results for direct disjoint decompositions and wreath products, it readily follows that automorphism groups of trees can be handled efficiently (see Theorem 4).

Proof of Theorem 4. The group G can be constructed by iterated disjoint direct decompositions and wreath products in which the top group is a full symmetric group [41]. Combining Lemma 20 and Proposition 21 thus yields a symmetry breaking predicate for G. Inductively, it follows from Remark 22 that the size of this predicate is linear. ◀

Wreath Products with Circuits. Using circuits and a few further assumptions, we treat general wreath products $W := G \wr H$.

▶ **Theorem 5.** *Let* $G \leq \mathrm{Sym}(n)$ *and* $H \leq \mathrm{Sym}(m)$ *be permutation groups. Assume that a complete symmetry breaking circuit for* G *can be computed in polynomial time. Moreover, suppose that for every partition* P *of* $[m]$*, the partition stabilizer* S *of* P *in* H *and a complete symmetry breaking circuit for* S *can be computed in polynomial time. Then there is a complete symmetry breaking circuit for the wreath product* $W := G \wr H$ *that can be computed in polynomial time.*

Proof sketch. It suffices to give a polynomial-time algorithm for symmetry breaking for W. Let $\theta\colon X \to \{0,1\}$ be an assignment and write $\theta = (\theta_1,\dots,\theta_m)$ as before. We obtain the elements of $\theta^{G\wr H}$ by applying elements of G to $\theta_1,\dots,\theta_m$ and permuting the resulting strings using an element of H.

Based on this, the representative $\theta = (\theta_1,\dots,\theta_m)$ of an orbit is chosen as follows: We may require that all of $\theta_1,\dots,\theta_m$ are accepted by the symmetry breaking circuit for G. It remains to fix a unique permutation of $\theta_1,\dots,\theta_m$ under an element of H. There, we proceed in a way resembling the lexicographic ordering used in the proof of Lemma 21: We require that the symmetry breaking circuit for H accepts the string s_1 consisting of the first entries of $\theta_1,\dots,\theta_m$. There might still be multiple permutations of $\theta_1,\dots,\theta_m$ with the same string of first entries. We thus proceed with the string of second entries s_2, but only taking those permutations in H into account that fix the string of first entries. Thus we require that the symmetry breaking circuit for $H \cap S_1$ accepts s_2, where S_1 is the stabilizer of s_1. The latter can be expressed as a partition stabilizer. We continue this way up to the last position and accept θ as the orbit representative if none of the intermediate verifications fails. ◀

The full proof of this theorem is a mere formalization of this idea. A detailed description as well as a discussion of correctness can be found in Appendix B. In general, it is unknown whether partition stabilizers can be efficiently computed (see [26]). However, for $H = \mathrm{Sym}(m)$, the stabilizer of $P = (\Omega_1, \ldots, \Omega_r)$ is simply given by $\mathrm{Sym}(\Omega_1) \times \cdots \times \mathrm{Sym}(\Omega_r) \leq \mathrm{Sym}(n)$, and hence readily computable. This also holds if the order of H is small. There, we obtain the following consequence of the preceding result:

▶ **Corollary 24.** *Let $G \leq \mathrm{Sym}(n)$ and $H \leq \mathrm{Sym}(m)$ be permutation groups. Assume that a complete symmetry breaking circuit for G can be computed in polynomial time and that $|H| \in \mathsf{poly}(n, m)$ holds. Then a complete symmetry breaking circuit for $G \wr H$ can be computed in polynomial time.*

6 Conclusion and Future Work

We laid the foundation for a systematic study of the complexity of symmetry breaking. A central tool in our investigation was the relation to the string canonization problem (see Section 3). In particular, we showed that polynomial time symmetry breaking for the row-column symmetry group, even with circuits and additional variables, implies $\mathsf{GI} \in \mathsf{coNP}$ (see Theorem 1). The same applies to proper Johnson actions (see Theorem 2). On the other hand, we showed that symmetry breaking in polynomial time is possible for several families of groups, including certain classes of wreath products (see Section 5).

Clearly, the dividing line between permutation groups that are "hard" instances for symmetry breaking, and those which admit efficient symmetry breaking, needs to be further explored. Among others, the following questions immediately arise:

1. Given a permutation group G and a subgroup H of polynomial index, does H admit efficient symmetry breaking if G does (i.e., does the converse of Lemma 18 hold)?
2. How difficult are permutation groups of intermediate size, in particular groups of quasi-polynomial order in the size of the domain?

A positive answer to Question 1 would not only contribute to further decomposition results, but it is particularly relevant as large primitive permutation groups are known to contain normal subgroups of small index exhibiting a Johnson action.

References

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

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

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

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

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

6 László Babai. Canonical form for graphs in quasipolynomial time: preliminary report. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 1237–1246. ACM, 2019. doi:10.1145/3313276.3316356.

7 László Babai and Eugene M. Luks. Canonical labeling of graphs. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 171–183, New York, NY, USA, 1983. Association for Computing Machinery.

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

9 László Babai and Shlomo Moran. Arthur-merlin games: A randomized proof system, and a hierarchy of complexity classes. *Journal of Computer and System Sciences*, 36(2):254–276, 1988. doi:10.1016/0022-0000(88)90028-1.

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

12 Mun See Chang and Christopher Jefferson. Disjoint direct product decompositions of permutation groups. *J. Symb. Comput.*, 108:1–16, 2022. doi:10.1016/j.jsc.2021.04.003.

13 Michael Codish, Graeme Gange, Avraham Itzhakov, and Peter J. Stuckey. Breaking symmetries in graphs: The nauty way. In *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2016. doi:10.1007/978-3-319-44953-1_11.

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

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

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

17 Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 104–122. Springer, 2016. doi:10.1007/978-3-319-40970-2_8.

18 Jo Devriendt, Bart Bogaerts, Broes De Cat, Marc Denecker, and Christopher Mears. Symmetry propagation: Improved dynamic symmetry breaking in SAT. In *IEEE 24th International*

Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012, pages 49–56. IEEE Computer Society, 2012. doi:10.1109/ICTAI.2012.16.

19 John D. Dixon and Brian Mortimer. *Permutation Groups*. Graduate Texts in Mathematics. Springer New York, 1996. URL: https://books.google.de/books?id=4QDpFN6k61EC.

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

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

22 Pierre Flener, Justin Pearson, and Meinolf Sellmann. Static and dynamic structural symmetry breaking. *Ann. Math. Artif. Intell.*, 57(1):37–57, 2009. doi:10.1007/S10472-009-9172-3.

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

24 Andrew Grayland, Chris Jefferson, Ian Miguel, and Colva M. Roney-Dougal. Minimal ordering constraints for some families of variable symmetries. *Annals of Mathematics and Artificial Intelligence*, 57:75–102, 2009.

25 Marijn J. H. Heule. Optimal symmetry breaking for graph problems. *Math. Comput. Sci.*, 13(4):533–548, 2019. doi:10.1007/S11786-019-00397-5.

26 D.F. Holt, B. Eick, and E.A. O'Brien. *Handbook of Computational Group Theory*. Discrete Mathematics and Its Applications. CRC Press, 2005. URL: https://books.google.de/books?id=i2UjAASZ33YC.

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

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

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

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

31 Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Progress in Theoretical Computer Science. Birkhäuser/Springer, 1993. doi:10.1007/978-1-4612-0333-9.

32 Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25(1):42–65, 1982. doi:10.1016/0022-0000(82)90009-5.

33 Eugene M. Luks. Hypergraph isomorphism and structural equivalence of boolean functions. In Jeffrey Scott Vitter, Lawrence L. Larmore, and Frank Thomson Leighton, editors, *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, May 1-4, 1999, Atlanta, Georgia, USA*, pages 652–658. ACM, 1999. doi:10.1145/301250.301427.

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

35 François Margot. Pruning by isomorphism in branch-and-cut. *Math. Program.*, 94(1):71–90, 2002. doi:10.1007/S10107-002-0358-2.

36 Brendan D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26(2):306–324, 1998. doi:10.1006/JAGM.1997.0898.

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

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

39 James Ostrowski, Jeff T. Linderoth, Fabrizio Rossi, and Stefano Smriglio. Constraint orbital branching. In *Integer Programming and Combinatorial Optimization, 13th International Conference, IPCO 2008, Bertinoro, Italy, May 26-28, 2008, Proceedings*, volume 5035 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2008. doi:10.1007/978-3-540-68891-4_16.

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

41 G. Pólya. Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen. *Acta Mathematica*, 68(none):145 – 254, 1937. doi:10.1007/BF02546665.

42 Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In William J. Clancey and Daniel S. Weld, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 1*, pages 209–215. AAAI Press / The MIT Press, 1996. URL: http://www.aaai.org/Library/AAAI/1996/aaai96-031.php.

43 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.

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

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

46 Toby Walsh. On the complexity of breaking symmetry. *CoRR*, abs/2005.08954, 2020. arXiv:2005.08954.

47 Viktor N Zemlyachenko, Nickolay M Korneenko, and Regina I Tyshkevich. Graph isomorphism problem. *Journal of Soviet Mathematics*, 29:1426–1481, 1985.

Appendix

A Proof of Theorem 13

Proof. It suffices to show that there exists a canonical form f such that d-GCANON$_f \in \mathsf{NP}$ (see Lemma 12).

Certificate. Given a graph $\Gamma = (V, E)$, we consider the row-column symmetry group with $n = |V|$ rows and $m = |E|$ columns. More precisely, we assume $V = [n]$. We let $\delta_{n,m}(x_{11}, \dots, x_{nm}, y_1, \dots, y_p)$ denote the symmetry breaking circuit as computed by the polynomial time algorithm for row-column symmetry. Obviously, $p \in \text{poly}(m, n)$. Our chosen certificate for decision canonization is σ, where σ is an assignment to the variables $V = \text{Var}(\delta_{n,m}(x_{11}, \dots, x_{nm}, y_1, \dots, y_p))$.

Bipartite Reordering. We observe a technicality of our reduction: d-GCANON expects as input a graph, whereas our row-column symmetry breaking circuits essentially solve the decision canonization for bipartite graphs. We proceed using the same encoding for graphs into bipartite graphs as discussed previously (see proof of Theorem 1). The order of a graph Γ is fully determined by the order of its vertices. However, observe that when only restricting the order of the left partition, the corresponding bipartite graph bip(Γ) may still differ in the order of the edges, i.e., in the order of the *right* partition. Indeed, the symmetry breaking circuit may choose to accept any of these orderings of the right partition. We define the set bip$'$(Γ) of bipartite graphs, where the left partition is ordered according to $V(\Gamma)$, and all potential reorderings of the right partition are contained. Note that $\bigcup_{\Delta \in \Gamma^{\text{Sym}(V(\Gamma))}} \text{bip}'(\Delta)$ covers all reorderings of the corresponding bipartite graphs.

Verifier. Given a certificate σ, our polynomial time verifier proceeds as follows:

1. For each column c in the matrix of x_{ij} variables, verify that the assignment has precisely two true variables in the column c. Formally, there exist $i, j \in [n]$ with $i \neq j$ such that $\sigma(x_{ic}) = 1$ and $\sigma(x_{jc}) = 1$, and for all $k \in [n]$ with $j \neq k \neq i$ it holds that $\sigma(x_{ic}) = 0$.
2. Check that Γ corresponds to the bipartite graph as given by the assignment σ: for each edge $\{v_1, v_2\} \in E$, we verify that there exists a column c such that $x_{v_1 c} = 1$ and $x_{v_2 c} = 1$.
3. Check if σ satisfies $\delta_{n,m}(x_{11}, \cdots, x_{nm}, y_1, \dots, y_p)$. If it satisfies $\delta_{n,m}$, accept, otherwise reject.

By assumption, $\delta_{n,m}(x_{11}, \dots, x_{nm}, y_1, \dots, y_p)$ can be computed in polynomial time. Clearly, all the other steps can be computed in polynomial time as well.

Correctness of Verifier. We need to argue that for each isomorphism class of graphs $\Gamma^{\text{Sym}(V(\Gamma))}$, there is precisely one ordered graph G accepted by the verifier.

First, assume towards a contradiction that there is a Γ such that for all $\varphi \in \text{Sym}(V(\Gamma))$, and all certificates, Γ^φ is rejected by the verifier. Consider the corresponding bipartite graph bip(Γ). By assumption, we know that for all orderings of the left partition, all orderings of the right partition are rejected by the verifier. Hence, all orderings of bip(Γ) under $\text{Sym}([n]) \times \text{Sym}([m])$ are rejected by the symmetry breaking circuit. Hence, $\delta_{n,m}$ can not be a correct symmetry breaking circuit, which is a contradiction.

Next, assume towards a contradiction that there are two distinct isomorphic graphs $\Gamma \cong \Delta$ which are both accepted by the verifier. Since $\delta_{n,m}$ is a correct symmetry breaking circuit, this may only occur if there are corresponding bipartite graphs $\Gamma^* \in \text{bip}'(\Gamma), \Delta^* \in \text{bip}'(\Delta)$ such that $\Gamma^* = \Delta^*$. However, this would immediately imply $\Gamma = \Delta$. ◀

B Proof of Theorem 5

▶ **Theorem 5.** *Let $G \leq \mathrm{Sym}(n)$ and $H \leq \mathrm{Sym}(m)$ be permutation groups. Assume that a complete symmetry breaking circuit for G can be computed in polynomial time. Moreover, suppose that for every partition P of $[m]$, the partition stabilizer S of P in H and a complete symmetry breaking circuit for S can be computed in polynomial time. Then there is a complete symmetry breaking circuit for the wreath product $W := G \wr H$ that can be computed in polynomial time.*

Proof. Since we may turn a polynomial time algorithm into a polynomial-sized circuit, it suffices to give a polynomial-time algorithm for symmetry breaking for W.

Let ψ_G denote the symmetry breaking circuit for G, and for any partition stabilizer S in H, write ψ_S for the corresponding symmetry breaking circuit. For an assignment $\theta\colon X \to \{0,1\}$, write $\theta = (\theta_1, \ldots, \theta_m)$ as before. For $i \in [n]$, let $c_i(\theta)$ be the string of length m consisting of the i-th entries of $\theta_1, \ldots, \theta_m$. We define partitions $P_1, \ldots, P_n$ of $[m]$ as follows: let P_1 denote the partition into the index sets of zero and one entries in $c_1(\theta)$. For $i \geq 2$, P_i is the refinement of P_{i-1} according to the zero-one-partition of $c_i(\theta)$. For $i \in [n]$, let S_i denote the partition stabilizer of P_i in H, and set $S_0 := H$.

Description of Algorithm. Given an assignment $\theta = (\theta_1, \ldots, \theta_m)$, we define our algorithm as follows:

1. If $\psi_G[\theta_i]$ is non-satisfying for some $i \in [m]$, return false.
2. For $i \in [n]$, compute the vectors $c_i(\theta)$ as well as the partitions P_i and their stabilizers S_i.
3. For $i \in [n]$, check if $\psi_{S_{i-1}}[c_i(\theta)]$ is satisfying. If this fails for some $i \in [n]$, return false. Otherwise, return true.

Correctness of Algorithm. By assumption, partition stabilizers in H as well as all the necessary symmetry breaking circuits can be computed in polynomial time. The remaining steps of the algorithm can clearly be computed in polynomial time.

Replacing the input assignment $\theta = (\theta_1, \ldots, \theta_m)$ by some element $\theta^{((g_1,\ldots,g_m),1)} \in \theta^W$, we may assume that $\psi_G[\theta_1], \ldots, \psi_G[\theta_m]$ are satisfying. By assumption, there exists $h_1 \in H$ such that $\psi_H[c_1(\theta^{(1,h_1)})]$ is satisfying. Moreover, there exists $h_2 \in S_1$ such that $\psi_{S_1}[c_2(\theta^{(1,h_2h_1)})]$ is satisfying. Note that $c_1(\theta^{(1,h_2h_1)}) = c_1(\theta^{(1,h_1)})$ holds due to $h_2 \in S_1$. Continuing, we obtain an element $\theta' := \theta^{(1,h_{n-1}\cdots h_1)} \in \theta^W$ for which the algorithm returns true.

On the other hand, suppose that $\theta = (\theta_1, \ldots, \theta_m)$ and $\theta' = (\theta'_1, \ldots, \theta'_m)$ are assignments in the same W-orbit accepted by the algorithm. Then $\psi_G[\theta_i]$ and $\psi_G[\theta'_i]$ are satisfying for all $i \in [m]$. Since θ and θ' lie in the same W-orbit, the strings $\theta_1, \ldots, \theta_m$ and $\theta'_1, \ldots, \theta'_m$ coincide up to reordering. The ordering of the substrings is lexicographic with respect to a successive application of H. This yields $\theta = \theta'$. ◀

C Proof of Theorem 16

▶ **Theorem 16.** *Let $t \geq 2$ be a fixed arity. Suppose there exists a polynomial time algorithm for generating complete symmetry breaking circuits with additional variables for the Johnson family of arity t. Then, d-GCANON $\in$ NP and hence GI $\in$ coNP.*

Proof. Again, Lemma 12 ensures that it suffices to prove d-GCANON $\in$ NP. From Lemma 14, it follows that it suffices to solve d-GCANON in non-deterministic polynomial time using a non-deterministic polynomial time oracle for decision canonization for uniform, symmetric t-ary relational structures.

Graph to t-ary Structure. Given a graph $\Gamma = (V, E)$, we define a t-uniform relational structure R_Γ as follows. Let $I \subseteq V$ be the set of isolated vertices. We have $V(R_\Gamma) = \{r_u : u \in V\} \cup \{v_1, \ldots, v_{t-2}, a, b\}$. Observe that we added t vertices, namely $v_1, \ldots, v_{t-2}, a, b$. These vertices will be called *bogus vertices*. We presume the order $r_{v_1} \prec \ldots \prec r_{v_n} \prec v_1 \prec \ldots \prec v_{t-2} \prec a \prec b$ for the symbols used in the construction. The hyperedges in R_Γ are given by

$$\Big\{\{r_u, r_w, v_1, \ldots, v_{t-2}\} : \{u, w\} \in E\Big\} \cup \Big\{\{r_u, v_1, \ldots, v_{t-2}, a\} : u \in V \setminus I\Big\} \\ \cup \Big\{\{v_1, \ldots, v_{t-2}, a, b\}\Big\}.$$

Observe that

$$\begin{cases} \deg_{R_\Gamma}(r_u) = 0, & u \in I \\ \deg_{R_\Gamma}(r_u) = \deg_\Gamma(u) + 1, & u \in V \setminus I \\ \deg_{R_\Gamma}(v_i) = |E| + |V \setminus I| + 1, & i \in [t-2] \\ \deg_{R_\Gamma}(a) = |V \setminus I| + 1, & \\ \deg_{R_\Gamma}(b) = 1. & \end{cases}$$

In particular, for $u \in V \setminus I$, we have $1 < \deg_{R_\Gamma}(r_u) \leq |V \setminus I|$.

Now let Γ and Δ be graphs on n vertices. Without loss of generality, we may assume that Γ and Δ contain edges. We claim that Γ and Δ are isomorphic precisely if R_Γ and R_Δ are isomorphic. Assume that there exists an isomorphism $\varphi : R_\Gamma \to R_\Delta$. Denote the vertices in R_Γ and R_Δ with an exponent Γ and Δ, respectively. By the above degree conditions, we have $\varphi(b^\Gamma) = b^\Delta$ (here, the notation b^Γ refers to node b of graph Γ). As b^Γ is adjacent to $v_1^\Gamma, \ldots, v_{t-2}^\Gamma, a^\Gamma$ (similarly in Δ), the degree conditions then imply $\varphi(a^\Gamma) = a^\Delta$. Now the vertices $v_1^\Gamma, \ldots, v_{t-2}^\Gamma$ are mapped bijectively to $v_1^\Delta, \ldots, v_{t-2}^\Delta$. In particular, φ induces a bijection between $\{r_u^\Gamma : u \in V(\Gamma)\}$ and $\{r_u^\Delta : u \in V(\Delta)\}$. It is then easy to see that φ induces an isomorphism between Γ and Δ.

On the other hand, if Γ and Δ are isomorphic, $R_\Gamma \cong R_\Delta$ follows from the fact that the above construction is isomorphism-invariant: all additional bogus vertices universally appear with all edges, as well as with all non-isolated vertices.

Furthermore, it is easy to see that if $\Gamma \neq \Delta$, then $R_\Gamma \neq R_\Delta$ follows.

Certificate. Our certificate will consist of a permutation $\varphi \in \mathrm{Sym}(V(R_\Gamma))$, as well as a certificate for decision canonization of t-ary structures σ.

Verifier. Our verifier proceeds as follows:

1. Using the decision canonization oracle for t-ary structures, continue if σ is a valid certificate for R_Γ^φ, and reject otherwise.
2. If for all pairs of vertices $v, v' \in V(\Gamma)$ with $v \prec v'$ it holds that $\varphi(r_v) \prec \varphi(r_{v'})$, accept, otherwise reject.

Correctness of Verifier. From the arguments above, we know that for all graphs Δ in the isomorphism class of Γ it holds that $R_\Gamma \cong R_\Delta$. The oracle in Step 1 will accept precisely one canonical t-ary structure R_Γ^φ in the isomorphism class of R_Γ. In turn, the verifier accepts a graph Γ, if and only if the order of the vertices is preserved in the canon R_Γ^φ (see Step 2). We remark that there may also be different φ' which map R_Γ to the canon, which may not preserve the order of $V(\Gamma)$. Clearly, there is at least one graph Γ in each isomorphism class that is accepted by the verifier.

Assume there is another graph $\Delta \neq \Gamma$ with $\Delta \cong \Gamma$ which is also accepted by the verifier. Since we know that $R_\Delta \cong R_\Gamma$ holds, this means there is a $\varphi' \in \mathrm{Sym}(V(R_\Delta))$ such that $R_\Delta^{\varphi'} = R_\Gamma^{\varphi}$ holds. In particular, φ' preserves the order of vertices in Δ. Recall that bogus vertices can only ever be mapped to bogus vertices. Therefore, $R_\Delta^{\varphi'} = R_\Gamma^{\varphi}$ immediately implies that the vertices of Γ and Δ can be mapped, in order, onto each other, while preserving the edge relation of the original graphs. In other words, $\Delta = \Gamma$ holds, which is a contradiction to the assumption that the verifier accepts Δ. ◀

Certifying Without Loss of Generality Reasoning in Solution-Improving Maximum Satisfiability

Jeremias Berg
Department of Computer Science, HIIT, Helsinki, Finland
University of Helsinki, Finland

Bart Bogaerts
Vrije Universiteit Brussel, Belgium

Jakob Nordström
University of Copenhagen, Denmark
Lund University, Sweden

Andy Oertel
Lund University, Sweden
University of Copenhagen, Denmark

Tobias Paxian
University of Freiburg, Germany

Dieter Vandesande
Vrije Universiteit Brussel, Belgium

Abstract

Proof logging has long been the established method to certify correctness of Boolean satisfiability (SAT) solvers, but has only recently been introduced for SAT-based optimization (MaxSAT). The focus of this paper is solution-improving search (SIS), in which a SAT solver is iteratively queried for increasingly better solutions until an optimal one is found. A challenging aspect of modern SIS solvers is that they make use of complex "without loss of generality" arguments that are quite involved to understand even at a human meta-level, let alone to express in a simple, machine-verifiable proof.

In this work, we develop pseudo-Boolean proof logging methods for solution-improving MaxSAT solving, and use them to produce a certifying version of the state-of-the-art solver Pacose with VeriPB proofs. Our experimental evaluation demonstrates that this approach works in practice. We hope that this is yet another step towards general adoption of proof logging in MaxSAT solving.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Constraint and logic programming; Mathematics of computing → Combinatorial optimization

Keywords and phrases proof logging, certifying algorithms, MaxSAT, solution-improving search, SAT-UNSAT, maximum satisfiability, combinatorial optimization, certification, pseudo-Boolean

Digital Object Identifier 10.4230/LIPIcs.CP.2024.4

Supplementary Material *Dataset (experimental data and source code)*: `https://zenodo.org/records/12591387` [8]

Funding *Jeremias Berg*: Research Council of Finland under grants 342145.
Jakob Nordström: Swedish Research Council grant 2016-00782 and Independent Research Fund Denmark grant 9040-00389B.
Andy Oertel: Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.
Dieter Vandesande: Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G070521N).

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 4; pp. 4:1–4:28
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Acknowledgements We want to thank Florian Pollitt and Mathias Fleury for their assistance with the CaDiCaL proof tracer and for fuzzing VeriPB within CaDiCaL. Their contributions were very helpful to further improve the robustness of the VeriPB toolchain. We also wish to acknowledge useful discussions with participants of the Dagstuhl workshop 23261 *SAT Encodings and Beyond*. The computational experiments were enabled by resources provided by LUNARC at Lund University.

1 Introduction

Thanks to tremendous progress over the last decades on algorithms for combinatorial search and optimization, today *NP*-hard problems are routinely solved in many practical applications. Unfortunately, as these algorithms get more and more sophisticated, it also gets more and more challenging to avoid errors sneaking in during algorithm design and implementation. It is well-known that modern combinatorial solving algorithms in different paradigms can sometimes produce "solutions" that violate hard constraints, claim that suboptimal solutions are optimal, or declare that feasible problems lack solutions [9, 15, 16, 19, 30, 39].

Although there are many ways to address this problem, including software testing techniques such as fuzzing [15, 50], and design of formally verified software [28], the most promising approach appears to be the use of *certifying algorithms* [1, 48] with so-called *proof logging*. What this means is the algorithm should not only produce an answer, but also a *proof* that this answer is correct. Such proofs should follow simple rules, as specified by a formal *proof system*, so that they can easily be verified by an independent *proof checker*. In addition to guaranteeing correctness, proof logging brings many other advantages: it enables advanced *testing* (since one can detect correct answers found for invalid reasons, and also test instances for which the answer is not known), detailed *debugging* (since invalid proof steps pinpoint where errors happened), *auditability* (since proofs can be stored and verified independently of which algorithm was used), and *performance analysis* (since proofs can be mined for insights on which reasoning steps were crucial for reaching the final conclusion).

Proof logging has been particularly successful in the domain of Boolean satisfiability (SAT) solving [11], where a large variety of proof systems has seen the light of day [4, 10, 35, 63]. Using proof logging has long been mandatory in the main track of the SAT competitions, and it is hard to overestimate the impact this has had on improving overall correctness and reliability of SAT solvers. This has stimulated the spread of proof logging into other combinatorial solving paradigms, including SAT modulo theories (SMT) [7, 57], automated planning [25–27, 56], and mixed integer linear programming [21, 24].

1.1 Proof Logging for MaxSAT Solving

In view of the above discussion, it is interesting to compare the developments in other combinatorial optimization paradigms to the state of affairs in maximum satisfiability (MaxSAT), the optimization version of the SAT problem. Without loss of generality, MaxSAT can be described as the problem of maximizing a linear objective O subject to satisfying a Boolean formula F in conjunctive normal form (CNF). Although MaxSAT is arguably the one optimization paradigm closest to SAT, and although several proof systems for formalizing MaxSAT reasoning have been proposed [14, 42, 49, 53–55], for a long time there has been no practically feasible proof logging method for state-of-the-art MaxSAT solvers. This changed only recently when pseudo-Boolean proof logging using VeriPB [12, 34] was proposed for MaxSAT [59, 60], a proposal that was followed by the successful design and implementation of VeriPB proof logging for modern core-guided MaxSAT solvers [9].

In this paper, we revisit proof logging work for *solution-improving search (SIS)* [59, 60], also referred to as *model-improving search* or *linear SAT-UNSAT (LSU) search*, and consider state-of-the-art solving techniques. In the SIS approach – which is much simpler to explain than, e.g., core-guided [29] or implicit hitting set [20] search – a SAT solver is repeatedly called on the formula F, each time with an added *solution-improving constraint* asking for increasingly better solutions with respect to the objective O, and the problem turns infeasible when the last solution found was optimal. In the work by Vandesande et al. [59, 60], the main technical challenge was to certify correctness of the CNF encodings of these solution-improving constraints, which could then essentially be concatenated with the proof logging generated by the SAT solver (modulo some non-trivial engineering).

At first sight, it seems that implementing pseudo-Boolean proof logging in a state-of-the-art MaxSAT solver using solution-improving search would mostly be a matter of carefully transferring already developed techniques [59, 60], perhaps combining them with proof logging ideas developed for other CNF encodings [31]. After all, the distinguishing feature of a top-of-the-line SIS solver is the choice of CNF translation for reasoning about the objective function, such as, in the case of Pacose, the *polynomial watchdog (DPW)* encoding [6]. Once proof logging for such a CNF encoding is in place, it seems reasonable to expect that the rest should be plain sailing.

It is all the more surprising, then, that it turns out nothing could be further from the truth. To minimize the time the MaxSAT solver spends on generating PW encodings, an essential step is to introduce completely unconstrained variables that can be used to perform different comparisons with a single CNF encoding; this is referred to as the *dynamic polynomial watchdog encoding (DPW)* [52]. Loosely speaking, if we know that the best possible objective value lies in the range $[lo, hi]$, then instead of generating repeated encodings $O \geq V$ to probe different possible objective values V in this range, one can introduce free variables t_i encoding a tare sum T taking values between 0 and $hi - lo$ and try to maximize the value $T = T^*$ for which one single DPW-encoded constraint $O - T \geq lo$ holds. Once the maximum T^* has been found, it is clear that $O = lo + T^*$ is the best possible objective value, since without loss of generality T could be set to any value. But how can such a meta-argument be expressed in simple propositional logic reasoning?

In what follows, we provide a brief, if still high-level, discussion of some of the challenges that arise when trying to design simple proofs to certify such fairly complex "without loss of generality" arguments, and then outline how such challenges can be overcome.

1.2 Solution-Improving "Without Loss of Generality" Reasoning

As already discussed above, the key aspect in which different solution-improving MaxSAT solvers differ is how they encode the solution-improving constraints. In order to compute the value of a linear expression L over 0–1 variables of interest, Pacose uses the polynomial watchdog encoding to describe a Boolean circuit BC with output variables z_k such that $z_k = 0$ implies $L \geq 1 + k \cdot 2^P$ (for some fixed integer P). If we chose L to be the objective function O that we are maximizing, this would allow to find the interval $\left[1 + k^* \cdot 2^P, (k^* + 1) \cdot 2^P\right]$ in which the optimal value lies by calling the SAT solver with the prechosen partial assignment $z_k = 0$ (referred to as an *assumption*) for increasing values of k until the solver returns that there is no satisfying assignment. To determine the exact location of the optimum in this interval, additional, completely unconstrained, variables t_i, called *tare variables*, are used to encode an integer $T = \sum_{i=0}^{P-1} 2^i t_i$ in the range $\left[0, 2^P - 1\right]$. The actual circuit in the encoding uses the linear form $L = O - T$, so that $z_k = 0$ means $O - T \geq 1 + k \cdot 2^P$. By making SAT solver calls with suitable assumptions on the unconstrained t_i-variables, the optimal value of the objective function can be computed.

Given the CNF encoding of a circuit $\mathsf{BC}\big(O - T \geq 1 + k \cdot 2^P\big)$ evaluating the inequality $O - T \geq 1 + k \cdot 2^P$ as outlined above, the solution-improving search proceeds in two phases:

(i) The *coarse convergence phase* identifies the largest k for which $z_k = 0$ is possible.

(ii) The *fine convergence phase* then maximizes the tare variable sum T.

Let us discuss this process in slightly more detail, and explain why it presents challenges from a proof logging point of view.

If during the coarse convergence phase a SAT solver call with assumption $z_k = 0$ returns a satisfying assignment α achieving objective value at least $1 + k \cdot 2^P$, the solver stores the information $z_k = 0$ (in the form of a unit clause $\overline{z}_k$), which enforces that any future solutions found have to be at least this good. The SAT solver is then called again with $z_{k'} = 0$ for some $k' > k$ to probe whether a solution exists with value at least $1 + k' \cdot 2^P$. Here it is relevant to note that fixing $z_k = 0$ could remove assignments corresponding to optimal solutions. For instance, if the optimal value is $V = V^* + 1 + k \cdot 2^P$, this value could be achieved by an assignment α' setting $T = T^* > V^* + 1$. For such an α' we would have $O - T = -T^* + V^* + 1 + k \cdot 2^P \leq k \cdot 2^P$, which would violate $z_k = 0$. However, since the tare variables are unconstrained, in this case there would also exist another assignment α'' achieving objective value $V^* + k \cdot 2^P$ for which $T = 0$, and so it is safe to require that solutions improving on α should satisfy $z_k = 0$.

In the fine convergence phase the z_k-variables are all fixed, and assumptions on the tare variables are made in the SAT solver calls to determine the exact value of the optimal solution. This again relies on reasoning without loss of generality, claiming that one can always choose $T \geq s$ for any value $0 \leq s < 2^P$. But now we are treading on dangerous ground: clearly, we cannot assume both $T = 0$ and $T \geq s > 0$ simultaneously! How can we convince ourselves, and more importantly, how can we convince a proof checker, that our derivations are consistent? At a meta-level, we can argue that since the tare variables are completely unconstrained in the original encoding, we should be able to fix them to any value we like at any given point in time. But how do we produce a simple, machine-verifiable proof that this is sound? And are we even sure this is sound?

1.3 Discussion of Our Contribution

In this work, we show how pseudo-Boolean proof logging with VeriPB [12, 34] can certify correctness of the complex CNF encodings used in state-of-the-art solution-improving MaxSAT solvers, as well as of the subtle without loss of generality reasoning applied on these encodings. To give a sense of how this can be done, we need to give a high-level description how VeriPB proofs work (referring the reader to later sections for the missing technical details).

A VeriPB proof maintains a set of *core constraints* $\mathcal{C}$, initialized to the formula F, together with a set of *derived constraints* $\mathcal{D}$ inferred by the solver. The proof semantics ensures that $\mathcal{C}$ and F have the same optimal value for O and that any solution to $\mathcal{C}$ can be extended to $\mathcal{D}$. A new constraint C can be derived "without loss of generality" by the *redundance-based strengthening rule*, which requires the explicit specification of a substitution ω (mapping variables to truth values or literals) together with explicit proofs

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega \cup \{O\restriction_\omega \geq O\} \tag{1}$$

that all consequences on the right (with the substitution ω applied to the constraints) follow from previously derived constraints $\mathcal{C} \cup \mathcal{D}$ together with the negation $\neg C$ of the constraint to be inferred. This guarantees that if some assignment α satisfies everything so far but violates C, the "patched" assignment $\alpha \circ \omega$ satisfies also C and does not worsen the objective.

To make our informal discussion simple and concrete, suppose that we have a CNF encoding of a circuit $\mathsf{BC}(O-T \geq lo)$ evaluating $O-T \geq lo$, and that the solver has derived no constraints but only has the input formula F. If we want to fix $T = T^*$ using the redundance rule (1), we would have to find a substitution ω such that $F \cup \{\mathsf{BC}(O-T \geq lo)\} \cup \{T \neq T^*\}$ implies $\big(F \cup \{\mathsf{BC}(O-T \geq lo)\} \cup \{T = T^*\}\big)\restriction_\omega$. But it seems like this would force us to prove that if we take any assignment satisfying the Boolean circuit and modify the value of some of its inputs (the tares), the circuit would remain satisfied, and this is just not true. So although the redundance-based strengthening rule is very strong, it is not clear how it can be used to argue that the tare variables are unconstrained.

We get around this problem by first deriving a copy *shadow circuit* BC' of the original circuit, but substituting fixed values t_i^* for the tare variables, so that $\mathsf{BC}'(O-T^* \geq lo)$ evaluates $O-T^* \geq lo$. We then let ω be the substitution setting $t_i = t_i^*$ for all i and mapping all other variables x in BC to the corresponding shadow variables x' in BC', so that, effectively, the shadow circuit computes the substitution needed. This turns our application of the redundance rule (1) into

$$F \cup \{\mathsf{BC}(O-T \geq lo)\} \cup \{\mathsf{BC}'(O-T^* \geq lo)\} \cup \{T \neq T^*\} \tag{2a}$$

$$\vdash \big(F \cup \{\mathsf{BC}(O-T \geq lo)\} \cup \{\mathsf{BC}'(O-T^* \geq lo)\} \cup \{T = T^*\}\big)\restriction_\omega \cup \{O\restriction_\omega \geq O\} \tag{2b}$$

$$= F \cup \{\mathsf{BC}'(O-T^* \geq lo)\} \cup \{\mathsf{BC}'(O-T^* \geq lo)\} \cup \{T^* = T^*\} \cup \{O \geq O\} \tag{2c}$$

(where the final line (2c) is simply the result of applying the substitution ω to (2b)). If we study (2c) carefully, we see that all we need to prove about the circuit now is that the two copies of the shadow circuit in the consequences are implied by the same shadow circuit in the premises, and so (2c) follows trivially from the premises (2a).

This idea of using shadow circuits is crucial for certifying the correctness of assigning tare variables without loss of generality. However, we need to get rid of the completely unrealistic assumption that the solver would not have learned any constraints in $\mathcal{D}$. This is a problem in that the above argument fails when such learned constraints $D \in \mathcal{D}$ contain variables in the BC-circuit, since then there is no way to prove $D\restriction_\omega$ as required in (1).

Here a second idea discovered in recent VeriPB development turns out to be very helpful. Very briefly, it can be shown that if in the proof we enforce the requirement that all new constraints D derived by strengthening are immediately moved to the core set $\mathcal{C}$, referred to as *strengthening-to-core*, then the redundance rule (1) can be simplified to

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \{C\})\restriction_\omega \cup \{O\restriction_\omega \geq O\}, \tag{3}$$

omitting the proof obligations for the derived set $\mathcal{D}$. This means that we can ignore the problems arising from derived constraints when using shadow circuit reasoning.

We stress that this is only a brief and informal discussion that sweeps many technical challenges under the rug. Perhaps one of the most annoying such challenges is that the tare variables are sometimes fixed one at a time, and then a new shadow circuit is required for every new fixing. It would be desirable to find better ways of dealing with this problem.

We have implemented our methods in the state-of-the-art solution-improving MaxSAT solver Pacose [52] to make it output VeriPB proofs, and have performed an extensive evaluation of how such proof logging works in practice. While there is certainly room for performance improvements in both proof generation and proof checking, the significance of our contribution is that we present practical methods to certify correctness for a solving paradigm that has previously been beyond the reach of proof logging. We hope that our work can serve as an impetus towards general adoption of proof logging for MaxSAT, and can stimulate further research on how to make these proof logging techniques more efficient.

As a final remark, we note that an interesting aspect of recent progress in proof logging is that it brings together all three software quality assurance methods discussed in the opening paragraphs above. While proof logging does seem like the most viable approach to certify correctness in combinatorial solving, extensive use of fuzzing techniques has been instrumental in our work to debug both proof logging routines and the VeriPB proof checker. This fuzzing, in turn, relies on the use of proof logging and on feedback from the proof checker. Finally, although we do not address this aspect in the current paper, formally verified proof checking backends as in [33,37] are crucially needed to ensure that the verdict of proof checkers for increasingly powerful proof logging systems can be trusted.

1.4 Outline of This Paper

After reviewing some preliminaries in Section 2, we discuss the dynamic polynomial watchdog (DPW) encoding in Section 3. In Section 4 we describe how to design proof logging for solution-improving solvers using the DPW encoding, including a discussion of possible variations of our method (and of why simply using SAT proof logging for the final unsatisfiability call does not work). We report results from an empirical evaluation in Section 5 and end with some conclusions and a discussion of future research directions in Section 6.

2 Preliminaries

In this section, we review some pseudo-Boolean basics and then discuss MaxSAT in general and solution-improving search in particular, referring the reader to [3,17,44] for more details.

2.1 Pseudo-Boolean Constraints and Proofs

We write x to denote a $\{0,1\}$-valued Boolean variable, and write $\overline{x}$ as a shorthand for $1-x$, using ℓ to denote such *positive* and *negative literals*, respectively. A (linear) *pseudo-Boolean (PB) constraint* C is a 0–1 integer linear inequality $\sum_i w_i \ell_i \geq A$. Without loss of generality, we will often assume our constraints to be *normalized*, meaning that all literal are over distinct variables and the coefficients w_i and the *degree* A are non-negative. A *PB formula* is a conjunction of PB constraints.

A (disjunctive) *clause* is a PB constraint $\sum_i \ell_i \geq 1$ with all coefficients and degree equal to 1. We sometimes refer to constraints $\ell \geq 1$ with a single literal as *unit clauses* ℓ. We say that a formula is in *conjunctive normal form (CNF)* if it is a conjunction of clauses. A *(linear) pseudo-Boolean term* is a weighted sum $\sum_i w_i \ell_i$ of literals with integer coefficients. A *(partial) assignment* α is a (partial) function from variables to $\{0,1\}$; it is extended to literals by respecting the meaning of negation. We write $C{\restriction}_\alpha$ for the constraint obtained from C by substituting all assigned variables x by $\alpha(x)$ (and simplifying). A constraint C is *satisfied* under α if $\sum_{\alpha(\ell_i)=1} w_i \geq A$, and a formula F is satisfied if all its constraints are. We say that F *implies* C, denoted $F \models C$, if all assignments that satisfy F also satisfy C.

A *pseudo-Boolean optimization* (PBO) instance consists of a formula F and a linear term $O = \sum_i w_i \ell_i$ (called the *objective*). An assignment α to the variables in F and O that satisfies F is a *solution* to the instance, which is optimal if it *maximizes* the value $O{\restriction}_\alpha = \sum_i w_i \alpha(\ell_i)$.[1] For a PBO instance (F, O) the VeriPB proof system maintains a

[1] Note that most of the PBO literature is formulated in terms of *minimization*, and this is also the perspective of VeriPB, but reasoning in terms of maximization is in line with the papers on solution-improving MaxSAT relevant for this work. We therefore adopt this perspective here, although the actual VeriPB proofs will argue in terms of minimizing the negation of the objective as described here.

proof configuration of *core* and *derived constraints* $(\mathcal{C}, \mathcal{D})$, initialized to F and $\emptyset$, respectively. The VeriPB proofs we consider are in the so-called *strengthening-to-core* mode, which maintains the invariant that all constraints in the derived set $\mathcal{D}$ are implied by the core set $\mathcal{C}$. Constraints can be moved from $\mathcal{D}$ to $\mathcal{C}$ but not vice versa. New constraints can be derived from $\mathcal{C} \cup \mathcal{D}$ and added to $\mathcal{D}$ using the *cutting planes* proof system [18] as follows:

Literal Axioms. For any literal ℓ_i, $\ell_i \geq 0$ is an axiom.

Linear Combination. Given two previously derived PB constraints C_1 and C_2, any positive integer linear combination of these constraints can be inferred.

Division. Given the normalized PB constraint $\sum_i w_i \ell_i \geq A$ and a positive integer c, the constraint $\sum_i \lceil w_i/c \rceil \ell_i \geq \lceil A/c \rceil$ can be inferred.

Some additional VeriPB proof rules extending cutting planes are as listed below – we refer to [12, 34, 36] for more details. For optimization problems we have rules for improvements of or rewriting of the objective function:

Objective Improvement. Given a total assignment α that satisfies $\mathcal{C} \cup \mathcal{D}$, one can add the constraint $O \geq 1 + O\restriction_\alpha$ to $\mathcal{C}$, which forces the search for strictly better solutions.

Objective Reformulation. The current objective O can be replaced by a new objective O_{new} given explicit proofs from the core set $\mathcal{C}$ (using the VeriPB proof rules above) of the constraints $O - O_{\text{new}} \geq 0$ and $O_{\text{new}} - O \geq 0$ (i.e., a proof that $O = O_{\text{new}}$ holds).

Importantly, there are also rules for deriving non-implied constraints as long as the optimal value of the objective is preserved. VeriPB has a generalization of the RAT rule [39] that makes use of *substitutions* ω, mapping variables to truth values or literals (where we extend the meaning of $C\restriction_\omega$ to denote C with each x replaced by $\omega(x)$):

Redundance-Based Strengthening. The constraint C can be inferred and added to $\mathcal{C}$ by explicitly specifying a substitution ω and proofs $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \{C\})\restriction_\omega \cup \{O\restriction_\omega \geq O\}$. This assumes strengthening-to-core mode – otherwise derivations for all constraints in $\mathcal{D}\restriction_\omega$ are also needed (but then C can be placed in $\mathcal{D}$ instead of $\mathcal{C}$).

Intuitively, this rule shows that ω remaps any solution of $\mathcal{C}$ that does not satisfy C to a solution of $\mathcal{C}$ that satisfies also C without worsening the objective value. A typical use case of redundance-based strengthening is *reification*, which is the derivation of two pseudo-Boolean constraints that encode $\ell \Leftrightarrow D$ for some PB constraint D and for some fresh literal ℓ.

Finally, VeriPB has rules for deleting constraint in a way that guarantees that no spurious better-than-optimal solutions are introduced:

Deletion. A constraint $D \in \mathcal{D}$ in the derived set can be deleted at any time. If strengthening-to-core mode is used, then deleting a constraint $C \in \mathcal{C}$ in the core set requires an explicit proof that C is implied by $\mathcal{C} \setminus \{C\}$. Otherwise, it is sufficient to show the weaker property that C can be derived from $\mathcal{C} \setminus \{C\}$ by redundance-based strengthening.

2.2 MaxSAT, Incremental SAT Solving, and Solution-Improving Search

An instance of (weighted partial) Maximum Satisfiability (MaxSAT) consists of a CNF formula F and a pseudo-Boolean objective $O = \sum_i w_i \ell_i$ to be maximized under satisfying assignments to F, where we can assume without loss of generality that all literals in O are over distinct variables and that the constants are positive. Viewing MaxSAT in terms of an objective function and a CNF formula is equivalent to the more classical definition in terms of hard and soft clauses, in the sense that maximizing the objective corresponds to maximizing the total weight of satisfied soft clauses (see, e.g., [43] for more details).

The solution-improving search (SIS) algorithm we focus on in this work makes extensive use of incremental SAT solving with assumptions [22]. Invoking a SAT solver on a CNF formula F with a set of *assumptions* $\mathcal{A}$, i.e., a partial assignment, returns either (i) SAT and an extension of $\mathcal{A}$ that satisfies F or (ii) UNSAT if no such assignment exists.

Given a MaxSAT instance (F, O), solution-improving search (SIS) computes an optimal solution by issuing a sequence of queries to a SAT solver asking for solutions of improving quality until an optimal one is found. More precisely, during search SIS maintains the best known solution α^*. In each iteration, the algorithm queries a SAT solver on the working formula $F \wedge \mathsf{AsCNF}(O > O\restriction_{\alpha^*})$, where $\mathsf{AsCNF}(O > O\restriction_{\alpha^*})$ is a CNF formula that is satisfied by an assignment α if and only if it is a better solution than α^*, i.e., if $O\restriction_{\alpha} > O\restriction_{\alpha^*}$. If the SAT solver returns SAT, a better solution has been obtained and the working formula updated accordingly. Otherwise, if the SAT solver reports UNSAT, the best known solution α^* is determined to be optimal and the search is terminated.

The existing practical instantiations of SIS differ mainly in how the encoding of the formula $\mathsf{AsCNF}(O > O\restriction_{\alpha^*})$ is realized. Numerous CNF encodings of pseudo-Boolean constraints have been proposed for this task [23, 38, 40, 45, 58]. For many instantiations of SIS the main challenge for proof logging is to certify the clauses added when encoding the objective constraint [59, 60], but as we will explain in the rest of this paper the so-called Dynamic Polynomial Watchdog encoding requires much more subtle arguments.

3 The Dynamic Polynomial Watchdog Encoding for SIS

The polynomial watchdog (PW) encoding [6] is currently one of the best approaches for encoding pseudo-Boolean constraints in CNF, in terms of being compact while still propagating well. Using it for solution-improving search requires some non-trivial alternations, however, such as the addition of a dynamic constant. In this section we review this dynamic polynomial watchdog (DPW) encoding to the extent required for MaxSAT solution-improving search (SIS), referring the reader to [52] for more details.

3.1 Initialization

Given a linear pseudo-Boolean term $L = \sum_i w_i \ell_i$, we define $w_{\max}$ to be the largest constant appearing in L. Additionally, we let $P := \lfloor \log_2(w_{\max}) \rfloor$ be one smaller than the number of bits in the binary representation of $w_{\max}$ and $W := \sum_i w_i$ be the maximum value for L. The polynomial watchdog encoding for L is a CNF formula $\mathsf{PW}(L)$ with $c := \lceil \frac{W}{2^P} \rceil$ output variables z_k for $k \in [0, c-1]$ enforcing the implications $\overline{z}_k \Rightarrow L \geq 1 + k \cdot 2^P$. In words, a satisfying assignment α of $\mathsf{PW}(L)$ that sets $\alpha(z_k) = 0$ will also satisfy $\sum_i w_i \alpha(\ell_i) \geq 1 + k \cdot 2^P$. We describe the formula $\mathsf{PW}(L)$ in more detail in Section 4.1.

▶ **Example 1.** Consider a MaxSAT instance (F, O) and a working formula $F^w = F \wedge \mathsf{PW}(O)$. Assume we first invoke a SAT solver on F^w under the assumption $z_{k-1} = 0$ and then a second time under the assumption $z_k = 0$, and that the solver reports SAT for the first call and UNSAT for the second. At this point, we know that an optimal solution α^{opt} has value $O\restriction_{\alpha^{\text{opt}}}$ in the range $\left[1 + (k-1) \cdot 2^P, k \cdot 2^P\right]$.

The PW encoding was proposed as a way of enforcing a fixed bound B on the term L by considering a (static) constant $T = B - (1 + k \cdot 2^P)$, where k is the largest integer for which $B \geq 1 + k \cdot 2^P$, and encoding $\mathsf{PW}(L - T)$ [6]. Then a solution that sets the k^{th} output z_k of $\mathsf{PW}(L - T)$ to 0 will also satisfy $\sum_i w_i \alpha(\ell_i) - T \geq 1 + k \cdot 2^P$, which is equivalent to

$\sum_i w_i\alpha(\ell_i) \geq B$. The dynamic polynomial watchdog (DPW) encoding [52] is an extension of the PW encoding that allows dynamically changing the value of T, and therefore also of B, so that the optimal value can be determined precisely with a single CNF encoding.

Consider a MaxSAT instance (F, O) and let $P = \lfloor \log_2(w_{\max}) \rfloor$ as described above. Instantiations of SIS with DPW introduce a "dynamic constant" in the form of a *tare* term $T := \sum_{i=0}^{P-1} 2^i \cdot t_i$, for fresh variables t_i not appearing anywhere else in the instance. The SAT solver is instantiated with the working formula $F \wedge \mathsf{PW}(O - T)$. Now we can use the output variables z_k to determine the optimal value within an additive constant 2^P, and then assign the tare T to values in $\left[0, 2^P - 1\right]$ to determine the precise value in that range. These are the *coarse convergence* and *fine convergence* phases mentioned in Section 1.2, which we describe in more detail next.

3.2 Coarse Convergence Phase

During the initial coarse convergence phase, only assumptions over the output variables z_k are made. Whenever a solution α is found, a call to the SAT solver is made with the assumption $z_k = 0$ where k is the largest natural number such that $O\restriction_\alpha \geq 1 + (k-1) \cdot 2^P$. The coarse convergence phase ends when the solver reports UNSAT. The following observation summarizes the relevant conclusions of coarse convergence.

▶ **Observation 2.** *Assume F is satisfiable and the SAT solver returns UNSAT under an assumption $z_{k^*} = 0$ in the coarse convergence phase. Then (i) there is a solution α^* to $F \wedge \mathsf{PW}(O-T)$ that assigns the tare variables so that $(O-T)\restriction_{\alpha^*} \geq 1 + (k^* - 1) \cdot 2^P$ holds, and (ii) no solution β to F assigning also the tare variables can satisfy $(O-T)\restriction_\beta \geq 1 + k^* \cdot 2^P$.*

In words, coarse convergence provides bounds on the maximum value of $O - T$ obtainable by any solution of F. Importantly, as the tare term T is unconstrained by the formula F, its value can without loss of generality be assumed to be 0 at this stage, resulting in bounds on the objective value of optimal solutions as well. From now on, the algorithm commits to only searching for solutions that have $O - T$ in the specified interval, adding the unit clauses $\overline{z}_{k^*-1}$ and z_{k^*} to the working formula before proceeding to the fine convergence phase. In practice, whenever the SAT solver returns SAT after being called with assumption $\overline{z}_k$, the unit clause $\overline{z}_k$ is added immediately, allowing the SAT solver to simplify its clause database.

3.3 Fine Convergence Phase

During the fine convergence phase, assumptions for the tare variables are used to pinpoint the precise optimal value. Let k^* be the value for which the assumption $z_{k^*} = 0$ returned UNSAT in coarse convergence, and $o^* = O\restriction_{\alpha^*}$ the objective value of the currently best known solution α^*. Then we define $s := o^* - (k^* - 1) \cdot 2^P$ to be the smallest value of the tare that would force an improved solution. The next call to the SAT solver assumes $t_i = 1$ for all tare variables for which the i^{th} bit in the binary representation of s is 1. These assumptions enforce $T \geq s$, so any solution α to the working formula (which now includes the unit clause $\overline{z}_{k^*-1} \geq 1$) that extends the assumptions will satisfy $O\restriction_\alpha \geq o^* + 1$.

The fine convergence phase continues in this manner until the SAT solver reports UNSAT, at which point an optimal solution has been found. As the value of s is monotonically increasing, we add unit clauses t_i to the working formula whenever we have deduced that the i^{th} bit t_i in the tare T can safely be set to 1 in any solution (and hence in any future SAT call), which is the case when $s - 1 \geq 1 + \sum_{j=i}^{P-1} 2^i \cdot t_j$ holds. The fact that we have $s - 1$ rather than s in this last inequality is related to *stratification*, which we discuss next.

3.4 Stratification

Stratification is a technique for partitioning the indices of an objective $O = \sum_{i=1}^{m} w_i \ell_i$ into two sets $\{H, L\}$ in a way that allows computing the maximum values first of $O_H = \sum_{i \in H} w_i \ell_i$ and then of $O_L = \sum_{i \in L} w_i \ell_i$, and finally combining them to get the maximum value of O.

Specifically, stratification is applied when $\gcd\{w_i \mid i \in H\} \geq \sum_{i \in L} w_i$, i.e., when the greatest common divisor of the coefficients in O_H is at least the sum of all coefficients in O_L. SIS with the DPW encoding and stratification will first run coarse and fine convergence only on O_H as described above. At the end of the fine convergence, the SAT solver returns UNSAT after being invoked with assumptions that enforce $T_H \geq s$ for the tare term T_H added to the DPW encoding of O_H and some constant s. At this stage, the value of T_H will be fixed to $s - 1$ with unit clauses, effectively fixing O_H to its maximum value. This fixing of O_H is consistent with the unit clauses learned in the previous section. After this O_L is optimized via coarse and fine convergence under the fixed value of O_H. The solution obtained at the end of the final fine convergence phase will be optimal with respect to the original instance. For more details on stratification, we refer the reader to [2, 51].

▶ **Example 3.** Consider the objective $O = 10x_1 + 5x_2 + 5x_3 + 3x_4 + 2x_5$ and the partition $H = \{1, 2, 3\}$ and $L = \{4, 5\}$. Since $\gcd\{10, 5, 5\} = 5 \geq 3 + 2$, changes of the objective restricted to $\{x_1, x_2, x_3\}$ will dominate any contributions from $3x_4 + 2x_5$. If a solution α with $O_H{\upharpoonright_\alpha} = 15$ is found, we can without loss of generality assume $O_H \geq 15$, since for any solution β with $O_H{\upharpoonright_\beta} < 15$ we have $O{\upharpoonright_\beta} \leq O{\upharpoonright_\alpha}$. Notice that maximizing first O_H and then O_L can remove some optimal solutions from the search space, but never all of them.

4 Certifying Solution-Improving MaxSAT with the DPW Encoding

We are now ready to describe how to do proof logging for solution-improving MaxSAT with the dynamic polynomial watchdog encoding. In addition to certifying the correctness of CNF encodings, as done in previous work on proof logging SIS for MaxSAT [59, 60], we need to certify the without loss of generality reasoning discussed in Section 3. This turns out to require quite intricate proof logging methods.

We start with a brief discussion how to certify the DPW encoding. We then turn to proof logging for the without loss of generality reasoning during the coarse and fine convergence phases. Afterwards, we deal with proof logging for stratification. We defer a discussion of minor additional heuristics used in state-of-the-art solvers to Appendix B. We note that for all clauses learned by the SAT solver we can use standard VeriPB proof logging, and since all such learned clauses are logically implied by the working formula it is safe to add them to the derived set $\mathcal{D}$. This means that we can ignore all constraints added to the database by the SAT solver when we perform redundance-based strengthening steps.

4.1 Proof Logging for Clauses of the DPW Encoding

Figure 1 depicts the structure of the DPW encoding of the term $2x_1 + 3x_2 + 5x_3 + 7x_4$. For a term L in which the largest coefficient has P bits, the encoding introduces P totalizers [5] (which are circuits that sort their inputs), and $P - 1$ mergers. The i^{th} totalizer takes as input all variables in L for which the corresponding coefficient has its i^{th} bit equal to 1.

Proof logging for the DPW encoding boils down to taking care of the totalizer encodings as described in [60]. At a high level, the proof for $\mathsf{PW}(O - T)$ derives a number of constraints encoding implications $y \Rightarrow C_y$ and $y \Leftarrow C_y$, where y are variables in the auxiliary variable set Y and C_y are suitably chosen PB constraints over the variables in $O - T$. A concrete

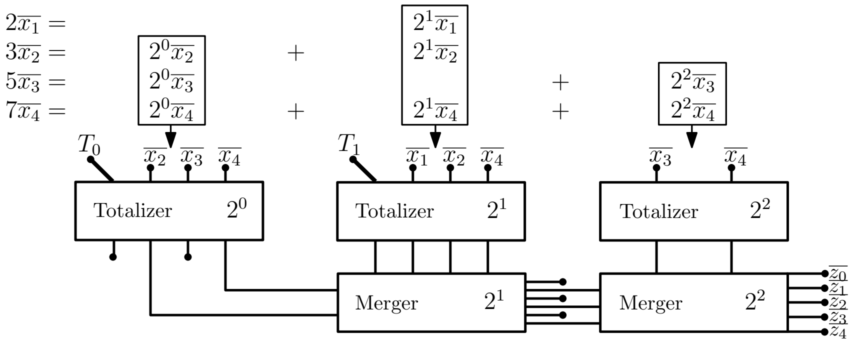


Figure 1 Illustration of the polynomial watchdog encoding.

example is the output variable z_k for which the constraint C_{z_k} is chosen as $O - T \leq k \cdot 2^P$. From these pseudo-Boolean definitions all clauses in the CNF encoding added to the solver database can be derived with explicit VeriPB derivations. A technical point that is crucial for the proof logging is that in this way we only need to add the PB definitions of new variables to the core set $\mathcal{C}$. The clauses actually used for the SAT solver calls are implied from these definitions, and can therefore be placed in the derived set $\mathcal{D}$.

4.2 Proofs Without Loss of Generality Using Shadow Circuits

The MaxSAT solving algorithm uses without loss of generality (wlog) reasoning when (i) introducing fresh variables for encoding $\mathsf{PW}(O - T)$; (ii) adding unit clauses $\overline{z}_k$ during coarse convergence; (iii) learning unit clause over the tare variables t_i during fine convergence; and (iv) concluding that the optimal value has been found.

To see why unit clauses $\overline{z}_k \geq 1$ require wlog reasoning, suppose in the coarse convergence phase that the SAT solver returns a solution α when invoked with the assumption $z_k = 0$, indicating that $(O - T)\restriction_\alpha \geq 1 + k \cdot 2^P$. The constraint $\overline{z}_k \geq 1$ is *not* entailed by the solution-improving constraint $O \geq O\restriction_\alpha$, since some other (possibly optimal) solution β might have $O\restriction_\beta \geq O\restriction_\alpha$ but assign the tare variables so that $(O - T)\restriction_\beta < 1 + k \cdot 2^P \leq (O - T)\restriction_\alpha$ holds. However, since the tare variables are not constrained by the original formula F, any solution to F could be extended to any fixed value for the tare T. Hence, in particular, we can assume without loss of generality that $T = 0$, which in turn implies that $\overline{z}_k \geq 1$.

The fine convergence phase makes use of the fact that the DPW encoding does not constrain T, which takes values in the range $\left[0, 2^P - 1\right]$. The unit clauses $t_i \geq 1$ learned are not entailed, but can be deduced since the tare variables are unconstrained in the DPW encoding. This requires a VeriPB proof that wlog $T \geq s - 1$. When the SAT solver reports UNSAT during fine convergence, it does so under the assumption that a specific set of tare variables take value 1. If this yields UNSAT, then we can conclude that the current solution is optimal (since we can wlog assume T to be equal to the value that led to UNSAT).

It is worth noticing that the without loss of generality arguments above are quite intricate even at a human meta-level. The coarse convergence phase repeatedly claims to be able to assume $T = 0$, after which the fine convergence phase picks an increasing sequence $0 < s_1 < s_2 < \ldots$ and assumes $T \geq s_i - 1$ wlog. Finally, a specific value $T = s_{i^*}$ is used to argue about optimality. The meta-level argument for why this works is that no conclusions are drawn from the assumptions made during coarse and fine convergence that invalidate subsequent assumptions. The challenge is how to convince a mechanical proof checker of this.

Consider first proof logging for the coarse convergence phase, and suppose the solver returns SAT when invoked with assumption $\overline{z}_k$. The only rule that would allow us to derive $\overline{z}_k \geq 1$ without loss of generality (from the argument that we can set $T = 0$ wlog) is *redundance-based strengthening*, which requires specification of a witness substitution ω that can be used to "patch" any assignment α in which $\overline{z}_k \geq 1$ is violated. More formally, our witness should guarantee that $\mathcal{C} \cup \mathcal{D} \cup \{\neg(\overline{z}_k \geq 1)\} \models \big(\mathcal{C} \cup \{\overline{z}_k \geq 1\}\big)\restriction_\omega \cup \{O \leq O\restriction_\omega\}$. A natural approach would be to choose a witness ω that maps (i) z_k to 0, (ii) all original variables to themselves, and (iii) T to 0. Such a witness would make $(\overline{z}_k \geq 1)\restriction_\omega$ trivially true and would incur no proof obligations for the formula F or the objective O. However, setting $T = 0$ will not work for the constraints $C \in \mathcal{C}$ defining variables in the DPW encoding. If we fix $T = 0$, then we also need to update all auxiliary variables Y in the circuit evaluating $\mathsf{PW}(O - T)$. But how this should be done depends on which assignment α we need to patch, and the redundance rule has no mechanism for defining "conditional witnesses" $\omega = \omega(\alpha)$.

To determine how the witness should assign the auxiliary variables in $\mathsf{PW}(O - T)$, we devise a new proof logging technique that we call *shadow circuits*. Corresponding to each auxiliary variable y defined as the reification of a PB constraint C_y in the original circuit, a *shadow circuit for a fixed value* v has a fresh variable $y^{T=v}$ defined by $y^{T=v} \Leftrightarrow C_y\restriction_{T\mapsto v}$. In words, the defining constraints of $y^{T=v}$ and y are the same except that we fix the tare variables t_i so that $T = v$. The definitions of such shadow circuits are stored in the core set $\mathcal{C}$ since they are derived using the redundance rule. Note that the shadow circuit only "copies" the pseudo-Boolean definitions of the variables and not their clausal encodings.

Shadow circuits provide us with a mechanism to compute witnesses for the redundance rule that allow us to assume the value of T and certify the without loss of generality reasoning. During coarse convergence, each addition of a constraint $\overline{z}_i \geq 1$ is logged with a witness that maps all tare variables t_i to 0 and other auxiliary variables y in $\mathsf{PW}(O - T)$ to their counterparts $y^{T=0}$ in the shadow circuit for $T = 0$. During fine convergence, the constraints $T \geq s - 1$ are derived using shadow circuits for $s - 1$, which allows adding unit constraints over individual tare variables to the proof. Finally, for proving optimality a shadow circuit for the final value s^* for which the SAT solver returned UNSAT will be used to derive contradiction.

The next proposition gives a more formal summary of the wlog proof logging performed during the coarse convergence phase. The proof for this proposition, together with precise descriptions of the other wlog proof logging steps, are given in Appendix A.

▶ **Proposition 4.** *Suppose the* VeriPB *proof log contains derivations of reification constraints* $\overline{z}_k \Leftrightarrow O - T \geq 1 + k \cdot 2^P$ *and a shadow circuit for* $T = 0$ *as well as the constraint* $O \geq 1 + k \cdot 2^P$. *Then the constraint* $\overline{z}_k \geq 1$ *can be derived using redundance-based strengthening with witness* $\omega = \{t_i \mapsto 0 \mid 0 \leq i \leq P - 1\} \cup \big\{y \mapsto y^{T=0} \mid y \in Y\big\}$.

The constraint $O \geq 1 + k \cdot 2^P$ in Proposition 4 can be obtained by weakening the solution-improving constraint $O \geq O\restriction_\alpha + 1$ for the previously found solution α. If stratification is used, deriving $O_H \geq 1 + k \cdot 2^P$ requires more work (see Section 3.4 for details).

Our technique with shadow circuits and repeated without loss of generality arguments selecting (different) values for the same variables in T heavily relies on that VeriPB proofs in the *strengthening-to-core* mode maintain the guarantee that all constraints in the derived set $\mathcal{D}$ are entailed by the core set $\mathcal{C}$. In particular, what this means is that whenever we want to apply redundance-based strengthening, fixing tare variables and using the corresponding shadow circuit, we do not need to worry about reproving any clauses learned by the SAT solver under the witness ω. It turns out that for all non-trivial proof obligations, the solution-improving constraint $O \geq O\restriction_\alpha$ for the latest solution α obtained is helpful. This also makes it easier to see why the entire pipeline is consistent. During coarse convergence, we never derive

$T = 0$, but instead derive $z_k = 0$ for certain values of k using the fact that we could set $T = 0$ wlog. This constraint $z_k = 0$ will be used by the solver for deriving several consequences. Later, when we make the wlog argument that $T \geq s - 1$ for some value s, this incurs the obligation to reprove that $z_k = 0$ holds! That is, the proof checker realizes that $z_k = 0$ was also derived wlog, and we need to prove that this is still consistent with the current wlog assumption to justify that we can "change our mind" about the value of T.

The use of *strengthening-to-core* requires some extra care when dealing with constraint deletions. SAT solvers use heuristics to aggressively erase clauses that are believed to no longer be useful, and this is crucial for performance. Also, clauses in the input are removed whenever some literal in the clause is deduced to be true. In strengthening-to-core mode, we can still do unrestricted deletions of constraints in the derived set $\mathcal{D}$, but a core constraint $C \in \mathcal{C}$ can only be erased if the implication $\mathcal{C} \setminus \{C\} \models C$ can be shown to hold. For this reason we did not implement deletion from the core set in our proof logging routines.

4.3 Stratification

For proof logging of stratification steps as in Section 3.4, we need to be able to convert known facts about the whole objective O to statements about the split objectives O_H and O_L. To certify a unit constraint added during coarse convergence or to derive the constraints $T \geq s - 1$ during fine convergence when maximizing O_H, we need to derive $O_H \geq O_H{\upharpoonright_\alpha}$ from $O \geq O{\upharpoonright_\alpha} + 1$. We do this by weakening away all terms in O_L – meaning that for every term $w_i \ell_i$ in O_L we add $w_i \overline{\ell}_i \geq 0$ to cancel the term – to get $O_H \geq O{\upharpoonright_\alpha} + 1 - g$, where g is the greatest common divisor of the coefficients in O_H. This clearly also entails $O_H \geq O_H{\upharpoonright_\alpha} - g + 1$. Dividing by g and rounding up yields $\frac{1}{g} O_H \geq \frac{O_H{\upharpoonright_\alpha}}{g} - 1 + 1$, and multiplying this again by g yields $O_H \geq O_H{\upharpoonright_\alpha}$.

By applying this reasoning, we can derive the constraint $O_H \geq o_H^*$ right after finding the optimal value o_H^* for O_H. Moreover, after introducing a shadow circuit for $T = s$, we can derive (local) optimality in the form of the constraint $O_H \leq o_H^*$. Hence, we can reformulate the objective by replacing O_H with the constant o_H^*, from which we can now derive the constraint $O_L + o_H^* \geq O{\upharpoonright_\alpha} + 1$. Observe that this constraint coincides with the solution-improving constraint for O_L. Once the constraints $O_L \geq o_L^*$ and $O_L \leq o_L^*$ have been derived in a similar way, the objective will be rewritten to a constant, for which proving optimality boils down to logging a solution that has objective value $o^* = o_H^* + o_L^*$.

4.4 Limiting the Use of Shadow Circuits

Our proof logging method makes repeated use of shadow circuits, which are copies of the original circuit, and repeatedly deriving all constraints defining such circuits could potentially incur serious overhead for proof generation in the solver. Let us discuss ways of limiting or completely eliminating the use of shadow circuits and the downside of such approaches.

First, the shadow circuits are introduced each time the solver deduces a unit clause over an output variable z_k or tare variable t_i. Instead of learning these unit clauses, we could do all subsequent solver calls with those literals as assumptions. At the very end of the fine convergence phase, we could then introduce a single shadow circuit to prove optimality (or, in case of stratification, two shadow circuits: one to prove optimality and one to fix the value of the tare variables). The disadvantage is that when variables used as assumptions, the solver cannot use them to simplify its clause database; so while this would have a positive effect on the time required to do the actual proof logging, it could have negative effects on solving time. Appendix C.2 reports on an experimental evaluation of this approach.

Second, there is a way to completely eliminate shadow circuits. By the end of the execution, the solver knows which value $T = s$ resulted in the final $UNSAT$ call in the fine convergence. What we could do at this point is insert at the *beginning* of the proof constraints saying that $T = s$ holds (which at this point can easily be derived by redundance-based strengthening). The rest of the proof will then be checked for a fixed value of T that happens to be the value needed at the end. There are two important reasons why we prefer the shadow circuit approach. The first reason is that it is not clear if and how this would work together with stratification, where after a stratification level we want to fix $T = s - 1$. The second reason is that fixing T in advance adds substantial new information that the solver did not have available when constructing the proof. This means that we would not be verifying that the reasoning the solver actually performed was correct, but only that its reasoning checks out given advance information about the optimal solution. While this could still be used to certify the correctness of the final answer, it would not provide any guarantees about the process leading there. It has been shown repeatedly that proof logging can catch subtle bugs in solvers that only report correct results but for the wrong reasons [9,24,32,41], but in order for this to be possible the correctness of solver-generated proofs should only depend on what the solver actually knows when the proof is being produced.

4.5 Discussion of an Even Simpler Approach and Why It Does Not Work

The proof logging techniques in this paper certify every single reasoning step in the solver. An alternative, and seemingly much simpler, way to get proofs of correctness for *any* MaxSAT solver would be to (i) compute an optimal solution by running the MaxSAT solver without proof logging, (ii) check that this solution is feasible, (iii) encode a solution-improving constraint into CNF, and (iv) call a SAT solver to generate a proof of unsatisfiability (and hence of optimality of the solution) with standard SAT proof logging. However, there are several serious issues with this approach that we would like to point out.

First, proofs of correctness are needed for the CNF encodings used in step (iii), and such proofs cannot be done with SAT proof logging since it cannot reason about values of objective functions. Second, it is not possible to just repeat the “final UNSAT call” of the MaxSAT solver in step (iv). Even if the same SAT solver is used, in the original UNSAT call this solver had access to all constraints learned in previous calls, and there is no guarantee that the solver will learn these constraints again, or other equally good constraints, when it is now run in a different way and with a different input. It is therefore impossible to know for sure whether the final SAT solver invocation with the solution-improving constraint would be faster or, more likely, slower, than the original solving process, and by how much. This defeats the whole idea of generating proofs with a small and predictable overhead, since there would be no way of knowing in advance whether “proof logging” for a previously claimed result would succeed or not. Moreover, when a solution-improving MaxSAT solver makes use of stratification (as discussed in Section 3.4), then optimality is not derived by a single UNSAT call but by a combination of UNSAT calls at different levels. It is hard to see how such a combination of calls could be replicated with the simple approach described above.

Third, an increasingly popular usage scenario for MaxSAT solvers is so-called anytime solving, where the solver can be terminated at any point and then returns the best upper and lower bounds on the objective computed so far. Proofs constructed as described in this paper (as well as in other MaxSAT papers using VeriPB proof logging) will at all times contain formal proofs of everything the solver knows about upper and lower bounds on the objective. Whenever the solver is terminated, it can therefore just end the generated proof at that point by printing a concluding line stating what upper and lower bounds have been proven. This functionality would be lost in the alternative approach.

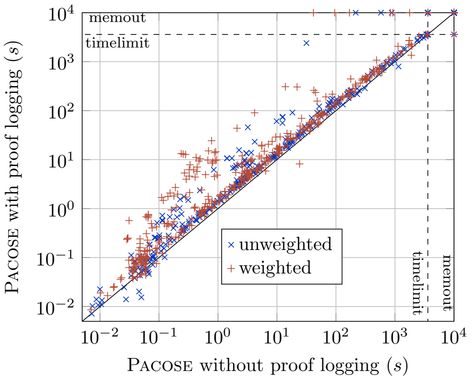


Figure 2 Proof logging overhead for PACOSE using the DPW encoding.

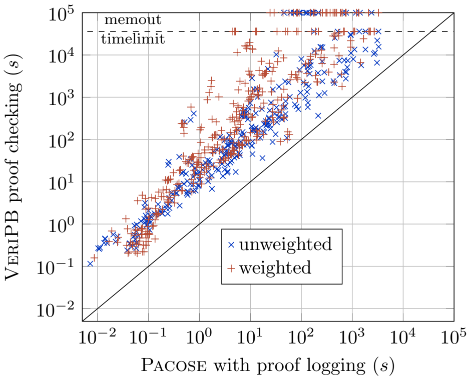


Figure 3 PACOSE vs. VERIPB running time using DPW encoding.

Finally, even if this approach could be made to work efficiently – which, as explained above, is not really the case, for several reasons – we would have the same problem as in Section 4.4 that we would only certify the final result and not the solver reasoning process.

5 Experimental Evaluation

To evaluate our proof logging approach in practice, we implemented it in the state-of-the-art solution-improving MaxSAT solver PACOSE [52]. The source code for all software tools used, as well as all experimental data, are available in [8]. During development, we extensively checked the correctness of our implementation with a fuzzer [50] and minimized failed instances with a delta debugger. This process accelerated the development, as we did not need to create instances for special cases, and helped us fix unexpected and sporadic bugs. The proofs emitted by PACOSE were verified by the pseudo-Boolean proof checker VERIPB [61], and our fuzzing also helped to debug the proof checker.

The experiments were performed on identical machines with an 11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60 GHz CPU and 16 GB of memory. Each benchmark ran exclusively on a machine and the memory limit was set to 14 GB. The time limits were set to 3 600 seconds for solving a MaxSAT instance with PACOSE and to 36 000 seconds for checking the proof with VERIPB. As our benchmark set we used the 558 weighted and 572 unweighted MaxSAT instances from the MaxSAT Evaluation 2023 [47].

Our implementation supports all techniques PACOSE employed in the MaxSAT Evaluation 2023. This means that in addition to the dynamic polynomial watchdog encoding we also implemented proof logging for the binary adder encoding [62] following the approach in [31,59] as well as support for stratification as described in Section 3.4 and for the preprocessing techniques in TRIMMAXSAT [51]. Appendix B discusses TRIMMAXSAT in detail and Appendix C contains detailed experimental results for the default setup in which PACOSE employs heuristics to choose between different encodings. In this section, we focus on the main novelty of this paper, namely proof logging for SIS with the DPW encoding.

To show the viability of enabling proof logging while solving, we analyse the overhead of generating proofs. In Figure 2 we compare the running time of PACOSE with and without proof logging. With proof logging enabled 674 instances were solved within the resource limits, which is 11 fewer instances than without proof logging. Out of the 11 instances that were not solved with proof logging enabled, 9 instances failed due to the memory limit and 2 instances due to the time limit. For the solved instances, PACOSE with proof logging was on average 1.93× slower than without proof logging. About 90% of the solved instances were solved at most 5.26× more slowly with proof logging enabled. This overhead for solving is to some extent caused by our shadow circuits approach. While we demonstrate that shadow circuits can be used to justify the without loss of generality reasoning in Pacose, it remains to investigate whether there is a better approach. It is important to note, though, that the average overhead of 1.93× is heavily biased by small instances: the cumulative solving time of all 674 instances, with proof logging is only 1.32× the cumulative solving time without proof logging. This suggests that proof logging overhead decreases for harder instances.

For proof logging to be maximally useful in practice, it is also desirable that it should be possible to check generated proofs within a time limit that is some small constant factor of the solving time for the instance. To evaluate the efficiency of proof checking, we compared the running time of Pacose with proof logging enabled with the running time of VeriPB, with results plotted in Figure 3. Out of the 674 instances solved by Pacose with proof logging, 592 were successfully checked by VeriPB, but 53 instances failed due to the memory limit and 29 instances due to the time limit. On average, checking the proof with VeriPB was 22.5× slower than solving and generating the proof with Pacose. 90% of the proofs were checked within 100× the running time of Pacose. These results for checking are in line with what has been reported in other works on proof logging for MaxSAT [9, 59]. While there is certainly room for further improvements, this shows that proof logging and checking is viable. It should also be emphasized that the only sources of problems for VeriPB were the time and memory limits – other than that all proofs were successfully checked.

6 Conclusion

In this paper, we demonstrate how to design proof logging for solution-improving MaxSAT solving using the dynamic polynomial watchdog encoding. This turns out to be surprisingly challenging, mainly due to the heavy use of reasoning without loss of generality. To understand the correctness of this reasoning at a human level is one thing, but convincing a proof checker by producing machine-verifiable proofs is quite another. What we show is that by combining the redundance-based strengthening rule and the strengthening-to-core mode in VeriPB, together with a technique we call shadow circuits for having more expressive witnessing capabilities, we are able to devise efficient pseudo-Boolean proof logging techniques.

We have implemented our approach in the state-of-the-art MaxSAT solver Pacose. Our experimental evaluation shows that while enabling proof logging is feasible, it does incur a non-negligible overhead in solving time. Moreover, the time needed to check the generated proofs is several times larger than the time needed to generate them, suggesting that more efficient algorithms and more optimized engineering are needed in VeriPB. This is not so surprising, since the focus of VeriPB development so far has been on providing support for certifying algorithms in combinatorial optimization paradigms previously beyond the reach of proof logging, rather than on optimizing the proof checker code base.

The addition of Pacose to the collection of certifying MaxSAT solvers using VeriPB proofs provides further support to the hypothesis that pseudo-Boolean proof logging hits a sweet spot for MaxSAT solving, being rich enough to support a wide variety of solving algorithms and complex reasoning tricks, but still being simple enough to support even formally verified proof checking as in [13, 33, 37].

We believe that in the longer term VeriPB can have a strong positive impact on the reliability and robustness of MaxSAT solvers. In the other direction, MaxSAT solving is likely to provide excellent benchmarks and performance challenges to further improve pseudo-Boolean proof logging and checking. Our suggestion for speeding up these developments is to introduce a certifying track in the yearly MaxSAT Evaluation [46].

References

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

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

3 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 24, pages 929–991. IOS Press, 2nd edition, February 2021.

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

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

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

7 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.

8 Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesande. Experimental Repository for "Certifying Without Loss of Generality Reasoning in Solution-Improving Maximum Satisfiability", June 2024. `doi:10.5281/zenodo.10826301`.

9 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.

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

11 Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2nd edition, February 2021.

12 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*.

13 Bart Bogaerts, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. Documentation of VeriPB and CakePB for the SAT competition 2023. Available at `https://satcompetition.github.io/2023/checkers.html`, March 2023.

14 Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8-9):606–618, June 2007. Extended version of paper in *SAT '06*.

15 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.

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

17 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.

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

19 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.

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

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

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

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

24 Leon Eifler and Ambros Gleixner. A computational status update for exact rational mixed integer programming. *Mathematical Programming*, 197(2):793–812, February 2023.

25 Salomé Eriksson and Malte Helmert. Certified unsolvability for SAT planning with property directed reachability. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling*, pages 90–100, October 2020.

26 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.

27 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.

28 Mathias Fleury. *Formalization of Logical Calculi in Isabelle/HOL*. PhD thesis, Universität des Saarlandes, 2020. Available at `https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/28722`.

29 Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT '06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, August 2006.

30 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.

31 Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Certified CNF translations for pseudo-Boolean solving. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25, August 2022.

32 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.

33 Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving. In *Proceedings of the 368h AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, February 2024.

34 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.

35 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.

36 Alexander Hoen, Andy Oertel, Ambros Gleixner, and Jakob Nordström. Certifying MIP-based presolve reductions for 0–1 integer linear programs. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14742 of *Lecture Notes in Computer Science*, pages 310–328. Springer, May 2024.

37 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.

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

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

40 Michal Karpinski and Marek Piotrów. Encoding cardinality constraints using multiway merge selection networks. *Constraints*, 24(3–4):234–251, October 2019.

41 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.

42 Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A framework for certified Boolean branch-and-bound optimization. *Journal of Automated Reasoning*, 46(1):81–102, January 2011.

43 Marcus Leivo, Jeremias Berg, and Matti Järvisalo. Preprocessing in incomplete MaxSAT solving. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI '20)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 347–354, August-September 2020.

44 Chu Min Li and Felip Manyà. MaxSAT, hard and soft constraints. 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 23, pages 903–927. IOS Press, 2nd edition, February 2021.

45 Norbert Manthey, Tobias Philipp, and Peter Steinke. A more compact translation of pseudo-Boolean constraints into CNF such that generalized arc consistency is maintained. In *Proceedings of the 37th Annual German Conference on Artificial Intelligence (KI '14)*, volume 8736 of *Lecture Notes in Computer Science*, pages 123–134. Springer, September 2014.

46 MaxSAT evaluations: Evaluating the state of the art in maximum satisfiability solver technology. https://maxsat-evaluations.github.io/.

47 MaxSAT evaluation 2023. https://maxsat-evaluations.github.io/2023, July 2023.

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

49 António Morgado and João P. Marques-Silva. On validating Boolean optimizers. In *Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI '12)*, pages 924–926, November 2011.

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

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

52 Tobias Paxian, Sven Reimer, and Bernd Becker. Dynamic polynomial watchdog encoding for solving weighted MaxSAT. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*, volume 10929 of *Lecture Notes in Computer Science*, pages 37–53. Springer, July 2018.

53 Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Towards bridging the gap between SAT and Max-SAT refutations. In *Proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI '20)*, pages 137–144, November 2020.

54 Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. A proof builder for Max-SAT. In *Proceedings of the 24th International Conference on Theory and Applications of Satisfiability Testing (SAT '21)*, volume 12831 of *Lecture Notes in Computer Science*, pages 488–498. Springer, July 2021.

55 Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Proofs and certificates for Max-SAT. *Journal of Artificial Intelligence Research*, 75:1373–1400, December 2022.

56 Gabriele Röger. Towards certified unsolvability in classical planning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI '17)*, pages 5141–5145, August 2017.

57 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.

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

59 Dieter Vandesande. Towards certified MaxSAT solving: Certified MaxSAT solving with SAT oracles and encodings of pseudo-Boolean constraints. Master's thesis, Vrije Universiteit Brussel (VUB), 2023. URL: https://researchportal.vub.be/nl/studentTheses/towards-certified-maxsat-solving.

60 Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. In *Proceedings of the 16th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '22)*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, September 2022.

61 VeriPB: Verifier for pseudo-Boolean proofs. https://gitlab.com/MIAOresearch/software/VeriPB.

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

63 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.

A Formalization of the Proof Logging of SIS with the DPW

In this appendix, we provide formal details on the claims made in the main body of the paper. In the proofs, we follow the same notation. The formalization of the reasoning in the coarse convergence is discussed in Section 4.2, here we discuss the other phases.

A.1 Coarse Convergence

Our first proposition formalizes the wlog performed during the coarse convergence phase.

▶ **Proposition 5** (Proposition 4, restated). *Assume the definition of z_k has been derived and a complete shadow circuit for $T = 0$ has been introduced. Furthermore assume the constraint*

$$O \geq 1 + k \cdot 2^P \tag{4}$$

has been derived. The constraint $\overline{z}_k \geq 1$ can be derived using redundance-based strengthening with witness

$$\omega = T \mapsto 0, Y \mapsto Y^{T=0}.$$

The notation for the witness in this proposition is a shorthand for the mapping that sends each variable t_i to 0 and every introduced circuit variable y to the corresponding shadow circuit variable $y^{T=0}$.

Proof. To verify this is indeed possible, we need to show that from

$$\mathcal{C} \cup \mathcal{D} \cup \{z_k \geq 1\}$$

we can derive the following constraints:

- $\overline{z}_k{\restriction_\omega} \geq 1$; in other words we need to show that $\overline{z}_k^{T=0} \geq 1$ holds. Recall that $z_k^{T=0}$ is defined by the reification

  $$\overline{z}_k^{T=0} \Leftrightarrow O - 0 \geq 1 + k \cdot 2^P.$$

  Adding up one direction of this definition to (4), immediately yields that $\overline{z}_k^{T=0} \geq 1$, as desired.
- $C{\restriction_\omega}$ for each $C \in \mathcal{C}$.
  - If C is a clause in the original input, $C{\restriction_\omega} = C$ and this is trivial.
  - If C is a previously derived solution-improving constraint, also $C{\restriction_\omega} = C$ (since ω does not touch any variable in O.
  - If C is a previously derived constraint of the form $\overline{z}_{k'} \geq 1$ with $k' < k$, this can either be derived analogously to $\overline{z}_k{\restriction_\omega} \geq 1$ or directly from the fact that the definitions of z_k and z_k' immediately imply that $z_k = 0$ implies that $z_{k'} = 0$.
- $O{\restriction_\omega} \geq O$; this is obvious since the variables in O are unaltered by ω. ◀

▶ Remark 6. Proposition 5 assumes the existence of a constraint (4). It can be seen that this constraint is actually a (potentially weakened version of a) non-strict solution improving constraint $O \geq O{\restriction_\alpha}$ where α is a previously found solution. During the coarse convergence phase, this constraint can be obtained by weakening the solution-improving constraint.

At the end of the coarse convergence phase, also the unit clause $z_{k^*} \geq 1$ is derived. This requires no additional proof logging: this clause is obtained by running the SAT solver with the assumption that $z_{k^*} = 0$ and failing. Whenever this is the case; we know that $z_{k^*} \geq 1$ is internally derived by standard conflict analysis; hence this constraint is added to $\mathcal{D}$ without any additional effort.

A.2 Fine Convergence

As with the coarse convergence, the constraints derived during fine convergence that require a justification in the proof are the unit clauses added to the solver. Proving this relies again on redundance-based strengthening and a shadow circuit.

▶ **Proposition 7.** *Assume $\overline{z}_{k^*-1} \geq 1$ has been derived. Let s be any number and assume a complete shadow circuit for $T = s-1$ has been introduced. Furthermore assume the constraint*

$$O \geq s + (k^* - 1) \cdot 2^P \tag{5}$$

has been derived. The constraint $T \geq s-1$ can be derived using redundance-based strengthening with witness

$$\omega = T \mapsto s, Y \mapsto Y^{T=s-1}.$$

Proof. As in the proof of Proposition 4, this yields several proof obligations. The only non-trivial ones are

- Previously derived constraints of this form $T \geq s' - 1$, but they are trivially satisfied under ω since $s \geq s'$.
- The unit clause $\overline{z}_{k^*-1} \geq 1\restriction_\omega$. In other words we need to show that $\overline{z}^{T=s-1}_{k^*-1}$ holds. Recall that $z^{T=s-1}_{k^*-1}$ is defined by the reification

  $$\overline{z}^{T=s-1}_{k^*-1} \Leftrightarrow O - (s-1) \geq 1 + (k^* - 1) \cdot 2^P$$

  which simplifies to

  $$\overline{z}^{T=s-1}_{k^*-1} \Leftrightarrow O - s \geq (k^* - 1) \cdot 2^P.$$

  Now (5) tells us precisely that the right-hand side of this equivalence is satisfied, hence a straightforward cutting planes derivation indeed allows us to conclude that $\overline{z}^{T=s}_{k^*-1} \geq 1$. ◀

▶ Remark 8. Just like Proposition 4, also Proposition 7 does not make use of the model-improving constraint, but rather makes the assumption on O it uses explicit in (5). As before, this turns out to be useful when applying Proposition 7 in the context of stratification.

Proposition 7 will be applied when a solution α is found taking

$$s := O\restriction_\alpha - (k^* - 1) \cdot 2^P.$$

In this case, the solution-improving tells us that

$$O \geq O\restriction_\alpha + 1 = s + (k^* - 1) \cdot 2^P + 1,$$

and (5) is indeed satisfied. Unit clauses are derived if for a certain j, $s \geq 2^P - 2^j + 1$. In this case, the derived constraint $T \geq s - 1$ guarantees that $T \geq 2^P - 2^j$, i.e., that all dominant bits of T up to j must be equal to one. This follows using reverse unit propagation or a straightforward cutting planes derivation.

A.3 Conclusion of Optimality

When the very last call to the SAT solver is unsatisfiable, we need to derive a contradiction in the proof, to complete the proof that the previously best found solution is optimal. We proceed as follows. First, we introduce a fresh variable, let us call it p using the reification

$$p \Leftrightarrow O \geq o^* + 1. \tag{6}$$

Our goal will be to show that p is false, which then allows us to conclude that the objective can no longer be improved, meaning we have indeed proven optimality. Recall that at this point, we have s defined as $s := o^* - (k^* - 1) \cdot 2^P$. The crucial step in our proof is showing that without loss of generality T can be set equal to s. We proceed as follows.

▶ **Proposition 9.** *Assume $\overline{z}_{k^*-1} \geq 1$ and the definition of p have been derived. Furthermore suppose that a shadow circuit for $T = s$ has been introduced. Using redundance-based strengthening with witness*

$$\omega = T \mapsto s, Y \mapsto Y^{T=s}$$

we can derive the PB constraints representing

$$p \Rightarrow T = s, \tag{7}$$

i.e., in normalised form, the constraints

$$s \cdot \overline{p} + T \geq s, \text{ and} \tag{8}$$

$$(2^P - s - 2) \cdot \overline{p} + \sum_{j=0}^{P-1} 2^j \cdot \overline{T}_j \geq (2^P - 1) - s - 1. \tag{9}$$

Proof. The proof for the two constraints is similar. The only proof goal where they differ is showing that the constraint to-be-derived is satisfied under ω, but this is trivial since the witness sets T equal to s by construction.

For all the other proof goals, we can make use the negation of the constraint to be derived (the negation of (8) or of (9)). From this negation, we can directly derive $p \geq 1$. Adding this up to (one direction of (6) yields $O \geq o^* + 1$, i.e., that

$$O \geq s + (k^* - 1) \cdot 2^P + 1. \tag{10}$$

In other words, the conditions of 7 are satisfied. All the other proof obligations are the same as the ones in the proof of that proposition and hence, making use of (10), the proof proceeds identically to the proof of Proposition 7. ◀

In words, Proposition 9 tells us is that *if* the objective is strictly improving on the previously found best value, *then* we can set T equal to s without loss of generality. The SAT solver, however, has in its last call that yielded UNSAT already derived a clause telling us that at least one of the bits of T does not correspond to s. So we can now straightforwardly derive that $\overline{p} \geq 1$ and hence that $O \leq o^*$, which is what we needed for concluding optimality.

B Proof Logging of Additional Techniques Implemented in Pacose

We detail some of the additional search techniques implemented in and how we proof log them. As a minor point, we note for completeness that in addition to the gcd-based criterion described in Section 3.4, Pacose attempts to find more partitions of the objective during stratification via exhaustive search, as illustrated by the following example:

▶ **Example 10.** Consider the objective $O := 14x_1 + 9x_2 + 5x_3 + 2x_4 + 1x_5 + 1x_6$ and the partition $H = \{1, 2, 3\}$ and $L = \{4, 5, 6\}$. According to the gcd-based criterion from Section 3.4, this partition is not viable due to the gcd not aligning with any single divisor that groups the weights cohesively. However, this partition still validly separates the weights of x_1 to x_6 through an alternative method: Define L_C as the set containing all possible summed combinations of weights from L: $L_C := 5, 9, 14, 5 + 9, 5 + 14, 9 + 14, 5 + 9 + 14$. To validate this partitioning, ensure that the total weight W_L from L is at most the difference between any two sums in L_C. This ensures that L forms a consistent grouping, as there is no weight combination of L invalidating a prior result of solving H.

A more in-depth explanation together with a proof can be found in [51]. While certifying the exhaustive search remains interesting future work, we note that it did not result in additional partitions on any of the benchmarks in our evaluation, nor on the weighted instances of the 2019 and 2020 MaxSAT Evaluation.

We would like to mention that a naive approach to certify the exhaustive search would be to derive the desired constraint $O_H \geq O_H{\restriction_\alpha}$ from the weakened constraint $O_H \geq O{\restriction_\alpha} - W_L + 1$ using redundance-based strengthening with an empty witness. As $O_H{\restriction_\alpha}$ is the sum of a subset of the coefficients in O_H and the distance between any two sums is at least W_L, the negation $O_H < O_H{\restriction_\alpha}$ of the desired constraint can only be satisfied if the sum of true literals in O_H is at most $O_H{\restriction_\alpha} - W_L$. As $O{\restriction_\alpha} \geq O_H{\restriction_\alpha}$, the weakened constraint can only be satisfied if the sum of true literals in O_H is at least $O_H{\restriction_\alpha} - W_L + 1$. Hence, there exists no assignment to the variables in O_H for which both constraints are satisfied. To show this we can iterate through every possible assignment α of the variables in O_H and derive the clause excluding this assignment by reverse unit propagation. This step works, as reverse unit propagation for this clause assigns all variables in O_H, which will falsify either the negated constraint or the weakened constraint by the arguments above. Resolving all the clauses will result in a contradiction that proves that $O_H \geq O_H{\restriction_\alpha}$ is implied.

B.1 TrimMaxSAT

TrimMaxSAT [51] is a preprocessing technique applied before the main SIS algorithm in order to decrease the number of literals in the objective that need to be encoded by the DPW and to get a good initial value of the objective. TrimMaxSAT heuristically splits the variables in the objective into partitions and queries the SAT solver for a solution that assigns at least one of the literals in each partition to 1. If such an assignment is found, the objective variables set to 1 are removed from consideration and the number of partitions are decreased. If the partition size is 1 and the SAT solver reports UNSAT, all remaining literals are fixed to 0 for the rest of the search. In other words TrimMaxSAT aims to find objective literals whose negation is implied by the constraints in the formula and fix their value, thus conceptually decreasing the size of the objective under consideration and–as a consequence–also the size of the DPW encoding built over it.

In more detail, assume $\mathcal{L}$ contains the set of objective variables that have not been set to 1 in any solutions found so far during TrimMaxSAT. During an iteration of TrimMaxSAT, $\mathcal{L}$ is partitioned into m subsets $\mathcal{L}^i$ for $i = 1, \ldots, m$. A new variable r is introduced and the clauses $r \Rightarrow (\sum_{\ell \in \mathcal{L}^i} \ell \geq 1)$ for every $i = 1, \ldots, m$ are added to the SAT solver and the proof via redundance-based strengthening to the core set. The SAT solver is then queried under the assumption that r is true. If the result is SAT, the literals in $\mathcal{L}$ assigned to 1 in the obtained solution are removed from the set under consideration and the unit clause $\overline{r} \geq 1$ is added to the solver such that the SAT solver can remove the clauses of the form

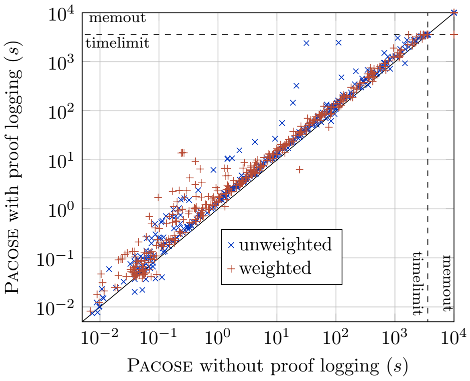


Figure 4 Proof logging overhead for PACOSE using the binary adder encoding.

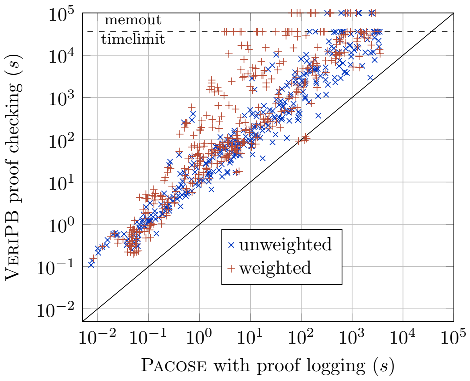


Figure 5 PACOSE vs. VERIPB running time using binary adder encoding.

$r \Rightarrow (\sum_{\ell \in \mathcal{L}^i} \ell \geq 1)$. This unit clause can be derived by redundance-based strengthening with witness $\omega = r \mapsto 0$. If, on the other hand, the result is UNSAT, the unit clause $\overline{r} \geq 1$ is added to the SAT solver and the SAT solver can simplify its clause database. This clause is derived by standard cutting planes reasoning in the conflict analysis by the SAT solver and is therefore added to the derived set in the proof. If in this case $m = 1$, we can also conclude that all literals $\ell \in \mathcal{L}$ are implied to be false. Hence, the solver learns the unit clauses $\overline{\ell} \geq 1$. In order to derive $\overline{\ell} \geq 1$ for each $\ell \in \mathcal{L}^i$, we first introduce the second part of the reification $r \Leftarrow (\sum_{\ell \in \mathcal{L}^i} \ell \geq 1)$ using the redundance rule with witness $r \mapsto 1$ and then use cutting planes reasoning to derive that since r is false, all literals in $\mathcal{L}^i$ must be false. Interestingly, thanks to the use of strengthening-to-core, the unit clause $\overline{r} \geq 1$ derived earlier does not interfere with the derivation of the second direction of the reification.

B.2 Hardening

Hardening refers to the addition of the unit clause l_i for an objective literal l_i if the currently best known solution o^* is larger than the sum of all weights in O excluding w_i. In the proof, the unit clause l_i can be derived easily from the solution-improving constraint and the objective reformulation rule can be used to replace l_i by the constant w_i in the objective.

C Additional Experimental Evaluation

In this appendix, we present some additional experimental analysis with data and plots to give some further insights into proof logging for PACOSE. In Section C.1, we present results for the binary adder encoding that is also used in PACOSE and how detail how well proof logging performs for PACOSE when it heuristically selects the encoding. We present data for an additional approach that uses assumptions instead of unit clauses for fixing variables in the coarse convergence in Section C.2. To better understand the proof logging overhead in PACOSE, we have a deeper look at some additional data for the proof logging process in Section C.3.

C.1 Binary Adder Encoding and Encoding Selection Heuristic

PACOSE also uses the binary adder encoding [62] instead of the DPW encoding. A comparison between these two encodings is beyond the scope of this paper, but as we implemented proof logging for both encodings, we can also have a look at the data for the binary adder

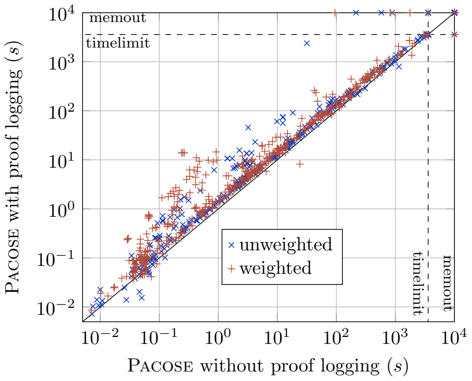


Figure 6 Proof logging overhead for Pacose using heuristic encoding selection.

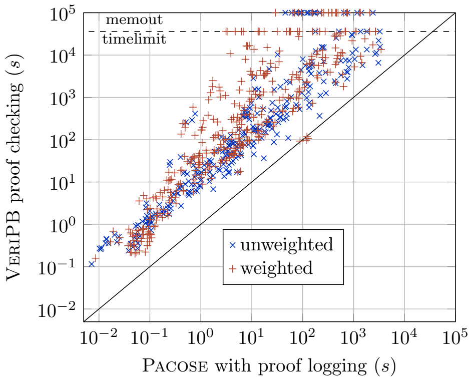


Figure 7 Pacose vs. VeriPB running time using heuristic encoding selection.

encoding. A comparison of solving with and without proof logging for this encoding can be found in Figure 4. With proof logging for the binary adder encoding 722 instances could be solved within the resource limits, which are 6 fewer instances than without proof logging. This also demonstrates that the heuristic for selecting the encoding works, as the number of solved instances for the heuristic is bigger than for any of the two encodings on their own. In the mean, Pacose with proof logging is 1.63× slower than without proof logging. This overhead is smaller than for the DPW encoding, which lead to the conclusion that more work is required to certify the DPW encoding compared to the binary adder encoding.

Out of the 722 instances that were solved with the binary adder encoding, 658 instances were successfully checked by VeriPB within the resource limits. In Figure 5, the running time of Pacose is compared to that of VeriPB. In the mean, VeriPB is 21.1× slower than Pacose for solving the instance with proof logging, which is similar to the DPW encoding. This could mean that the bottleneck for checking the proofs is the implementation of the checker.

Using the default settings, Pacose heuristically selects between the DPW and binary adder encoding. A plot comparing Pacose with and without proof logging in the default settings in Figure 6 and a plot comparing Pacose with proof logging with VeriPB for checking the proof in Figure 7. With this heuristic activated, 698 instances are solved within the resource limits with proof logging enabled and 707 instances without. Pacose with proof logging is 1.83× slower in the mean than Pacose without proof logging. Checking the proof with VeriPB is 21.8× slower than running Pacose with proof logging in the mean.

C.2 Coarse Convergence with Assumptions Instead of Unit Clauses

An alternative approach for representing the information that output variables of the DPW encoding are fixed to a value in the coarse convergence is to use additional assumptions for the SAT solver instead of unit clauses. As we need a shadow circuit to derive each unit clause, we could reduce the number of shadow circuits by using assumptions. The idea is that we add the variable fixing to the assumptions for all future calls to the SAT solver. This approach is supported in Pacose, and we ran additional experiments using this approach.

The following data always use assumptions instead of unit clauses for fixing variables. In Figure 8, Pacose with proof logging is compared to Pacose without proof logging. Using assumptions Pacose with proof logging could solve 666 instances, which is 10 fewer instances than without proof logging. Pacose with proof logging is 1.81× slower than without proof

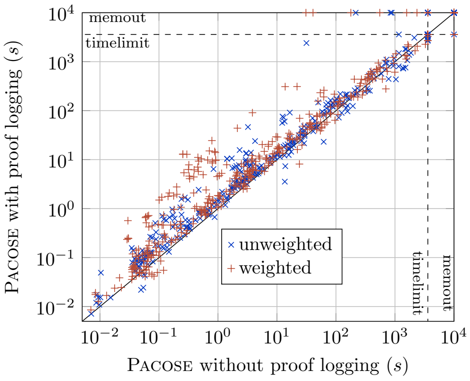


Figure 8 Proof logging overhead for PACOSE using DPW encoding and assumptions.

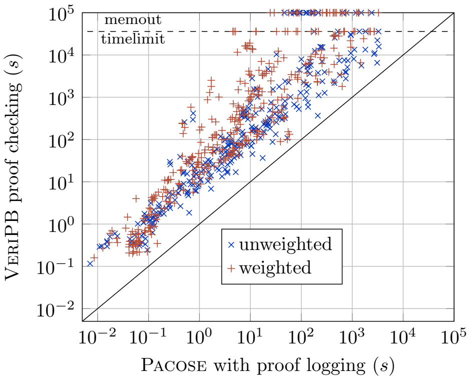


Figure 9 PACOSE vs. VERIPB running time using DPW encoding and assumptions.

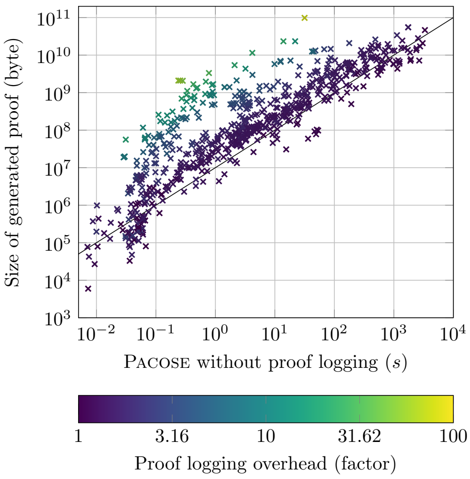


Figure 10 Solving time vs. proof size vs. solving overhead for proof logging for the DPW encoding.

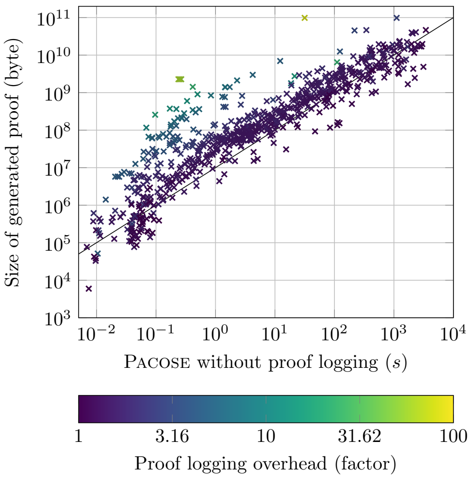


Figure 11 Solving time vs. proof size vs. solving overhead for proof logging for the binary adder encoding.

logging in the mean. This is very similar to PACOSE with the DPW encoding where the variables are fixed by unit clauses and introducing shadow circuits. In the mean, the proof checking is 22.2× slower than solving the instance with proof logging.

It can be concluded that this alternative approach of fixing variables by adding assumptions is about as good as doing the fixing by unit clauses. Hence, it could be that introducing additional shadow circuits for deriving the unit clauses does not slow down the solving a lot, or it is a coincidence that the performance gains are countered by the additional work required for keeping track of the assumptions.

C.3 Proof Logging Overhead Analysis

To get a better understanding of the 1.93× slowdown of PACOSE with proof logging compared to without proof logging, we investigate different causes for the extra running time with proof logging. The idea for doing so is to get insights into how to improve the running time of the solvers.

The expectation is that the proof size scales linearly with the running time of the solver. It would be interesting to look into the instances where this is not the case and if there is a correlation with the solving overhead. We can illustrate this by plotting the solving time against the proof size and colour the marks depending on the overhead as it is done in Figure 10 for the DPW encoding and in Figure 11 for the binary adder encoding. We added a diagonal line representing linear scaling of proof size with running time for better orientation, which is not related to the data at all. It can be seen that for the instances that have a proof size that is significantly bigger than expected, the overhead also seems to increase similarly. To confirm this observation, we compute the correlation of the proof logging overhead and the proof size divided by the solving time. For the DPW encoding we have a correlation of 0.92 and for the binary adder encoding we have a correlation of 0.88, which shows that the two parameters are highly correlated. This mean that the slowdown is due to proof being larger than expected for some instances.

We can conclude with some ideas to improve the performance of proof logging in Pacose. First, the performance can be improved by engineering better data structures to handle the proof logging to increase the speed for writing the proof. This idea only works if we have not reached the maximum persistent disk write speed, which is not the case for our experiments. Second, the proof could be done in a smarter way to reduce the size of the proof, where slow parts of the proof logging could be identified by profiling. Considering that we also have a 1.63× slowdown for the binary adder encoding, the slowdown is not purely caused by the shadow circuits, as they are not used for this encoding.

ParLS-PBO: A Parallel Local Search Solver for Pseudo Boolean Optimization

Zhihan Chen
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

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

Hao Hu
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

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

Abstract

As a broadly applied technique in numerous optimization problems, recently, local search has been employed to solve Pseudo-Boolean Optimization (PBO) problem. A representative local search solver for PBO is *LS-PBO*. In this paper, firstly, we improve *LS-PBO* by a dynamic scoring mechanism, which dynamically strikes a balance between score on hard constraints and score on the objective function.

Moreover, on top of this improved *LS-PBO*, we develop the first parallel local search PBO solver. The main idea is to share good solutions among different threads to guide the search, by maintaining a pool of feasible solutions. For evaluating solutions when updating the pool, we propose a function that considers both the solution quality and the diversity of the pool. Furthermore, we calculate the polarity density in the pool to enhance the scoring function of local search. Our empirical experiments show clear benefits of the proposed parallel approach, making it competitive with the parallel version of the famous commercial solver *Gurobi*.

2012 ACM Subject Classification Computing methodologies → Parallel algorithms; Theory of computation → Randomized local search

Keywords and phrases Pseudo-Boolean Optimization, Parallel Solving, Local Search, Scoring Function, Solution Pool

Digital Object Identifier 10.4230/LIPIcs.CP.2024.5

Supplementary Material *Software*: `https://github.com/shaowei-cai-group/ParLS-PBO.git`
archived at `swh:1:dir:7aa353d93bbe537f550a8c0e4bea165b7cc2f136`

Funding This work is supported by National Key R&D Program of China (2023YFA1009500).

[1] Corresponding author

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 5; pp. 5:1–5:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

With the recent impressive progress in high-performance Boolean Satisfiability (SAT) and Maximum Boolean Satisfiability (MaxSAT) solvers, increasing real-world problems are solved with the Conjunctive Normal Form (CNF) encoding. However, in practice, CNF is ineffective in dealing with cardinality constraints, resulting in its size growing dramatically [3]. As a rich subject in various fields, Pseudo-Boolean Optimization (PBO) provides a better formalization than CNF in expressive power, with the use of Linear Pseudo-Boolean (LPB) constraints. Meanwhile, LPB constraints stay close to CNF and can benefit from advances in SAT solving [33]. The PBO problem is to find an assignment satisfying all LPB constraints that maximizes the objective function given.

Briefly, there are three categories of complete algorithms to solve the PBO problem. The first one is the linear search, which extends the PB solver by adding a constraint enforcing to find a better solution (in terms of the objective function value) when finding a solution satisfying all constraints [2]. Several well-known PBO solvers are based on this idea, including *Sat4j* [22], *RoundingSAT* [14], and *HYBRID* [12]. The second one is Branch-and-Bound, which focuses on the techniques to estimate the lower bounds of the objective value, as the search can be pruned whenever the lower bound is greater than or equal to the upper bound. The symbolic techniques to determine the lower bounds include Maximum Hitting Set [11] and Linear Programming Relaxation [26]. The third one is to call the SAT solvers after encoding PB constraints into the CNF formula, such as *MINISAT+* [13] and *OpenWBO* [31]. Moreover, mixed-integer programming (MIP) solvers can be directly applied to solve the PBO problem, as PB constraints can be treated as 0-1 linear constraints, representative solvers include *SCIP* [5] and *Gurobi* [17].

Complete algorithms often suffer from the scalability issue, which motivates the development of incomplete algorithms. A typical incomplete approach is local search, which has been successfully used in many problems, including SAT [25, 1, 6], MaxSAT [23, 9], etc. Nevertheless, the literature on local search algorithms to solve PBO problem is quite limited. The first local search-based PBO solver was proposed in [24], called *LS-PBO*. This local search solver introduced a constraint weighting scheme and a scoring function considering both hard and soft constraints to select Boolean variables to flip. Later, *LS-PBO* was improved by using a unit propagation-based method to produce better initial assignments [19], resulting in the *DeciLS-PBO* solver. Very recently, on top of *LS-PBO*, Chu et. al. developed *NuPBO* [10], which established the latest state-of-the-art local search based PBO solving. Additionally, Iser et. al. proposed an oracle-based local search approach in the context of PBO [18], which outperforms on various benchmark domains clearly the recent pure stochastic local search approach.

Recently, with the evolution of multi-core processors, parallel solving received growing interest. The SAT competition[2] set up a parallel track from 2009, while Satisfiability Modulo Theories (SMT) competition[3] introduces parallel tracks in 2021. In short, parallel algorithms contain two major directions. The first one is based on the concept of divide-and-conquer, which divides the problem into several sub-problems, and each thread solves sub-problems. For example, Treengeling [15] is a representative SAT solver of this kind. Meanwhile, commercial solvers, such as *CPLEX*[4] and *Gurobi* [17] also implement their parallel versions via this approach. The other parallel approach is to integrate different solvers, including a solver with

[2] `http://www.satcompetition.org/`

[3] `https://smt-comp.github.io/`

[4] `http://www.cplex.com/`

different configurations, and each thread runs a solver. This approach is commonly known as portfolio, which is simple but effective. The portfolio-based parallel SAT solvers, such as *PRS* [8], *P-mcomsps* [16], and *Pakis* [36], dominate the parallel track of SAT competitions in recent years. The parallel MaxSAT solvers [29, 30] based on the portfolio method also demonstrate a strong ability to efficiently solve a large number of problem instances due to the use of complementary search strategies and sharing learned clauses between threads.

In this paper, at first, we improve the typical *LS-PBO* solver by introducing a new dynamic scoring mechanism, which can somehow avoid the local optimum situation even after flipping thousands of variables. This leads to an improved algorithm called *DLS-PBO*. Then, based on the *DLS-PBO*, we develop *ParLS-PBO*, to the best of our knowledge, the first parallel local search-based PBO solver. Our parallel solver runs different local search procedures in the worker threads and maintains a solution pool, which collects good feasible solutions from the working threads and in turn can be used to guide the local search procedures. During the search process, the solution pool is updated by adding new solutions and removing solutions from it. To update the solution pool, we propose a function to measure the feasible solutions found by local search, which considers both the objective value of the solution and the diversity of the solutions in the pool.

The most important part of our parallel solver is how to use the solution pool to help the local search. In this work, this is done in two ways. Firstly, the solutions in the pool can be directly used to help local search when it stagnates for a long time. Specifically, in such a situation, a local search process of a thread restarts from a good feasible solution picked from the pool. Secondly, we calculate the polarity density (the proportions of being 1 and 0) for each variable once a solution is added into the solution pool. This polarity density information is used to enhance the scoring function of local search when picking the variable to flip in each step. The intuition is when a certain polarity (either 1 or 0) of a variable occurs in most high-quality solutions, it brings preference to assign the variable to that polarity.

We carry out experiments to evaluate our algorithms *DLS-PBO* and *ParLS-PBO* on both real-world applications encoded benchmark and standard benchmarks, compared with state-of-the-art solvers including *LS-PBO* [24], *DeciLS-PBO* [19], *NuPBO* [10], *SCIP* [5], *HYBRID* [12], *PBO-IHS* [35], *Gurobi* [17]), and *FiberSCIP* [34]. Our results show that our parallel solver has significantly better performance than all sequential solvers, and competes well with the parallel versions of *Gurobi*. Furthermore, *ParLS-PBO* shows good scalability up to 32 threads as its performance improves with the number of threads.

The remainder of this paper is structured as follows. Section 2 introduces preliminary knowledge. Section 3 analyzes the weakness of *LS-PBO* and introduces an improved solver *DLS-PBO*. Section 4 presents the proposed parallel solver *ParLS-PBO*. Experimental studies are presented in Section 5. Finally, we give some concluding remarks in Section 6.

2 Preliminaries

2.1 Pseudo-Boolean Optimization

A Boolean variable x_i can take only two values *false* and *true*, or equivalently $\{0, 1\}$. A literal l_i is either a variable x_i or its negation $\neg x_i$. Given a set of n Boolean variables $\{x_1, \ldots, x_n\}$, a *linear pseudo-Boolean constraint* (LPB constraint) is formed as follows:

$$\sum_{i=1}^{n} a_i \cdot l_i \triangleright b, \quad a_i, b \in \mathbb{Z}, \quad \triangleright \in \{=, \leq, <, \geq, >\}$$

where a_i is the coefficient for literal l_i, b is called as the degree of the constraint, and $\triangleright$ is one of the classical relational operators. With a given assignment or partial assignment, the constraint is satisfied if its left and right terms satisfy the relational operator. Otherwise, it is unsatisfied.

Moreover, replacing all literals x_i (respectively $\neg x_i$) with negative coefficients with $1-\neg x_i$ (respectively $1 - x_i$), a LPB constraint can be normalized into the following form [33]:

$$\sum_{i=1}^{n} a_i \cdot l_i \geq b, \quad a_i, b \in \mathbb{N}_0^+$$

Given a conjunction of LPB constraints, *Pseudo-Boolean Solving* (PBS) problem is a decision problem to find an assignment such that all constraints are satisfied. *Pseudo-Boolean Optimization* (PBO) problem is an optimized version of the PBS problem, aiming to find an assignment satisfying all constraints with the minimal value of a given objective function. In this paper, we focus on the PBO problem consisting of a conjunction of LPB constraints and a linear objective function. Therefore, a PBO instance subjecting to m LPB constraints has the following form:

$$\begin{aligned} &\min_{\{x_1,\ldots,x_n\}} && \sum_{i=1}^{n} c_i \cdot l_i, \quad c_i \in \mathbb{Z} \\ &\text{subject to:} && \bigwedge_{j=1}^{m} \sum_{i=1}^{n} a_{ji} \cdot l_i \geq b_j, \quad a_{ji}, b_j \in \mathbb{N}_0^+ \end{aligned}$$

where c_i is the objective coefficient for literal l_i.

2.2 A Review of *LS-PBO* Solver

LS-PBO is a representative local search solver for PBO, and serves as the basis of other local search PBO solvers. Briefly, it contains two main ideas: a *Constraint Weighting Scheme* and *Scoring Functions* for guiding the search process.

To solve a standard PBO instance, *LS-PBO* proposed a soft *objective constraint*: $\sum_{i=1}^{n} c_i \cdot l_i < obj^*$, where obj^* indicates the objective value of the best solution in the current run, and other constraints are set as hard. *LS-PBO* uses a weighting technique to increase the weights of falsified constraints, so that the search process is biased toward satisfying them. Specifically, it used dynamic weights (denoted as $w(\cdot)$) to help the search avoid stuck in the local optimum, while increasing the weights of hard constraints to find feasible solutions, and the weight of objective constraint to find better solutions.

Besides, scoring functions are essential in local search algorithms to guide the search process, which typically measures the benefits of flipping a Boolean variable. In *LS-PBO*, the score of flipping a variable x (denoted as $score(x)$) was defined as follow:

$$score(x) = hscore(x) + oscore(x) \tag{1}$$

where $hscore(x)$ indicates the decrease of the total penalty of falsified hard constraints caused by flipping x, and $oscore(x)$ indicates the decrease of the penalty of the objective constraint caused by flipping x. In detail, the penalty of falsifying a hard constraint hc was defined as $w(hc) \cdot \max(0, b - \sum_{i=1}^{n} a_i \cdot l_i)$, and the penalty for the objective constraint oc was defined as $w(oc) \cdot \sum_{i=1}^{n} c_i \cdot l_i$.

3 Improving *LS-PBO* Solver with Dynamic Scoring Mechanism

As introduced in the preliminary, the score of a filliping variable x ($score(x)$) in *LS-PBO* is presented as Equation 1. The algorithm selects the variable with the highest positive score, indicating the biggest decrease in the penalty of hard constraints and objective constraint. A drawback of *LS-PBO* is the lack of dynamic adjustments to the ratio of the soft and hard constraints. If a feasible solution cannot be found within a certain period of time, the search mechanism should adaptively prioritize finding feasible solutions, thereby increasing the ratio attributed to the hard constraints. Conversely, if feasible solutions have been frequently found recently, then it would be beneficial to increase the ratio of the soft constraints to guide the search towards discovering better solutions.

To resolve this drawback, we introduce a new *dynamic scoring function*, denoted as $score^*(x)$, to adjust the significance of $oscore(x)$ for every given K steps (K is a parameter), which is defined as follows:

$$score^*(x) = hscore(x) + p \cdot oscore(x) \tag{2}$$

where p is a dynamic ratio initially set as 1. It would be decreased as p/inc (where $inc > 1$) if no feasible solution is found during the recent K steps, to guide the search towards a feasible solution. Otherwise, respectively, it would be increased as $p \cdot inc$ when a feasible solution is found within the recent K steps, to guide the search process for a better solution.

▶ **Example 1.** Considering a PBO instance:

$$\begin{aligned} \min_{\{x_1,x_2,x_3\}} \quad & 10 \cdot x_1 + 20 \cdot x_2 + 30 \cdot x_3 \\ \text{subject to:} \quad & 2 \cdot x_1 + 3 \cdot x_2 + 4 \cdot x_3 \geq 5 \end{aligned}$$

and suppose current weights $w(hc)$ and $w(oc)$ are 2 and 1. For the given assignment $(x_1 = 1, x_2 = 0, x_3 = 0)$, the corresponding $hscore(\cdot)$ and $oscore(\cdot)$ are as follows:

$\cdot$	x_1	x_2	x_3
$hscore(\cdot)$	-4	6	6
$oscore(\cdot)$	10	-20	-30

Consider the following two situations:

- If feasible solutions are found frequently in recent period, the value of p will gradually increase, guiding the search to lower the cost of the objective constraint. Suppose the current value of p is 2, then $score^*(x_1) = 16, score^*(x_2) = -34, score^*(x_3) = -54$. In this case, x_1 will be picked and flipped, resulting in a decrease of 10 in the cost of the objective constraint. (even if it is not a feasible solution.)
- If the algorithm has not visited feasible solutions for a period (K steps), the value of p will gradually decrease, guiding the search to find feasible solutions. Suppose the current value of p is 0.1. then $score^*(x_1) = -3, score^*(x_2) = 4, score^*(x_3) = 3$. In this case, x_2 will be picked and flipped, resulting in a feasible solution.

We denote the improved version of *LS-PBO* solver with dynamic scoring mechanism as *DLS-PBO*.

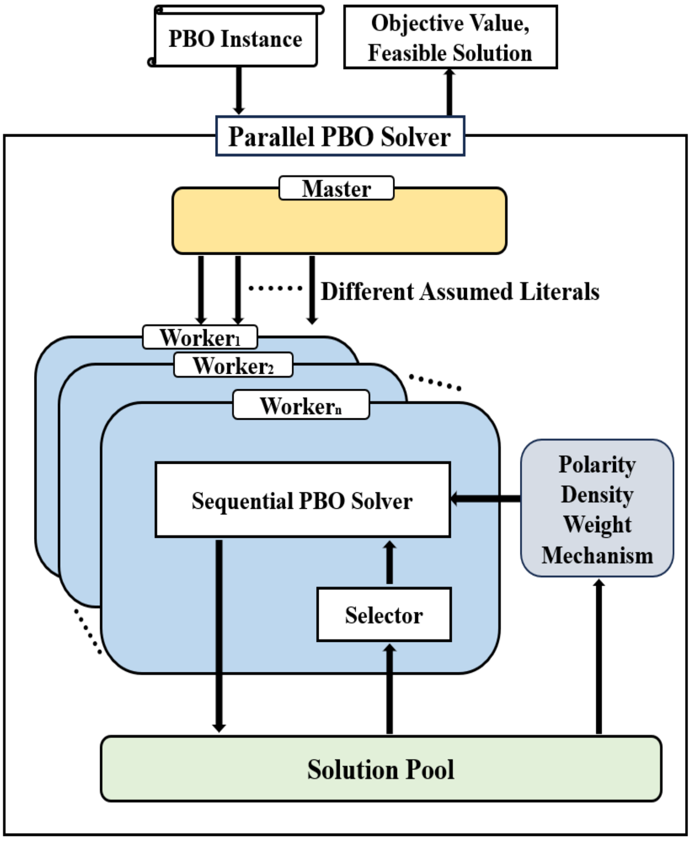


Figure 1 Architecture of the *ParLS-PBO*.

4 Parallel Local Search Solver for Pseudo-Boolean Optimization

In this section, we propose a parallel local search solver *ParLS-PBO*. The architecture of *ParLS-PBO* is shown in Figure 1, which consists of two major contributions: *Solution Pool* and *Polarity Density Weight*. We first describe the global framework of *ParLS-PBO*, then we present the contributions in detail separately.

4.1 Framework of *ParLS-PBO* Solver

As a portfolio-based local search PBO solver, *ParLS-PBO* contains a *master* thread and multiple *worker* threads. The master thread reads the input PBO instance, and then produces different initial partial assignments via *literal assume* technique for worker threads; finally, when the time limit is reached, it outputs the best solution returned from all worker threads. In detail, supposing there are T worker threads, the master thread selects $\lceil \frac{T}{2} \rceil$ random variables. Then, for each variable selected x_i, it generates a positive literal x_i and a negative literal $\neg x_i$. Therefore, it generates T (or $T+1$ if T is odd) different assumed literals in total for worker threads.

Each worker thread receives an assumed literal ℓ (either x_i or $\neg x_i$) and applies the *unit propagation* [28] technique to simplify the formula. Note that a solution found by local search for such a formula can be directly transformed to a solution for the original PBO instance, by adding the assumed literal as the value for the corresponding variable, and adding the value for reduced variables via unit propagation if any. Then the worker thread launches a local search solver to solve the PBO instance. In default, the worker thread launches *DLS-PBO*. To bridge different worker threads, we propose a *Solution Pool* to share high-quality feasible solutions found from different worker threads. When the search process of a worker thread is blocked in the local optimum after flipping a certain number of variables, it attempts to restart with a high-quality feasible solution in the solution pool. Furthermore, we introduce the concept of *Polarity Density Weight* with the intuition of preference of certain polarity of a variable if it occurs in most high-quality solutions.

4.2 Maintaining the Solution Pool

The solution pool aims to collect good feasible solutions, preferring those with more differences. To this end, we consider a mixed quality rating function $r_{mix}(\cdot)$ by measuring two terms together: the quality in objective value, and the diversity. For a feasible solution S, the rating function $r_{mix}(S)$ w.r.t. a solution pool is defined as follows:

$$r_{mix}(S) = rank_{obj}(S) \cdot p^* + rank_{div}(S) \cdot (1 - p^*) \tag{3}$$

where $rank_{obj}(\cdot)$ and $rank_{div}(\cdot)$ represent the ranking of S in the solution pool in objective value and diversity value. Specifically, for the objective value, the solution with the minimal objective value is considered as the best solution, hence its $rank_{obj}(\cdot)$ value is 1. While for the diversity value, the solution with the maximum diversity value is considered the best, thus its $rank_{div}(\cdot)$ value is 1. p^* is a penalty parameter within $[0, 1]$ to adjust the significance of the objective value term and diversity term.

The difference between two solutions is measured as the sum of the number of different polarities, and the diversity value of a solution S w.r.t. The solution pool is measured as the sum of differences between S and all other solutions in the solution pool. Formally,

$$div(S) = \sum_{S' \in \mathcal{P}} Hamming(S, S')$$

When a worker thread finds a new feasible solution S, if the solution pool is not full, then just add it. Otherwise, S replaces the worst one (the solution with the biggest $r_{mix}(\cdot)$ value).

We note that the $r_{mix}(S)$ function in this work resembles a previous population management strategy [7]. We focus on the ranking rather than the value, which can be seen as a normalization.

4.3 Using the Solution Pool to Guide the Search

In this subsection, we discussed how the solution pool guides each worker, including replacing solutions with better solutions from the solution pool when a worker being trapped, and utilizing the variable polarity preference in the solution pool to influence the selection of variables to flip during the search process.

4.3.1 Solution Sharing Strategy

When a worker thread fails to find a better feasible solution for a while, which means that it may be trapped in a local optimum, it selects a feasible solution with a smaller objective value from the solution pool and replaces the current one.

In practice, each worker thread preserves the current best feasible solution (denoted as S^*) as well as the corresponding objective value obj^*. When the search process fails to find a better solution after R steps, it picks a solution from the solution pool as a new starting point.

To prevent excessive overlap of search spaces among various threads, we employ a probability-based method to select solutions in the pool, rather than directly choosing the best solution in the pool. Specifically, let $\{S_1, \ldots, S_k\}$ denotes the set of feasible solutions in the solution pool with objective values not bigger than obj^* (the set will not be empty, as it at least contains S^*), and Δ_i denotes the difference between the objective value of S_i and obj^*. Then the probability of selecting S_i is $\Delta_i / \sum_{j=1}^{k} \Delta_j$.

4.3.2 Polarity Density Weight

Besides using the solutions in the pool to guide the search process directly when it gets stuck, we propose a deeper guiding method, which utilizes a piece of valuable hidden information in the solution pool – the occurrence of polarities (0 or 1) of variables. To measure the effect of this kind of information, we propose the concept of *polarity density weight* for a variable x, denoted as $w_{pd}(x)$, which reflects the preference of certain polarity of x appearing in high-quality solutions.

In detail, for a variable x, $w_{pd}(x)$ is initialized as 1. Once a high-quality feasible solution S is added into the solution pool, $w_{pd}(x)$ will add (respectively, minus) a step value β when positive (respectively, negative) polarity of x appears in S. In fact, via this updating mechanism, the higher (respectively, lower) value in $w_{pd}(x)$ indicates the higher preference of positive (respectively, negative) polarity for x in high-quality solutions. To limit the influence of $w_{pd}(x)$ and avoid possible calculation problems in negative values, we restrict $w_{pd}(x)$ into an interval of $[1-\epsilon, 1+\epsilon]$, where ϵ scales the bound. Therefore, the $w_{pd}(x)$ is updated as follows:

$$w_{pd}(x) = \begin{cases} \max(w_{pd}(x) - \beta, 1-\epsilon), & \text{if } x = 0 \text{ in } S \\ \min(w_{pd}(x) + \beta, 1+\epsilon), & \text{if } x = 1 \text{ in } S \end{cases} \tag{4}$$

The polarity density weight is used to enhance the scoring function of picking a variable to flip during the search process. The resulting enhanced scoring function, denoted as $score^{**}(x)$, is defined as follows:

$$score^{**}(x) = \begin{cases} score^*(x) \cdot w_{pd}(x), & \text{if } x = 0 \text{ in } S_{cur} \\ score^*(x) / w_{pd}(x), & \text{if } x = 1 \text{ in } S_{cur} \end{cases} \tag{5}$$

where S_{cur} is the current assignment maintained by the local search process.

The multiplication of polarity density weight influences the flip of a variable x from 0 to 1, as it increases the combined score if the preference of positive polarity exists ($w_{pd}(x) > 1$) to guide the search process to realise the flip. Respectively, in reverse, the division of polarity density weight influences the flip from 1 to 0.

▶ **Example 2.** Continuing with Example 1, Suppose that most of the solutions that entered the solution pool have the assignment $(x_1 = 1, x_2 = 1, x_3 = 0)$, resulting in $w_{pd}(x_1) = 1.1, w_{pd}(x_2) = 1.1, w_{pd}(x_3) = 0.9$.

For the given assignment $(x_1 = 1, x_2 = 0, x_3 = 0)$ and $p = 1$, the corresponding $score^*(\cdot)$ can be calculated as: $score^*(x_1) = 6, score^*(x_2) = -14, score^*(x_3) = -24$.

Then $score^{**}(x_1) = 6 \div 1.1, score^{**}(x_2) = (-14) \times 1.1, score^{**}(x_3) = (-24) \times 0.9$.

5 Experiments

The experiments are organized as three parts. At first, we focus on comparing *DLS-PBO*, *ParLS-PBO* with state-of-the-art solvers including commercial solvers. Secondly, we analyze the effectiveness of the strategies to guide the search via the solution pool in *ParLS-PBO*. Finally, we present the tendency in performance of *ParLS-PBO* with the increase of the number of threads. Source code and detailed results are made publicly available on GitHub[5].

5.1 Benchmark

- **Real-World**: Three real-world application problems, which are presented in the literature [24], including the Minimum-Width Confidence Band Problem [4][6] (24 instances), the Seating Arrangements Problem [32] (21 instances), the Wireless Sensor Network Optimization Problem [20, 21] (18 instances).
- **MIPLIB**: All satisfiable 0-1 integer programs from the MIPLIB 2017 library and earlier MIPLIB releases[7], which contains 252 instances, provided in the literature [35].
- **PB16**: The OPT-SMALL-INT benchmark from the most recent Pseudo-Boolean Competition 2016[8]. We filter out the duplicated instances that appear in both MIPLIB and PB16, resulting in 1524 instances in the final. PB16 contains different problem categories. We select those representatives (containing more than 30 instances) categories for finer-grained experimental analysis.

5.2 Candidate Methods to Compare

In the sequential track, we compare *DLS-PBO* with 7 state-or-the-art sequential PBO solvers, including 3 local search-based solvers: *LS-PBO*, *DeciLS-PBO* and *NuPBO*, 3 complete non-commercial solvers: *HYBRID*, *PBO-IHS*, and *SCIP* and the commercial solver *Gurobi* (both complete and heuristic versions).

In the parallel track, we compare *ParLS-PBO* with the academic solver *FiberSCIP*, and the parallel version of the commercial solver *Gurobi*.

- *LS-PBO* [24]: the state-of-the-art SLS algorithm for solving PBO[9].
- *DeciLS-PBO* [19]: a recent SLS algorithm based on LS-PBO[10].
- *NuPBO* [10]: a recent SLS algorithm based on LS-PBO, which established the latest state-of-the-art local search based PBO solving[11].
- *HYBRID* [12]: a recent core-guided PBO solver building upon RoundingSAT [14][12].
- *PBO-IHS* [35]: a recent IHS PBO solver building upon RoundingSAT[13].
- *Gurobi* [17]: one of the most powerful commercial MIP solvers. We use both its complete and heuristic versions[14].

[5] `https://github.com/shaowei-cai-group/ParLS-PBO.git`
[6] `http://physionet.org/physiobank/database/mitdb/`
[7] `https://zenodo.org/record/3870965`
[8] `http://www.cril.univ-artois.fr/PB16/bench/PB16-used.tar`
[9] `https://lcs.ios.ac.cn/~caisw/Resource/LS-PBO/`
[10] `https://github.com/jiangluyu1998/DeciLS-PBO` (commit number: 3dce881)
[11] `https://github.com/filyouzicha/NuPBO` (commit number: 821d901)
[12] `https://zenodo.org/record/4043124` (version 2)
[13] `https://bitbucket.org/coreo-group/pbo-ihs-solver` (version 1.1)
[14] `https://www.gurobi.com/solutions/gurobi-optimizer` (version 10.0.0)

- *SCIP* [5]: one of the fastest non-commercial solvers for MIP (the latest version 8.0.1)[15],
- *FiberSCIP* [34]: a parallel non-commercial MIP solvers based on SCIP (the latest version 1.0.0)[16].

We download the latest version of all candidate methods to compare from their published links. In all experiments, we always use their default parameter settings.

5.3 Experimental Settings

DLS-PBO and *ParLS-PBO* are implemented in C++, and compiled with g++ (version 9.2.0) using the option '-O2'. All experiments are carried out on a cluster with two AMD EPYC 7763 CPUs @ 2.45Ghz of 128 physical cores and 1TB memory running the operating system Ubuntu 20.04 LTS (64bit).

As with the previous research on PBO solvers [24, 19], we set the time limit for each run as 300 and 3600 seconds. For each sequential randomized solver, we run 10 times for each instance with different seeds from $\{0, 1, \dots, 9\}$, and select the median of the 10 runs as the final result. Without making any additional claims, the number of CPU cores that can be used for parallel solvers is set as 32.

For parameter tuning, we employed Sequential Model-based Algorithm Configuration (SMAC) [27], conducting the tuning on 300 instances randomly selected from all the benchmarks, with a time limit set to 300 seconds. The parameter values obtained after tuning are listed in Table 1[17].

Table 1 The parameter settings of our solvers.

Parameter	K	R	inc	$poolsize$	p^*	β	ϵ
Value	566024	86295	1.15	18	0.58	0.03	0.144

Referring to the MaxSAT competition and previous research on PBO, we use 2 metrics to evaluate the performance of each solver:

- *#win*: the number of instances that a solver finds the best solution among all solutions output by tested solvers (i.e., the number of winning instances).
- avg_{sc^*}: Since 2017, in the incomplete track of recent MaxSAT Evaluations, the performance of various solvers is measured by competition scores. For an instance and a solver given, the competition score sc is defined as $(1 + cost_{best})/(1 + cost_s)$, where $cost_{best}$ represents the objective value of the best solution found among all solvers, $cost_s$ represents the objective value of the solution found by the given solver. However, in PBO problem, the objective value of a solution may be negative, leading to an incorrect calculation of sc. To address this issue, we modify slightly the definition of competition score:

  $$sc^* = \frac{1 + cost_{best} + \sum_{c_i<0} |c_i|}{1 + cost_s + \sum_{c_i<0} |c_i|}$$

  Adding all negative objective coefficients ensures the competition score of each instance is normalized in $[0, 1]$. We use avg_{sc^*} to denote the average competition score of a solver.

[15] https://www.scipopt.org/index.php#download (version 8.0.1)

[16] https://ug.zib.de/index.php#download (version 1.0.0)

[17] In fact, our solver is not sensitive to the parameter configurations. For example, a simple configuration (K=100000, R=100000, inc=1.1, $poolsize$=10, β=0.1, ϵ=0.15) leads to a performance close to the one in Table 1, with a gap of avg_{sc^*} less than 1%.

We do not use average time as a metric because our primary focus is on the quality of the solution. If the quality of the solutions found is different, then the comparison based on run time would be misleading.

5.4 Performance Evaluations

5.4.1 The Sequential Track

We first compare *DLS-PBO* with *LS-PBO*, and the results are shown in Table 2. *DLS-PBO* significantly improves *LS-PBO* in terms of both $\#win$ and avg_{sc^*} on all the benchmarks.

Further, we evaluate *DLS-PBO* with other PBO solvers, as well as integer programming solvers. The results (Table 3) indicate that *NuPBO* performs best for the Real-world benchmark, while *Gurobi* is the best on MIPLIB and PB16 benchmarks. *DLS-PBO* cannot rival these two solvers, yet it is better than other PBO solvers. We note that the emphasis of this work is to develop an effective parallel method for PBO solvers. We choose *LS-PBO* as the baseline as it is the typical local search PBO solver (*NuPBO* is also developed on top of it). We simply remedy its drawback to obtain *DLS-PBO*, and do not perform other modifications. *NuPBO* was published very recently, and we believe our parallel method can be applied to *NuPBO* as well.

5.4.2 The Parallel Track

The comparative results of our parallel solver *ParLS-PBO* with other parallel solvers are shown in Table 4 (We only show $\#win$ due to the space limit). *ParLS-PBO* gives the best performance on all categories of the Real-World benchmark, and 3 categories of the PB16 benchmark, including Kexu, Logic Synthesis, and Prime.

In terms of the Total instances, *ParLS-PBO* outperforms the non-commercial solver *FiberSCIP*, and is competitive with the commercial solver *Gurobi*. Comparing Table 3 and Table 4, it can be found that the gap between *ParLS-PBO* and *Gurobi* (32 threads) is decreasing compared with the gap between *DLS-PBO* and *Gurobi* (1 thread), which indicates the effectiveness of our solver in parallel solving.

We also observe that *ParLS-PBO* outperforms the best sequential PBO solver *NuPBO* on all benchmarks (44 vs. 32, 171 vs. 156, and 1238 vs. 1002). Although this comparison is unfair (and thus we do not report it in the table), it indicates that by parallelization, the performance of PBO solvers can be significantly improved.

5.5 Effectiveness Analysis

This subsection evaluates the effectiveness of the key strategies of *ParLS-PBO*. In Table 5, we compare *ParLS-PBO* with its 2 variants:

- V_1: to analyze the effectiveness of the solution-pool-based sharing, we modify *ParLS-PBO* by disabling the sharing mechanism and making each thread solve separately.
- V_2: to analyze the effectiveness of the global score mechanism, we modify *ParLS-PBO* by disabling the global score mechanism and using $score^*(x)$ directly in the local search.

As shown in Table 5, *ParLS-PBO* outperforms other variations, confirming the effectiveness of the strategies.

Table 2 Evaluation between *DLS-PBO* and *LS-PBO*.

Benchmark	#Ins	*LS-PBO*		*DLS-PBO*	
		$\#win$	avg_{sc^*}	$\#win$	avg_{sc^*}
cutoff=300s					
Real-World	63	25	0.976	**48**	**0.996**
miplib	252	118	0.777	**182**	**0.836**
PB16	1524	711	0.692	**1124**	**0.776**
Total	1839	854	0.713	**1354**	**0.792**
cutoff=3600s					
Real-World	63	**38**	**0.991**	35	0.988
miplib	252	121	0.81	**186**	**0.863**
PB16	1524	829	0.753	**1189**	**0.825**
Total	1839	988	0.769	**1410**	**0.836**

Table 3 Performance evaluation between *DLS-PBO* and sequential SOTA solvers (The results of *DeciLS-PBO* are not presented due to the space limit. In fact, *DeciLS-PBO* is dominated by *NuPBO* and *DLS-PBO*).

Benchmark	#Ins	*SCIP*	*HYBRID*	*PBO-IHS*	*NuPBO*	*Gurobi*(Comp.)	*Gurobi*(Heur.)	*DLS-PBO*
		$\#win$ avg_{sc^*}	$\#win$ avg_{sc^*}	$\#win$ avg_{sc^*}	$\#win$ avg_{sc^*}	$\#win$ avg_{sc^*}	$\#win$ avg_{sc^*}	$\#win$ avg_{sc^*}
cutoff=300s								
Real-World	63	0 0.126	3 0.109	2 0.266	**46** **0.972**	4 0.289	4 0.292	29 0.977
MIPLIB	252	88 0.614	53 0.572	81 0.741	116 0.854	152 0.838	**165** **0.849**	101 0.803
PB16	1524	810 0.687	663 0.624	882 0.804	980 0.813	1071 0.84	**1072** **0.84**	842 0.741
cutoff=3600s								
Real-World	63	0 0.171	11 0.494	5 0.401	**43** **0.997**	11 0.34	9 0.38	27 0.974
MIPLIB	252	113 0.675	65 0.697	96 0.789	113 0.859	171 0.893	**181** **0.895**	106 0.831
PB16	1524	906 0.734	729 0.715	939 0.814	1012 0.822	1138 0.86	**1144** **0.862**	940 0.797

5.6 Scalability Analysis

In order to analyze the scalability of *ParLS-PBO*, we choose *Gurobi* (complete version) with 32 threads as the comparison baseline to test the performance gap between different threads of *ParLS-PBO*. We report $\#win$ for threads set to $\{4, 8, 16, 32\}$ compared to baseline. As is shown in Figure 2, in each benchmark, $\#win$ is gradually increasing, which verifies the scalability of *ParLS-PBO*.

Table 4 Performance evaluation between *ParLS-PBO* and parallel SOTA solvers.

Benchmark	Category	#ins	*FiberSCIP*	*Gurobi*		*ParLS-PBO*
				Comp.	Heur.	
cutoff=300s						
Real-World	MWCB	24	0	0	0	**24**
	WSNO	18	0	4	4	**18**
	SAP	21	0	0	0	**21**
	Total	63	0	4	4	**63**
MIPLIB	Total	252	113	**190**	180	129
PB16	Factor	192	**186**	**186**	**186**	172
	Kexu	40	6	10	7	**40**
	Logic synthesis	74	71	**73**	72	**73**
	Market split	40	12	**21**	13	5
	Mps	35	30	33	**34**	23
	Numerical	34	13	18	**21**	8
	Prime	156	123	128	129	**131**
	Reduced mps	273	76	145	**150**	39
	Total	1524	898	**1147**	1143	1100
cutoff=3600s						
Real-World	MWCB	24	0	5	2	**20**
	WSNO	18	0	10	10	**18**
	SAP	21	0	0	0	**21**
	Total	63	0	15	12	**59**
MIPLIB	Total	252	129	184	**193**	140
PB16	Factor	192	**186**	**186**	**186**	182
	Kexu	40	14	17	14	**40**
	Logic synthesis	74	72	72	72	**74**
	Market split	40	16	**22**	12	8
	Mps	35	30	**33**	**33**	25
	Numerical	34	13	19	**25**	8
	Prime	156	127	130	131	**132**
	Reduced mps	273	100	150	**160**	43
	Total	1524	995	1198	**1201**	1107

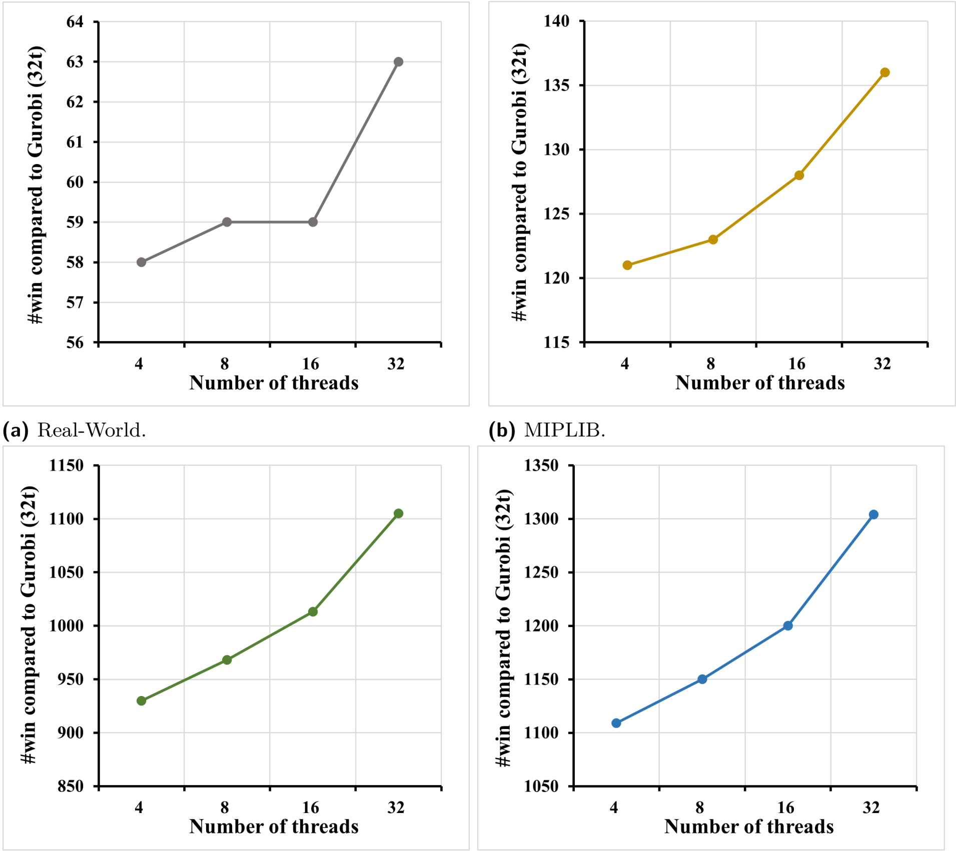
6 Conclusions

We proposed two local search solvers for the PBO problem: *DLS-PBO* and *ParLS-PBO*. *DLS-PBO* is an enhanced version of the *LS-PBO* solver, incorporating a dynamic scoring mechanism. *ParLS-PBO* is a parallel solver with a solution pool collecting good solutions from multiple threads. The solution pool guides the local search process by providing better starting points and utilizing polarity information from high-quality solutions to improve the scoring function. Experimental results show that our parallel solver has significantly better performance than all sequential solvers and exhibits strong competitiveness against the parallel versions of *Gurobi*.

The ideas of this work can be applied to other problems, particularly including SAT and MaxSAT. It is also interesting to implement a distributed version of *ParLS-PBO* for cloud computation.

Table 5 Performance evaluation between *ParLS-PBO* and its variants.

Benchmark	#Ins	V_1 vs. *ParLS-PBO*				V_2 vs. *ParLS-PBO*			
		V_1		*ParLS-PBO*		V_2		*ParLS-PBO*	
		$\#win$	avg_{sc^*}	$\#win$	avg_{sc^*}	$\#win$	avg_{sc^*}	$\#win$	avg_{sc^*}
cutoff=300s									
Real-World	63	28	0.993	**52**	**0.998**	36	0.996	**44**	**0.998**
MIPLIB	252	137	0.857	**199**	**0.868**	166	**0.871**	**185**	0.870
PB16	1524	1095	0.835	**1231**	**0.847**	1168	0.841	**1182**	**0.844**
cutoff=3600s									
Real-World	63	19	0.981	**62**	**1.0**	37	0.995	**48**	**0.999**
MIPLIB	252	145	0.870	**203**	**0.897**	**181**	**0.897**	180	0.893
PB16	1524	1111	0.840	**1251**	**0.854**	1191	0.852	**1214**	**0.852**



(a) Real-World.

(b) MIPLIB.

(c) PB16.

(d) Total.

Figure 2 Scalability Analysis (Time limit is set to 300).

References

1 Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2012. doi:10.1007/978-3-642-31612-8_3.

2 Peter Barth. A davis-putnam based enumeration algorithm for linear pseudo-boolean optimization. Technical report, Max Plank Institute for Computer Science, 1995.

3 Belaid Benhamou, Lakhdar Sais, and Pierre Siegel. Two proof procedures for a cardinality based language in propositional calculus. In Patrice Enjalbert, Ernst W. Mayr, and Klaus W. Wagner, editors, *STACS 94, 11th Annual Symposium on Theoretical Aspects of Computer Science, Caen, France, February 24-26, 1994, Proceedings*, volume 775 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 1994. doi:10.1007/3-540-57785-8_132.

4 Jeremias Berg, Emilia Oikarinen, Matti Järvisalo, and Kai Puolamäki. Minimum-width confidence bands via constraint optimization. In *International Conference on Principles and Practice of Constraint Programming*, pages 443–459. Springer, 2017.

5 Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, et al. The scip optimization suite 8.0. *arXiv preprint arXiv:2112.08872*, 2021.

6 Shaowei Cai, Chuan Luo, and Kaile Su. Ccanr: A configuration checking based local search solver for non-random satisfiability. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 2015. doi:10.1007/978-3-319-24318-4_1.

7 Yuning Chen and Jin-Kao Hao. Memetic search for the generalized quadratic multiple knapsack problem. *IEEE Trans. Evol. Comput.*, 20(6):908–923, 2016. doi:10.1109/TEVC.2016.2546340.

8 Zhihan Chen, Xindi Zhang, Yuhang Qian, and Shaowei Cai. Prs: A new parallel/distributed framework for sat. *SAT COMPETITION 2023*, page 39, 2023.

9 Yi Chu, Shaowei Cai, and Chuan Luo. Nuwls: Improving local search for (weighted) partial maxsat by new weighting techniques. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 3915–3923. AAAI Press, 2023. URL: https://ojs.aaai.org/index.php/AAAI/article/view/25505, doi:10.1609/AAAI.V37I4.25505.

10 Yi Chu, Shaowei Cai, Chuan Luo, Zhendong Lei, and Cong Peng. Towards more efficient local search for pseudo-boolean optimization. In *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

11 Olivier Coudert and Jean Christophe Madre. New ideas for solving covering problems. In Bryan Preas, editor, *Proceedings of the 32st Conference on Design Automation, San Francisco, California, USA, Moscone Center, June 12-16, 1995*, pages 641–646. ACM Press, 1995. doi:10.1145/217474.217603.

12 Jo Devriendt, Stephan Gocht, Emir Demirovic, Jakob Nordström, and Peter J. Stuckey. Cutting to the core of pseudo-boolean optimization: Combining core-guided search with cutting planes reasoning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3750–3758. AAAI Press, 2021. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16492, doi:10.1609/AAAI.V35I5.16492.

13 Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.

14 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.

15 Armin Biere Katalin Fazekas Mathias Fleury and Maximilian Heisinger. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. *SAT COMPETITION*, 2020:50, 2020.

16 Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. Painless: A framework for parallel SAT solving. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 233–250. Springer, 2017. doi:10.1007/978-3-319-66263-3_15.

17 LLC Gurobi Optimization. Gurobi optimizer reference manual, 2021.

18 Markus Iser, Jeremias Berg, and Matti Järvisalo. Oracle-based local search for pseudo-boolean optimization. In Kobi Gal, Ann Nowé, Grzegorz J. Nalepa, Roy Fairstein, and Roxana Radulescu, editors, *ECAI 2023 - 26th European Conference on Artificial Intelligence, September 30 - October 4, 2023, Kraków, Poland - Including 12th Conference on Prestigious Applications of Intelligent Systems (PAIS 2023)*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, pages 1124–1131. IOS Press, 2023. doi:10.3233/FAIA230387.

19 Luyu Jiang, Dantong Ouyang, Qi Zhang, and Liming Zhang. Decils-pbo: an effective local search method for pseudo-boolean optimization. *CoRR*, abs/2301.12251, 2023. doi:10.48550/arXiv.2301.12251.

20 Gergely Kovásznai, Balázs Erdélyi, and Csaba Biró. Investigations of graph properties in terms of wireless sensor network optimization. In *2018 IEEE International Conference on Future IoT Technologies (Future IoT)*, pages 1–8. IEEE, 2018.

21 Gergely Kovásznai, Krisztián Gajdár, and Laura Kovács. Portfolio sat and smt solving of cardinality constraints in sensor network optimization. In *2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 85–91. IEEE, 2019.

22 Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–64, 2010.

23 Zhendong Lei and Shaowei Cai. Solving (weighted) partial maxsat by dynamic local search for SAT. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1346–1352. ijcai.org, 2018. doi:10.24963/ijcai.2018/187.

24 Zhendong Lei, Shaowei Cai, Chuan Luo, and Holger H. Hoos. Efficient local search for pseudo boolean optimization. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 332–348. Springer, 2021. doi:10.1007/978-3-030-80223-3_23.

25 Chu Min Li and Yu Li. Satisfying versus falsifying in local search for satisfiability - (poster presentation). In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 477–478. Springer, 2012. doi:10.1007/978-3-642-31612-8_43.

26 Stan Y. Liao and Srinivas Devadas. Solving covering problems using lpr-based lower bounds. In Ellen J. Yoffa, Giovanni De Micheli, and Jan M. Rabaey, editors, *Proceedings of the 34st Conference on Design Automation, Anaheim, California, USA, Anaheim Convention Center, June 9-13, 1997*, pages 117–120. ACM Press, 1997. doi:10.1145/266021.266046.

27 Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022.

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.

29 Ruben Martins, Vasco Manquinho, and Inês Lynce. Exploiting cardinality encodings in parallel maximum satisfiability. In *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, pages 313–320. IEEE, 2011.

30 Ruben Martins, Vasco Manquinho, and Inês Lynce. Parallel search for maximum satisfiability. *AI Communications*, 25(2):75–95, 2012.

31 Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-wbo: A modular maxsat solver,. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, 2014. doi:10.1007/978-3-319-09284-3_33.

32 Ruben Martins and Justine Sherry. Lisbon wedding: seating arrangements using maxsat. *MaxSAT Evaluation*, pages 25–26, 2017.

33 Olivier Roussel and Vasco M. 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.

34 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.

35 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.

36 Rodrigue Konan Tchinda and Clémentin Tayou Djamegni. Hkis, hcad, pakis and painless exmaplelcmdistchronobt in the sc21. *SAT COMPETITION*, 2021:26, 2021.

Deep Cooperation of Local Search and Unit Propagation Techniques

Xiamin Chen
Shanghai University of Finance and Economics, China

Zhendong Lei[1]
Huawei Taylor Lab, Shanghai, China

Pinyan Lu[2]
Shanghai University of Finance and Economics, Shanghai, China
Huawei Taylor Lab, Shanghai, China

Abstract

Local search (LS) is an efficient method for solving combinatorial optimization problems such as MaxSAT and Pseudo Boolean Problems (PBO). However, due to a lack of reasoning power and global information, LS methods get stuck at local optima easily. In contrast to the LS, Systematic Search utilizes unit propagation and clause learning techniques with strong reasoning capabilities to avoid falling into local optima. Nevertheless, the complete search is generally time-consuming to obtain a global optimal solution. This work proposes a deep cooperation framework combining local search and unit propagation to address their inherent disadvantages. First, we design a mechanism to detect when LS gets stuck, and then a well-designed unit propagation procedure is called upon to help escape the local optima. To the best of our knowledge, we are the first to integrate unit propagation technique within LS to overcome local optima. Experiments based on a broad range of benchmarks from MaxSAT Evaluations, PBO competitions, the Mixed Integer Programming Library, and three real-life cases validate that our method significantly improves three state-of-the-art MaxSAT and PBO local search solvers.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization

Keywords and phrases PBO, Partial MaxSAT, LS, CDCL

Digital Object Identifier 10.4230/LIPIcs.CP.2024.6

Funding *Pinyan Lu*: National Key R&D Program of China (2023YFA1009500).

1 Introduction

The Maximum Satisfiability (MaxSAT) and Pseudo-Boolean Optimization (PBO) are two fundamental and important constraint optimization problems. The Maximum Satisfiability problem (MaxSAT) is the optimization version of the Satisfiability problem (SAT). In general MaxSAT problems, clauses are divided into hard and soft clauses, and each soft clause has an associated weight. The goal of MaxSAT is to find an assignment that satisfies all hard clauses and maximizes the total weight of satisfied soft clauses. PBO consists of a set of pseudo-Boolean constraints and an objective function aiming to find a solution that satisfies all pseudo-Boolean constraints while minimizing the objective function. With the continuous improvements, MaxSAT and PBO solvers have broad applications in real-world problems from operations research, economics, manufacturing, etc. [19, 35, 37].

[1] X. Chen and Z. Lei - The authors are considered to have equal contributions

[2] Corresponding author.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 6; pp. 6:1–6:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Existing practical algorithms for MaxSAT and PBO can be classified into two categories: complete and incomplete methods. Local search is one of the most important incomplete methods and has been shown to be effective for solving many combinatorial optimization problems. Many advanced techniques have been proposed to enhance the performance of local search algorithms for MaxSAT, such as variable selection heuristics [8], clause weighting [11, 26], and Multi-armed Bandit soft clause selection [38]. Recently, local search algorithms for PBO have also achieved breakthroughs by using well-designed scoring functions for variable selection [12, 27]. Generally, local search algorithms converge quickly, enabling them to find high-quality feasible solutions within a reasonable time. As a result, they are widely employed for solving large instances, including real-world applications. However, local search algorithms often get stuck in local optima easily due to their limited reasoning ability and lack of global information.

On the contrary, complete algorithms adopt techniques that utilize global information and powerful reasoning, such as unit propagation (UP) and conflict-driven clause learning (CDCL) [34], to seek optimal solutions. Core-guided algorithms and branch-and-bound algorithms are two commonly used complete methods for solving PBO [10, 16, 17, 18, 33, 36] and MaxSAT [1, 2, 5, 6, 14, 15, 20, 23, 28, 31, 32] problems, where UP and CDCL play critical roles in improving the performance of these solvers. However, these methods are typically too time-consuming to solve some large-scale industrial instances.

Numerous research studies have been conducted to combine incomplete and complete methods to develop effective solution approaches for the constraint optimization problems. Some of these works utilize CDCL as the main solver, while LS is invoked to provide a search heuristic [13, 22, 30], or to perform deep search at a branching node [3, 9, 24, 28]. Other approaches use local search as the primary solver, with CDCL called upon for preprocessing [21, 25, 29], building initial assignments [7], or solving sub-problems as black boxes [4].

In sharp contrast to these combined methods, our work proposes a new search framework, in which local search acts as the main solver, and unit propagation techniques are used to help local search algorithms escape from local optima. Specifically, we first design a mechanism to detect when the local search algorithm is trapped in a local optimum. Upon detection, unit propagation is invoked to change the current assignment. Since binary constraints are common in MaxSAT and PBO instances, unit propagation will guide the solver into boarder search spaces, thereby offer more chance to jump out of the local optimum. Finally, we propose an acceptance criterion to determine whether to accept the propagated solution.

We have applied our new methods to improve the state-of-the-art MaxSAT solvers SATLike and NuWLS, as well as PBO solvers LS-PBO and NuPBO. Validation was conducted across it in MaxSAT Evaluations, PBO competitions, the Mixed Integer Programming Library, and three real-life cases. Experimental results demonstrate that the combination of UP technology significantly enhances the performance of local search algorithms.

2 Preliminary

2.1 Preliminaries Definitions and Notations

Given a set of n Boolean variables $x_1, x_2, \ldots, x_n$, a *literal* l_i is either a variable x_i (which is called a positive literal) or its negation $\overline{x_i} = 1 - x_i$ (which is called a negative literal). An *assignment* α is a mapping that assigns each variable a value (0 or 1).

A *clause* C_i of length k_i is a disjunction of k_i literals (i.e., $C_i = l_{i1} \vee l_{i2} \vee \cdots \vee l_{ik_i}$). A conjunctive normal form (CNF) formula $F = C_1 \wedge C_2 \wedge C_3 \cdots \wedge C_m$ is a conjunction of clauses. Given an assignment α, a clause is satisfied by α if it contains at least one true literal, and is falsified otherwise.

The Partial MaxSAT (PMS) problem is defined on a CNF formula, in which some clauses are designated as hard clauses and the reminder as soft. The objective is to find an assignment that satisfies all hard clauses and maximizes the total number of satisfied soft clauses. In Weighted PMS (WPMS), each soft clause is assigned a positive integer weight, and the goal is to satisfy all hard clauses while maximizing the total weight of satisfied soft clauses.

The Pseudo-Boolean Optimization (PBO) problem consists of a set of Pseudo-Boolean (PB) constraints and an objective function. A normalized Pseudo-Boolean constraint is represented as $\hat{C}_i : \sum_j a_{ij} \cdot l_{ij} \geq d_i$, where $a_{ij}, d_i \in \mathbb{Z}^+$ and l_{ij} are literals. The objective function is in the form $\sum_j c_{oj} \cdot l_{oj}$. The goal of PBO is to find an assignment that satisfies all PB constraints while minimizing the objective function.

Partial MaxSAT (PMS) can be considered as a specialization of PBO. Therefore, Partial MaxSAT instances can be readily encoded into PBO instances. Given a PMS instance with a set of hard clauses $H = \{C_1 \wedge C_2 \wedge \cdots \wedge C_m\}$, where $C_i = l_{i1} \vee l_{i2} \vee \cdots \vee l_{ik_i}$ and a set of soft clauses $S = \{S_1 \wedge S_2 \wedge \cdots \wedge S_n\}$, where $S_j = (w_j, l_{j1} \vee l_{j2} \vee \cdots \vee l_{jk_j})$ and w_j is the weight of the soft clause, the equivalent PBO format is constructed by transforming C_i's into constraints $\hat{C}_i : l_{i1} + l_{i2} + \cdots + l_{ik_i} \geq 1$, and S_j's into $\hat{C}_j : l_{j1} + l_{j2} + \cdots + l_{jk_j} + y_j \geq 1$, where y_j are auxiliary variables representing satisfaction of soft clauses. The objective function is $\sum_j w_j y_j$, which is weighted sum of auxiliary variables. Thus techniques used in PBO can be effectively adapted to PMS instances. For clarity and consistency, thorough the reminder of this paper, we discuss these concepts in the field of PBO.

2.2 Local Search and Unit Propagation

As shown in Algorithm 1, a local search solver maintains a complete assignment and keeps track of the best solution found during the search. In each step, the local search algorithm modifies the complete assignment locally (i.e., flips the value of a variable) to find a better solution and it returns the best solution when the termination condition is reached (lines 3-16).

State-of-the-art LS solvers often employ clause-weighting techniques. A dynamic weight is attached to each constraint to indicate its importance or difficulty. If an assignment α fails to satisfy a constraint $C_i : \sum_j a_{ij} l_{ij} \geq d_i$, the violation of C_i is defined as $\mathit{Violation}(C_i) = d_i - (\sum_j a_{ij} l_{ij})|_\alpha$, and the violation is 0 otherwise. In this way, the quality of an assignment can be evaluated by *Punishment*, which is defined as the weighted sum of *Violation* of all constraints. Specially, the objective function is treated as a never-satisfied constraint $C_o : \sum_j c_{oj} l_{oj} \geq M$ (where M is sufficiently large). In this way the objective is also counted into *Punishment*. The change in *Punishment* before and after a variable flip is then defined as the *score* of the corresponding variable.

In each step, if there exists a variable meeting the greedy heuristics, it is selected and flipped to decrease the Punishment(lines 8-9). If no such variable exists, or in other words the search get stuck, some local-optimum-escaping heuristics will be applied (lines 11-13). Typically, these heuristics involve increasing the weights of unsatisfied hard constraints and performing a random flip.

Unit propagation is the deduction of assignments over constraints. In the scope of MaxSAT, a *unit* clause is a clause containing only one literal. Generalizing this concept, a *unit* PB constraint is a constraint that cannot be satisfied unless one certain literal is satisfied. Similarly, a constraint is *binary*, if it becomes unit after one literal is falsified.

▶ **Example 1.** $2x_1 \geq 1$ and $3x_1 + x_2 + x_3 \geq 4$ are unit constraints, since they are unsatisfied unless $x_1 = 1$.

Algorithm 1 Typical Local Search Algorithm.

```
Input: A given instance $F$, $cutoff$
Output: An assignment $\alpha$ of $F$ and its cost
$\alpha$ := an initial complete assignment.
$\alpha^* := \emptyset$.
while elapsed time < $cutoff$ do
    if $cost[\alpha] < cost[\alpha^*]$ then
        $\alpha^* := \alpha$.                                      ▷ update best solution
    end if
    if meet greedy heuristics then
        $v$ := a variable is picked accordingly.
        $\alpha := \alpha$ with $v$ flipped.                             ▷ greedy flips
    else
        update clause/constraint weights.                      ▷ weight adjustments
        $v$ := a variable is picked according to local-optima-escaping heuristics.
        $\alpha := \alpha$ with $v$ flipped.                             ▷ random flips
    end if
end while
Return ($\alpha^*$, $cost^*$).
```

▶ **Example 2.** $2x_1 + x_2 + x_3 \geq 2$ is a binary constraint, since fixing $\overline{x_1} = 1$ leads to a unit constraint implying $x_2 = x_3 = 1$.

Generalized unit propagation works as follows: First, the algorithm assumes a literal l to be true, or in other words assigns a variable v a value of 0 or 1. Then, the algorithm accesses all the constraints that involve v and substitutes v with its value. If any constraints become unit, the value of another variable can be derived, and the derived variables will be assigned. The algorithm applies this rule iteratively until no new variable assignments can be derived or a conflict occurs, which means there exist some constraints that cannot be satisfied under current assignment. If no conflict occurs, the derived literals form the *Implication* of l, denoted as $\text{Imply}(l)$.

▶ **Example 3.** Suppose there is a constraint $x_1 + x_2 + x_3 \geq 2$, then $\{x_2, x_3\} \subset \text{Imply}(\overline{x_1})$.

3 Main contribution

In this section, we present in detail how Systematic Search works and cooperates with local search. While we focus on PBO problems in this section, the method can be easily extended to PMS problems. The source code will be available at `https://github.com/SystematicSearch`.

3.1 Deep Cooperation of Local Search and Unit Propagation

The main drawback of local search is that it converges to a sub-optimal solution quickly, but the subsequent improvement is exceedingly challenging, because it easily gets trapped in a local optimum. We ran NuWLS, one of the state-of-the-art local search solver for PMS, for 300 seconds to solve the MSE benchmarks, and summarized the scores at different cutoff times in Figure 1 (benchmark and scoring will be introduced in Section 4). As shown in

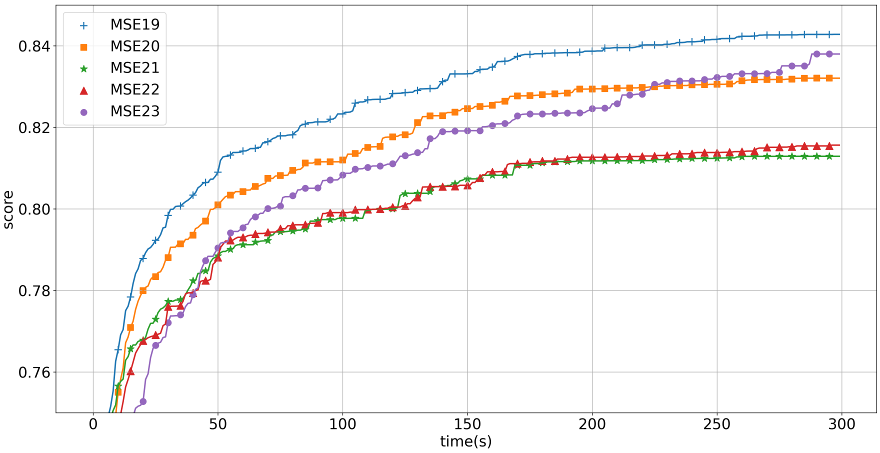


Figure 1 Baseline behavior: scores at different cutoff time of NuWLS.

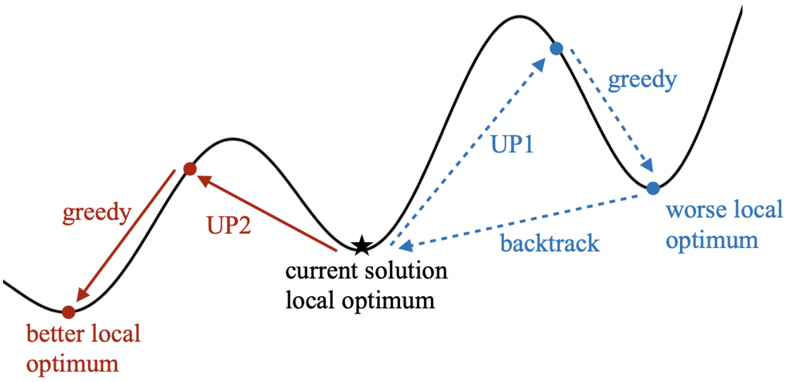


Figure 2 The horizontal direction represents the search direction, and the vertical direction indicates the *Punishment* of each assignment. Moving in the curve simulates the flips of local search.

the figure, the score rushes to 90% of its peak value in less than 10 seconds, but the last 100 seconds contribute only 0.01 to the score. Therefore, our motivation is to spend these seconds on addressing the trapping problem effectively.

To address this problem, we designed a new mechanism through deep cooperation of local search and unit propagation. As illustrated in Figure 2, the general idea of our approach is that, whenever the algorithm get stuck in a local optimum, a variable is picked and flipped together with the unit propagation procedure of this variable. Then, a greedy heuristic procedure is executed until the algorithm reaches another local optimum. If the new optimum is worse than the original one, we backtrack to the original assignment (the dashed line route). Otherwise, if the *Punishment* is reduced, we accept the new assignment (the solid line route). The modified LS framework involves Systematic Search if the original LS solvers get stuck or fail to improve the objectives, as described in Algorithm 2. Theoretically, this mechanism can be extended to all kinds of local search algorithms as long as a proper stuck-detection mechanism is designed.

Algorithm 2 Local Search Algorithm with Systematic Search.

```
Input: A given instance $F$, $cutoff$
Output: An assignment $\alpha$ of $F$ and its cost
 1: $\alpha$ := an initial complete assignment.
 2: $\alpha^* := \emptyset$.
 3: while elapsed time < $cutoff$ do
 4:    if $cost[\alpha] < cost[\alpha^*]$ then
 5:       $\alpha^* := \alpha$.                                        ▷ update best solution
 6:    end if
 7:    if meet Systematic Search invocation criteria then
 8:       $candidates$ := PickCandidates().                            ▷ see Section 3.3
 9:       for all $v \in candidates$ do
10:          $\alpha' :=$ FlipUP$(v, \alpha)$.                          ▷ see Section 3.4
11:          if new assignment satisfy accept criteria then
12:             $\alpha := \alpha'$ and Break.                            ▷ see Section 3.5
13:          end if
14:       end for
15:    else
16:       $\alpha$ : is modified by the original heuristic of the given algorithm.
17:    end if
18: end while
19: Return $(\alpha^*, cost^*)$.
```

Our new local search scheme is illustrated in Algorithm 2 (lines 7-14). When the algorithm meets the systematic search invocation criteria (more details can be found in Section 3.2), which means it gets stuck in a local optimum, the algorithm will run the systematic search. The first step is to pick a candidate starting variable (line 8). Then the algorithm flips it, considers its assignment to be temporarily fixed, and performs the unit propagation procedure for this variable (line 10). Finally the new assignment will be evaluated. If the assignment is improved, we accept it and break (line 12). Otherwise, we backtrack to the last local optimum and try the next candidate. A thorough description of candidate selection, unit propagation, and acceptance criteria will be provided in the rest of this section.

3.2 Invocation of Hybrid Local Search

The hybrid method is involved when the best objective has not been updated for a long time, indicating that the original algorithm could be stuck. Intuitively, there are two favorable situations for invocation: when stuck in a local optimum, or right before restarting.

In the first situation, our new method is called every k times the algorithm reaches a local optimum, where k is initialized to a relatively small integer k_{init}. After each call, if the acceptance criteria are satisfied, indicating the method is effective, k will be reset to k_{init}. If all candidates fail, k will be doubled until it reaches a preset upper bound k_{max}. Theoretically, unit propagation does not need to be performed at every single local optimum for the following reasons. Firstly, since backtracking means flipping some variables twice without actual changes, frequent calls will cause a notable decrease of total valid flips, and may narrow the overall search space in the end. This effect will be discussed in Section 4. Secondly, updating weights plays a crucial role in balancing finding feasible solutions and reducing the objective cost, so we still need a number of weight updates and random flips (lines 11-13 in Algorithm 1). Finally, random flips is still one way to escape from some local optima. By keeping both methods, we can combine their strengths.

Algorithm 3 PickCandidates.

```
Input: current local optimum assignment
Output: a set of variables
Params: #unsat constraints chosen $n_1$, size of candidate $n_2$.
 1: $candidates := \emptyset$.
 2: if current assignment is infeasible then
 3:     for c: up to $n_1$ random unsatisfied constraint do
 4:         for l: all falsified literal in c do
 5:             $x :=$ the variable of $l$.
 6:             $candidates := candidates \cup \{x\}$.
 7:         end for
 8:     end for
 9: else
10:     for $l$: up to $n_1$ falsified literal in objective do
11:         $x :=$ the variable of $l$.
12:         $candidates := candidates \cup \{x\}$.
13:     end for
14: end if
15: $candidates :=$ top $n_2$ elements with highest scores in $candidates$.
16: Return $candidates$.
```

The invocation in the second situation is based on the observation that feasible solutions are not uniformly distributed in the domain but rather cluster together. It is highly possible that a better solution lies near the best-found solution $\hat{\alpha}$, but the search might have chosen a different direction there. In this case, a deep search around the $\hat{\alpha}$ helps to improve the performance. To revisit that neighborhood, we flip to $\hat{\alpha}$, then call unit propagation to choose a search direction to leave $\hat{\alpha}$, followed by *final_steps* steps of the original search method. The restart will be delayed until all attempts are tried.

3.3 Picking Candidate Starting Variables

Our hybrid method begins with flipping one variable, followed by other propagated flips, so the correctness of the first flip is important. Similar to local-optimum-escaping heuristics in LS solvers, we filter some variables from unsatisfied constraints or falsified objective literals, and rank them by *score* to form a candidate set. As shown in Algorithm 3, if the local optimum is infeasible, (lines 2-8), at least one of falsified variable in each unsatisfied constraint needs to be flipped, so **PickCandidates** will visit falsified constraints, caching the falsified literals in *candidates* without duplication. Else, when the local optimum is feasible (lines 9-13), to improve the objective, at least one of the falsified literals in the objective function must be satisfied, so **PickCandidates** randomly picks among them. For the sake of efficiency, **PickCandidates** visits at most n_1 constraints and returns n_2 variables.

3.4 Flipping Based on Unit Propagation

It is challenging to obtain propagated information within a very tight time limit. To achieve efficient unit propagation, we propose a lightweight approach. Suppose we query for $UPList(x_0)$. Instead of calling a complete unit propagation over the formula, the algorithm checks every constraint containing the literal $\overline{x}_0$. For each of these constraints, the algorithm

Algorithm 4 FlipUP.

```
Input: a literal $l$, current assignment $\alpha$
Output: local optimum assignment
 1: $U$ := UPList($\bar{l}$).
 2: $\alpha := \alpha$ with flip $l \to \bar{l}$.                ▷ flip the candidate first
 3: for all $l' \in U$ do
 4:     if current assignments falsify $l'$ then
 5:         $\alpha := \alpha$ with flip $l' \to \overline{l'}$.         ▷ to be consist with UPList
 6:     end if
 7: end for
 8: while meet original greedy heuristics do
 9:     $v$ := a variable is picked accordingly.
10:     flip($v$).                                  ▷ follow the original greedy scheme
11: end while
12: Return $\alpha$.
```

verifies if satisfying x_0 turns the constraint into a unit constraint. If so, we record the propagated literals in *UPList*(x_0). Additionally, if time permits, we will run a breadth-first search on the propagated variables to gather more information. Sometimes a conflict occurs, such as when l and $\bar{l}$ appear simultaneously, or $\overline{x_0} \in$ *UPList*(x_0). These cases indicate that x_0 is an infeasible assignment, so the algorithm returns *UPList*$(x_0) = \{\overline{x_0}\}$ to prompt not to flip $\overline{x_0} \to x_0$. Since the implications of literals are unrelated to current assignments, *UPList* are cached to avoid redundant calculations.

▶ **Example 4.** Suppose a formula $\overline{x}_1 + x_2 \geq 1$, $\overline{x}_1 + x_3 \geq 1$, $\overline{x}_2 + \overline{x}_3 + x_4 \geq 1$, Imply($x_1$) $= \{x_2, x_3, x_4\}$ but *UPList*$(x_1) = \{x_2, x_3\}$, or *UPList*(x_2) $\cup$ *UPList*(x_3) after a breadth first search.

The implementation of unit propagation is shown in Algorithm 4. First, the algorithm queries the *UPList* and flips until all literals in the *UPList* are satisfied (lines 3-7). Then, the search follows the original greedy heuristics and flips until a new local optimum is reached (lines 8-11). Consequently, we extend a single flip into a multi-step search action, which will strengthen the ability to escape from local optima.

The trick of getting the *UPList* works for several reasons. First of all, as an incomplete solver, our *UPList* is not obligated to be strictly complete and can therefore save time. Additionally, there will be greedy search steps after the propagation, providing another opportunity to flip those omitted variables. Finally, binary clauses are common in PMS and PBO instances, so the size of the derived *UPList* is large enough to move the search out of the local optimum, fulfilling our purpose.

3.5 Acceptance Criteria

In the previous text, we discussed that the motivation of our new method is to jump to another local optimum. If a correct candidate variable is chosen, we are supposed to get closer to the global optimum. However, when the candidate is a mistake, we ought to stop searching in that direction, go back, and try the next candidate (line 9 in Algorithm 2). We measure each assignment by *Punishment*, which is consistent with the measure of greedy search and the definition of a local optimum. If the *Punishment* decreases, we have successfully escaped the local optimum, so no more candidates need to be tried. Otherwise, our method ends up in a worse local optimum, the new assignment is discarded by flipping changed variables again, followed by trying the next candidate. Finally, if all candidates fail to lead the search to a better local optimum, it is possible that unit propagation does not work in this situation, and the original local-optima-escaping heuristics will take place. In the next section, we will exhibit the proportion of success and failure of our method in the experiments.

4 Experiment Results

In this section, we implement our hybrid method on four state-of-the-art solvers for PMS and PBO problems, listed as follows:

- **LS-PBO**: The state-of-the-art local search PBO solver, proposed by [27].
- **NuPBO**: A recent PBO solver based on LS-PBO but using different scoring and weighting schemes, proposed by [12].
- **SATLike3.0**: The state-of-the-art local search MaxSAT solver and competition winner, proposed by [7].
- **NuWLS**: A recent MaxSAT solver based on SATLike3.0 but using different scoring and weighting schemes, proposed by [11].

We validate our approach on a wide range of benchmarks, described as follows:

- **CRAFTED**[3]: The crafted combinatorial problem set.
- **MIPLIB**[4]: A set of 0-1 integer linear programming problems.
- **PB16**[5]: The OPT-SMALLINT-LIN benchmark from the latest 2016 pseudo-Boolean competition.
- **Industry**[6]: A combination of some real-world problems, including the seating arrangement problem (SAP), the wireless sensor network optimization problem (WSNO), and the minimum-width confidence band problem (MWCB) [27].
- **MSE19-MSE23**[7]: The benchmarks used in the MaxSAT Evaluations in the last five years.

4.1 Experiment Settings

All competitor solvers are implemented in C++ and compiled with `g++` with the make option `-O3`. All the experiments are run on a workstation with an Intel Xeon Platinum 8380H CPU @ 2.90GHz. Each instance is solved once with a cutoff time of 300 seconds. The parameters of the planted local search solvers are kept the same as those used in [7, 11, 12, 27]. The newly added parameters used in our experiments are $n_2 = 5, k_{init} = 10, k_{max} = 2560$, and $\mathit{final\_steps} = 10000$. We will discuss the sensitivity of results to these parameters and random seeds in Section 4.4.

We conclude the results in two dimensions, the number of winning instances and the average score. Each experiment compares the original algorithm with its unit-propagation-implemented version. For each instance, if the results are different, #win for the superior algorithm is incremented by 1, and #lose for the inferior algorithm is also incremented by 1. The score for each result follows the method used in MSE competitions. Specifically, within

[3] `https://zenodo.org/record/4036016`
[4] `https://zenodo.org/record/3870965`
[5] `http://www.cril.univ-artois.fr/PB16/bench/PB16-used.tar`
[6] `https://lcs.ios.ac.cn/~caisw/Resource/LS-PBO/benchmark/`
[7] `https://maxsat-evaluations.github.io/2023/benchmarks.html`

each experiment set, let v^* denote the best objective found by all competing algorithms. A solver returning objective v receives a score of $(1+v^*)/(1+v)$. This score is 1 if $v = v^*$, less than 1 if it is not the best result, and equals 0 if no feasible solution is found. Given that v^* is non-negative in the context of MaxSAT and PBO problems, scores range between 0 and 1. We report the average score across all benchmarks in this section.

To analyze the performance of our method, we also record several other indices. *accept(%)* represents the acceptance rate, calculated as the percentage of accepted local optima over the total number of candidate attempts. *dist* denotes the average distance, or the Hamming distance between two local optima before and after an accepted *FlipUP* action. *step(%)* indicates the average number of valid step, which is the sum of greedy steps, random flips, and the number of flips in accepted *FlipUP* actions, but excludes rejected propagations and backtrack flips.

4.2 Experiment Results on MaxSAT Benchmarks

Results on SatLike3.0 and NuWLS are shown in Table 1 and 2 respectively. Both solvers perform significantly better with the cooperation of unit propagation, evidenced by the increase in the number of winning instances and the average scores across these benchmarks. The progress of scores over time is depicted in Figure 3. For MSE19-22, the score increase of original NuWLS slows down at around 0.72, whereas our approach delays this slowdown until around 0.76. However, in MSE23, a distinct pattern emerges where efficiency becomes more crucial. The distance between the solid curve and the dotted curve was narrowed to nearly zero twice (around 50s and 200s) and then widened again (around 130s and 290s). We attribute this pattern to the fact that some improvements at these timestamps are delayed because the unit-propagation-implemented iterations are slower than the original ones.

Under the acceptance criteria, approximately 30 to 40 percent of propagation attempts are accepted, resulting in the search jumps from current local minima to around 6 flips away. In contrast, the rejected attempts are purely wastes of time. The average time cost of our method, including calculating *UPList* and performing backtracking, amounts to approximately 15 to 20 percent of the cutoff time. The results depicted in Figure 1 suggest that this amount of time overhead has limited impact on the average score, but there are still some instances to be sensitive to the effective running time. Particularly in MSE23, the score curve showed steep improvements after 200 seconds, indicating that the solver finally finds a feasible solution and turns the score from 0 to 1. In these cases, the unit propagation integrated version ends up with no feasible solution, which explains for the score decrease in MSE23 in Table 1.

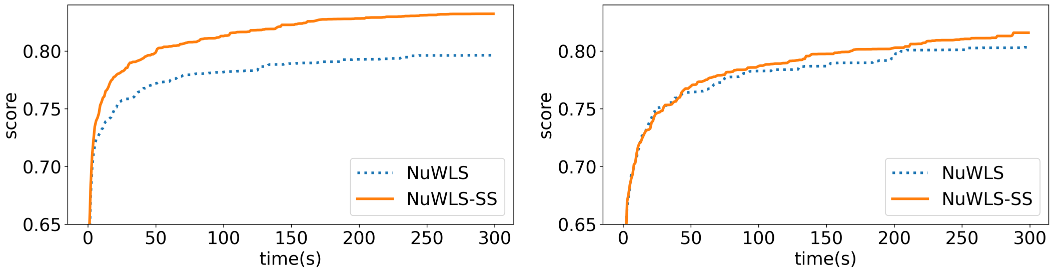


Figure 3 Comparison of the score in different cutoff time, by NuWLS. The left and right plots show MSE19 and MSE 23 respectively. The pattern of MSE20, MSE21 and MSE22 are so similar to MSE19, so we omit these plots here to be concise.

Table 1 Experiment results of SatLike3.0.

benchmark	#inst.	#win.	#lose.	$satlike_{ss}$	$satlike$	$accept(\%)$	$dist$	$step(\%)$
MSE19	299	**72**	13	**0.7108**	0.6911	39	7.6	79
MSE20	262	**59**	13	**0.6923**	0.6845	41	8.4	80
MSE21	155	**30**	10	**0.6155**	0.6065	40	5.0	77
MSE22	179	**39**	9	**0.6637**	0.6459	42	4.9	82
MSE23	179	**61**	16	0.6193	**0.6257**	33	6.0	80

Table 2 Experiment results of NuWLS.

benchmark	#inst.	#win.	#lose.	$nuwls_{ss}$	$nuwls$	$accept(\%)$	$dist$	$step(\%)$
MSE19	299	**74**	38	**0.8322**	0.7963	28	4.5	85
MSE20	262	**75**	30	**0.8219**	0.7882	30	5.0	86
MSE21	155	**34**	19	**0.7991**	0.7723	27	4.7	84
MSE22	179	**46**	25	**0.8044**	0.7856	28	4.0	86
MSE23	179	**61**	30	**0.8158**	0.8032	25	5.3	86

4.3 Experiment Results on PBO benchmarks

We have also applied our methods in PBO solvers LS-PBO and NuPBO, and summarized the results in Table 3 and 4. The conclusion is consistent with that in the previous subsection: in CRAFTED, MIPLIB, and PB16 benchmarks, both the number of winning cases and the score are significantly improved. The only deterioration falls in the score of Industry problem set by NuPBO, which can be explained by the *step*(%) metric in Table 4. Given that the Industry set comprises larger-scale problems compared to other benchmarks, the original solver usually cannot converge in 300 seconds, in other words the solution quality highly depends on the number of flips. However, our new method achieves only performed 69% valid flips compared to the original solver. As a result, despite unit propagation demonstrating effectiveness by achieving more winning cases, the average score drops slightly.

4.4 Sensitivity analysis

Our algorithm shows stability across a wide range of parameter values. In each of the following experiments, we vary one parameter while keeping the rest as specified in Section 4.1. For each solver, we conduct 27 experiments (1 default, 6 for $k_{\max}$, 4 for $k_{\min}$, 3 for $final\_steps$, 3 for n_2, and 10 for random seeds) on all benchmarks in our study. We compute the scores in comparison with the best value among these 27 outputs. For conciseness, we present the average score of all benchmarks for each experiment in Table 5.

The first part of Table 5 shows the average scores and standard errors of experiments with 10 different random seeds. The statistics demonstrate that the algorithm is robust against randomness.

$k_{\max}$ and $k_{\min}$ control the frequency of UP calls. The results of varying these parameters are shown in the second and third parts of Table 5. $k_{\min}$ corresponds to the highest frequency of UP calls. As discussed in Section 3.2, a smaller $k_{\min}$ results in a more significant UP effect but consumes more time. Conversely, a larger $k_{\min}$ reduces the frequency of UP calls, making performance closer to the original solver. Setting $k_{\min} = \infty$ means no UP is conducted, making the algorithm identical to the original.

$k_{\max}$ corresponds to the lowest frequency of UP calls. This parameter is set to reduce time waste when UP is ineffective during the search. We tested $k_{\max}$ from 10 ($= k_{\min}$) to 10240 (extremely low frequency). The performance is not sensitive to this parameter.

The results of varying the number of UP candidates n_2 are shown in the fourth part of Table 5. A larger n_2 increases the chances of finding a better UP but also increases the time cost. We chose a balanced value of 5 for our paper.

The results of varying $final\_steps$, the extra steps before restart (described in Section 3.2), are shown in the last part of Table 5. Setting $final\_steps = 0$ disables revisiting the best-found solution, resulting in less intensified search and relatively lower scores. There is not much difference when using other values.

4.5 Validation of Acceptance Criteria

In this part, we tested a variant that accepts all local optima after unit propagation. The results of the SATLike3.0-based experiment are shown in Table 6, while the other solvers support the same conclusion. Across all benchmarks, the acceptance rate is increased to around 80% because only conflicting flips or candidates with an empty *UPList* were rejected. As a result, approximately 10% more valid steps could be attempted before the cutoff time. However, the number of deteriorated instances is significantly outweighed the number of improved ones, and the score drops across four benchmarks at the same time. These results demonstrate the necessity of our acceptance criteria. Moreover, it is worth noting that the average propagated distance *dist* decreased compared to Table 2. As mentioned in the previous text, solutions often cluster closely together. If the algorithm has no enforcement to accept a better local optimum, it is allowed to visit these solutions in loops, which means it is stuck.

5 Related Works

There have been many applications in combining Local Search and CDCL solvers. In this section, we classify these works into two categories based on whether Local Search is the master solver, and discuss some noticeable works from each category.

The first category use CDCL as the main solver, with local search methods often viewed as a black box. In SAT problems, the solve ends if a feasible solution is found, prompting some hybrid SAT solvers to use LS on branch nodes to accelerate solving. [30] calls LS at every node in the CDCL search tree, and the solve succeeds if LS finds a solution at any nodes. [9] branches until the length of the partial assignment exceeds a threshold, then it is extended into a complete assignment and passes as the initial assignment to a LS solver. Some other works also use LS to estimate the priority of branch variables ([13]), or to obtain a upper bound of the optimization model ([28]).

The second category considers LS as the main body. [29] shows learnt clauses by CDCL will be beneficial to LS solving. [21] uses implication graph to discover variable equivalency, so the redundant variables will be substituted. The most related work is SatHys proposed in [3], where the CDCL solver maintains a partial assignment $\mathcal{I}_p$ while LS conducts a search with complete assignment $\mathcal{I}_c$. If LS is stuck, a literal l is added to $\mathcal{I}_c$ and propagated. If conflict occurs, CDCL learns a clause and backtracks. Finally $\mathcal{I}_c$ will be overwritten by $\mathcal{I}_p$. Our approach differs in two aspects. The major difference is that our unit propagation is based on one literal instead of on $\mathcal{I}_p$, which is a better adaption to optimization problems. We notice that in SatHys, if an incorrect decision is made and added in $\mathcal{I}_p$, it cannot be cancelled unless it is backtracked or restarted. It works in SAT problems because the

Table 3 Experiment results of LS-PBO.

benchmark	#inst.	#win.	#lose.	$lspbo_{ss}$	$lspbo$	$accept$(%)	$dist$	$step$(%)
CRAFTED	955	**38**	6	**0.9093**	0.9081	46	2.4	78
MIPLIB	291	**76**	36	**0.7260**	0.7042	32	5.4	88
PB16	1600	**346**	92	**0.7653**	0.7147	30	8.6	80
Industry	63	**45**	9	**0.9835**	0.9295	47	3.4	80

Table 4 Experiment results of NuPBO.

benchmark	#inst.	#win.	#lose.	$nupbo_{ss}$	$nupbo$	$accept$(%)	$dist$	$step$(%)
CRAFTED	955	**46**	6	**0.9171**	0.9168	38	2.8	79
MIPLIB	291	**95**	34	**0.8219**	0.7932	27	8.9	82
PB16	1600	**269**	88	**0.8293**	0.8283	24	20.3	81
Industry	63	**32**	20	0.9456	**0.9561**	47	3.7	69

Table 5 Performance of our hybrid algorithm under different parameter settings.

(seeds)	avg.score	std.err
LS-PBO	0.8011	0.0015
NuPBO	0.8379	0.0007
SATLike	0.6503	0.0044
NuWLS	0.7835	0.0030

$k_{\min}$ =	5	10	20	40	80	∞
LS-PBO	0.8145	0.8010	0.7984	0.7858	0.7823	0.7762
NuPBO	0.8356	0.8386	0.8381	0.8376	0.8382	0.8360
SATLike	0.6527	0.6493	0.6491	0.6491	0.6436	0.6405
NuWLS	0.7914	0.7867	0.7784	0.7763	0.7733	0.7626

$k_{\max}$ =	10	160	640	1280	2560	5120	10240
LS-PBO	0.8055	0.7997	0.7988	0.8049	0.8010	0.8017	0.8016
NuPBO	0.8326	0.8368	0.8353	0.8368	0.8386	0.8380	0.8370
SATLike	0.6459	0.6471	0.6493	0.6506	0.6496	0.6470	0.6471
NuWLS	0.7869	0.7823	0.7829	0.7861	0.7867	0.7902	0.7891

$final\_steps$ =	0	1000	10000	100000
LS-PBO	0.7978	0.8021	0.8010	0.8029
NuPBO	0.8358	0.8370	0.8386	0.8380
Satlike	0.6449	0.6473	0.6493	0.6470
NuWLS	0.7891	0.7922	0.7867	0.7899

n_2 =	1	5	10	15
LS-PBO	0.7770	0.8010	0.8131	0.8155
NuPBO	0.8373	0.8386	0.8374	0.8402
Satlike	0.6490	0.6493	0.6490	0.6560
NuWLS	0.7733	0.7867	0.7867	0.7890

Table 6 Experiment results of SATLike3.0, comparing normal version $satlike_{ss}$ with an all-accept variant $satlike_{any}$. The last three columns of statistics are from $satlike_{any}$.

benchmark	#inst.	#win.	#lose.	$satlike_{ss}$	$satlike_{any}$	$accept$(%)	$dist$	$step$(%)
MSE19	299	**65**	17	**0.7132**	0.6962	83	6.0	106
MSE20	262	**54**	17	**0.6920**	0.6831	85	5.4	112
MSE21	155	**28**	11	**0.6041**	0.5761	86	3.2	113
MSE22	179	**43**	13	**0.6508**	0.6430	85	3.6	112
MSE23	179	**48**	22	0.6193	**0.6210**	79	3.5	104

cancelling situation (i.e. conflict occurs) is consistent with the target to find a feasible assignment. But in optimization problems, the target is to optimize an objective, and poor decisions (with respect to objective) may be kept in $\mathcal{I}_p$ and cause LS ending up in feasible but bad solutions. Also, the adaption of our work allows the tricks described in section 3.4 to promise a light and efficient way of unit propagation. Another novelty of our work lies in the backtrack mechanism, it gives LS the highest authority to judge propagated assignments with LS *Punishment*, and reject unfavorable CDCL moves.

6 Conclusion

We propose a novel search framework which embeds unit propagation into local search algorithms to help escaping local optima. Then, we introduce three innovative ideas to enhance this new search framework. The first one is a mechanism to determine when to invoke UP to change the current assignment obtained by the local search method, which helps the algorithm jump out of local optima. The second one is the *FlipUP* algorithm to perform unit propagation and heuristic search. Finally, we propose an acceptance mechanism to decide whether to accept the propagated solution. Experiments conducted on benchmarks from MaxSAT Evaluations, PBO competitions and realistic instances demonstrate that our method can bring significant improvement in terms of the number of winning instances and average scores across four state-of-the-art MaxSAT and PBO local search solvers. Consequently, we believe that the cooperation of local search and unit propagation techniques represents a promising research field which has great potential for exploration.

References

1 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. SAT-based MaxSAT algorithms. *Artif. Intell.*, 196:77–105, 2013.

2 Carlos Ansótegui and Joel Gabàs. WPM3: an (in)complete algorithm for weighted partial maxsat. *Artif. Intell.*, 250:37–57, 2017.

3 Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. Boosting local search thanks to cdcl. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2010.

4 Adrian Balint, Michael Henn, and Oliver Gableske. A novel approach to combine a sls-and a dpll-solver for the satisfiability problem. In *Theory and Applications of Satisfiability Testing-SAT 2009: 12th International Conference, SAT 2009, Swansea, UK, June 30-July 3, 2009. Proceedings 12*, pages 284–297. Springer, 2009.

5 Jeremias Berg, Emir Demirovic, and Peter J. Stuckey. Core-boosted linear search for incomplete maxsat. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Proceedings CPAIOR 2019,*, volume 11494, pages 39–56, 2019.

6 Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–64, 2010.

7 Shaowei Cai and Zhendong Lei. Old techniques in new ways: Clause weighting, unit propagation and hybridization for maximum satisfiability. *Artif. Intell.*, 287:103354, 2020.

8 Shaowei Cai, Chuan Luo, John Thornton, and Kaile Su. Tailoring local search for partial MaxSAT. In *Proceedings of AAAI 2014*, pages 2623–2629, 2014.

9 Shaowei Cai and Xindi Zhang. Deep cooperation of CDCL and local search for SAT (extended abstract). In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 5274–5278. ijcai.org, 2022.

10 Donald Chai and Andreas Kuehlmann. A fast pseudo-Boolean constraint solver. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(3):305–317, 2005.

11 Yi Chu, Shaowei Cai, and Chuan Luo. Nuwls: Improving local search for (weighted) partial maxsat by new weighting techniques. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 3915–3923. AAAI Press, 2023.

12 Yi Chu, Shaowei Cai, Chuan Luo, Zhendong Lei, and Cong Peng. Towards more efficient local search for pseudo-boolean optimization. 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 12:1–12:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

13 James M Crawford. Solving satisfiability problems using a combination of systematic and local search. In *Second DIMACS Challenge: cliques, coloring, and satisfiability.* Citeseer, 1993.

14 Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Proceedings of CP 2011*, pages 225–239, 2011.

15 Emir Demirovic and Peter J. Stuckey. Techniques inspired by local search for incomplete maxsat and the linear algorithm: Varying resolution and solution-guided search. In Thomas Schiex and Simon de Givry, editors, *Proceedings of CP 2019*, volume 11802, pages 177–194, 2019.

16 Jo Devriendt, Stephan Gocht, Emir Demirović, Peter Stuckey, and Jakob Nordström. Cutting to the core of pseudo-Boolean optimization: Combining core-guided search with cutting planes reasoning. In *AAAI 2021,Accepted*, 2021. URL: `http://www.csc.kth.se/~jakobn/research/CuttingToTheCore_AAAI.pdf`.

17 Jan Elffers, Jesús Giráldez-Cru, Jakob Nordström, and Marc Vinyals. Using combinatorial benchmarks to probe the reasoning power of pseudo-Boolean solvers. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Proceedings of SAT 2018*, pages 75–93, 2018.

18 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-Boolean solving. In Jérôme Lang, editor, *Proceedings of IJCAI 2018*, pages 1291–1299, 2018.

19 Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006.

20 Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Proceedings of SAT 2006*, pages 252–265, 2006.

21 Djamal Habet, Chu Min Li, Laure Devendeville, and Michel Vasquez. A hybrid approach for sat. In *International Conference on Principles and Practice of Constraint Programming*, pages 172–184. Springer, 2002.

22 William S Havens and Bistra N Dilkina. A hybrid schema for systematic local search. In *Advances in Artificial Intelligence: 17th Conference of the Canadian Society for Computational Studies of Intelligence, Canadian AI 2004, London, Ontario, Canada, May 17-19, 2004. Proceedings 17*, pages 248–260. Springer, 2004.

23 Saurabh Joshi, Prateek Kumar, Sukrut Rao, and Ruben Martins. Open-wbo-inc: Approximation strategies for incomplete weighted maxsat. *J. Satisf. Boolean Model. Comput.*, 11(1):73–97, 2019.

24 Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, 2002.

25 Tuukka Korhonen, Jeremias Berg, Paul Saikko, and Matti Järvisalo. Maxpre: an extended maxsat preprocessor. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 449–456. Springer, 2017.

26 Zhendong Lei and Shaowei Cai. Solving (weighted) partial maxsat by dynamic local search for SAT. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, pages 1346–1352, 2018.

27 Zhendong Lei, Shaowei Cai, Chuan Luo, and Holger H. Hoos. Efficient local search for pseudo boolean optimization. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 332–348. Springer, 2021.

28 Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Combining clause learning and branch and bound for maxsat. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 38:1–38:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

29 Jan-Hendrik Lorenz and Florian Wörz. On the effect of learned clauses on stochastic local search. In *Theory and Applications of Satisfiability Testing–SAT 2020: 23rd International Conference, Alghero, Italy, July 3–10, 2020, Proceedings 23*, pages 89–106. Springer, 2020.

30 Bertrand Mazure, Lakhdar Sais, and Éric Grégoire. Boosting complete techniques thanks to local search methods. *Annals of mathematics and artificial intelligence*, 22:319–331, 1998.

31 Alexander Nadel. Anytime weighted maxsat with improved polarity selection and bit-vector optimization. In Clark W. Barrett and Jin Yang, editors, *Proceedings of FMCAD 2019*, pages 193–202, 2019.

32 Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided MaxSAT resolution. In *Proceedings of AAAI 2014*, pages 2717–2723, 2014.

33 Steven Prestwich. Randomised backtracking for linear pseudo-Boolean constraint problems. In *Proceedings of CPAIOR 2002*, pages 7–20, 2002.

34 João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiabilitysatlike. In Rob A. Rutenbar and Ralph H. J. M. Otten, editors, *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, pages 220–227. IEEE Computer Society / ACM, 1996.

35 Robert Wille, Hongyan Zhang, and Rolf Drechsler. ATPG for reversible circuits using simulation, boolean satisfiability, and pseudo boolean optimization. In *IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2011, 4-6 July 2011, Chennai, India*, pages 120–125. IEEE Computer Society, 2011.

36 Aolong Zha, Miyuki Koshimura, and Hiroshi Fujita. A hybrid encoding of pseudo-Boolean constraints into CNF. In *Proceedings of TAAI 2017*, pages 9–12. IEEE Computer Society, 2017.

37 Yuhang Zhang, Richard I. Hartley, John Mashford, and Stewart Burn. Superpixels via pseudo-boolean optimization. In Dimitris N. Metaxas, Long Quan, Alberto Sanfeliu, and Luc Van Gool, editors, *IEEE International Conference on Computer Vision, ICCV 2011, Barcelona, Spain, November 6-13, 2011*, pages 1387–1394. IEEE Computer Society, 2011.

38 Jiongzhi Zheng, Kun He, Jianrong Zhou, Yan Jin, Chu-Min Li, and Felip Manyà. Bandmaxsat: A local search maxsat solver with multi-armed bandit. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 1901–1907. ijcai.org, 2022.

Cumulative Scheduling with Calendars and Overtime

Samuel Cloutier
Université Laval, Québec, Canada

Claude-Guy Quimper
Université Laval, Québec, Canada

Abstract

In project scheduling, calendar considerations can increase the duration of a task when its execution overlaps with holidays. On the other hand, the use of overtime may decrease the task's duration. We introduce the CalendarOvertime constraint which verifies that a task follows a calendar with overtime and holidays. We also introduce the CumulativeOvertime constraint, a variant of the Cumulative constraint, that also reasons with the calendars when propagating according to the resource consumption, the overtime, and the holidays. Experimental results of a RCPSP model on the PSPLIB, BL, and PACK instances augmented with calendars and overtime show that the use of the CalendarOvertime constraint offers a speedup greater than 2.9 on the instances optimally solved and finds better solutions on more than 79% of the remaining instances when compared to a decomposition of the constraint. We also show that the use of our CumulativeOvertime constraint further improves these results.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling; Theory of computation → Constraint and logic programming

Keywords and phrases Constraint programming, Scheduling, Global constraints, Calendars, Overtime, Cumulative constraint, Time-Tabling

Digital Object Identifier 10.4230/LIPIcs.CP.2024.7

Supplementary Material *Software (Source Code and Instances)*: `https://github.com/Samclou/chuffed/releases/tag/Calendars-cp2024`

1 Introduction

In project management, it is common to schedule a variety of tasks on a project timeline. With multiple machines and workers, some tasks can easily be done in parallel. For example, a furniture factory can build a table at the same time as a chair, as long as sufficient workers and workspace are available. Cumulative scheduling allows the simultaneous execution of tasks while limiting these executions in order not to overload the resources.

Scheduling problems (with release times and deadlines) are generally NP-hard [9]. Constraint programming is frequently used to solve these problems.

In practice, tasks can be suspended for some time periods. The factory can be closed at night and during weekends. If every operation is stopped at these times, these time periods can simply be ignored. If some tasks must be stopped at specific times while others do not because, for example, the machines keep working at night, side constraints become necessary to encode these suspensions and this may undermine the efficiency of the models.

The aim of this research is to design constraints that facilitate the modeling and solving of scheduling problems where tasks must be interrupted according to a calendar, or may be shortened by working overtime.

Section 2 provides background on the cumulative scheduling problem, the Time-Tabling rule, and the generalizations with calendars. Section 3 presents the new constraints we introduce. Section 4 describes how these constraints can be decomposed into elementary constraints while Section 5 details the propagators of the new constraints. Section 6 describes the methodology we used to test our new propagators. Section 7 evaluates the performances of the new propagators and the decomposition.

2 Background

2.1 Cumulative Scheduling

The cumulative scheduling problem is often modeled with the CUMULATIVE global constraint [1]. In what follows, lower-case symbols represent constants and indices while upper-case ones represent variables. Symbols in bold represent arrays that we define using list comprehension. Let $\mathcal{I}$ be the set of tasks and let p_i, h_i, and S_i for $i \in \mathcal{I}$ be the task's processing time, usage of a resource, and starting time. Let $\mathbf{S}$ be $[S_i \mid i \in \mathcal{I}]$, $\mathbf{p}$ be $[p_i \mid i \in \mathcal{I}]$, and $\mathbf{h}$ be $[h_i \mid i \in \mathcal{I}]$. The constraint CUMULATIVE$(\mathbf{S}, \mathbf{p}, \mathbf{h}, h^{\max})$ asserts that for a resource of capacity $h^{\max}$ that executes the tasks in $\mathcal{I}$, for any integer time point t in the *horizon* (the complete time interval considered, $[0, t^{\max}]$), $\sum_{i \in \mathcal{I}: t \in [S_i, S_i+p_i)} h_i \leq h^{\max}$. This means that tasks running simultaneously cannot, at any time, consume more than the resource's capacity. The execution window of a task is considered to be $[S_i, S_i + p_i)$, with $S_i + p_i$ being its ending time. As such, S_i takes integer values in $[0, t^{\max} - p_i]$. The CUMULATIVE constraint uses filtering algorithms to prune the variable domains during the search. Since enforcing bounds consistency is NP-hard [17], one usually applies simple filtering rules that offer a weaker level of consistency such as the Time-Tabling rule [2]. In a multi-resource problem, each resource is associated with its own CUMULATIVE constraint. A common objective is to minimize the *makespan*, i.e., the completion time of the last task.

Lazy clause generation [18] is a technique that deduces new logical constraints, in the form of a disjunction of literals representing domain states of the variables, from the failures encountered during the search. It permits learning previous bad decisions and prune them from the remaining search tree. Solvers that implement lazy clause generation, such as Chuffed [8], have been shown to perform well on cumulative scheduling problems [20].

2.2 The Time-Tabling Rule

Let $\underline{X}$ and $\overline{X}$ respectively be the smallest and largest values a variable X can take. We note $\mathrm{dom}(X)$ the set of all values variable X may take, i.e., its domain. The Time-Tabling rule [2] filters the domains of the starting time variables subject to the CUMULATIVE constraint. We note the earliest starting time of task i as est_i, its latest starting time as lst_i, its earliest completion time as ect_i, and its latest completion time as lct_i. These are defined as follows:

$$\mathrm{est}_i \overset{def}{=} \underline{S_i} \tag{1}$$

$$\mathrm{lst}_i \overset{def}{=} \overline{S_i} \tag{2}$$

$$\mathrm{ect}_i \overset{def}{=} \underline{S_i} + p_i \tag{3}$$

$$\mathrm{lct}_i \overset{def}{=} \overline{S_i} + p_i \tag{4}$$

If a task duration is a variable, these definitions use the lower bound of that variable rather than p_i. These four concepts bound the time points at which a task can be in execution. A task must be in execution in the interval $[\mathrm{lst}_i, \mathrm{ect}_i)$, called the *compulsory part*, if it is non-empty.

The Time-tabling rule computes the compulsory part of each task and aggregates them to create a *consumption profile*, i.e., a lower bound of the resource consumption at each time point. The Time-Tabling check identifies a conflict when a point in this profile overloads the resource. The Time-Tabling filtering algorithm makes sure that if a task overloads the resource when executing at time t, then the task must either start after or finish before t [19]. Let $f(\Omega, t)$ be the consumption profile of a resource at time t given the tasks in the set Ω.

$$f(\Omega, t) = \sum_{\{i \in \Omega \mid t \in [\mathrm{lst}_i, \mathrm{ect}_i)\}} h_i \tag{5}$$

The checking and filtering rules for the cumulative constraint can then be expressed as:

$$\exists\, t, f(\mathcal{I}, t) > h^{\max} \implies \text{conflict} \tag{6}$$

$$\mathrm{ect}_i > t \wedge h^{\max} < h_i + f(\mathcal{I} \setminus \{i\}, t) \implies \underline{S_i'} > t \tag{7}$$

Rule (7) can be adapted to filter $\overline{S_i}$. Propagators applying the Time-Tabling rule can have a complexity as low as $\mathcal{O}(n)$, n being the number of tasks. However, there exist efficient implementations with a complexity of $\mathcal{O}(n^2)$ [10].

2.3 Augmentation With Calendars

It is possible that, at some specific times, some tasks must be paused while others remain unaffected. We say that these special times are defined by a *calendar*. This notion is close to preemption, but it is still in a non-preemptive context. A task can only be suspended because of calendars and it must resume as soon as each calendar affecting the task permit it. There are multiple ways to conceptualize calendars and many ways to solve the problem have been studied.

2.3.1 Calendars Associated to Resources

One way to add calendars into the cumulative scheduling problem is to assign to each resource an arbitrary array of Booleans indicating whether the resource is available or not at a specific time. When a resource is unavailable, tasks cannot progress in their execution, which has the effect of artificially lengthening their execution time.

Kreter et al. [12, 13, 14] use *releasable resources* that stop being consumed by tasks that are paused. Their tasks may have an initial uninterruptible setup time. To deal with their complex problem, Kreter et al. study various methods:

- They use multiple binary linear model formulations and search methods that they compare against each other [12].
- They implement a new constraint, namely CumulativeCalendar, in a constraint solver and compare its efficiency with various models using existing constraints [13].
- They compare both previous methods on the resource investment problem, i.e., the problem of minimizing the cost associated to the maximum consumption of each resources [14].

Kreter et al. [13, 14] show that the use of CumulativeCalendar constraints with a lazy clause generation solver such as Chuffed is highly competitive to solve their problems.

2.3.2 Calendars Associated to Tasks

Boudreault et al. [6] directly assigns the calendars to the tasks, meaning that each task follows its own calendar, rather than following one implied by those of the resources. This might be wanted over the preceding option when some resources are plentiful enough that modeling them with a Cumulative constraint would be useless. However, if these omitted resources have a calendar, they still need to affect the actual calendar of a task, which justifies using this more general type of calendar. For Boudreault et al., the calendars are not arbitrary as the composition of a working day is fixed and shared between all calendars: the regular execution time starts at a given time in the morning and finishes at a given time in the afternoon. Some tasks can execute during the weekend while other tasks cannot. The calendars are periodic on weeks and do not allow exceptions even for holidays.

Boudreault et al. [6] allow overtime, i.e., a way to shorten the execution time of a task while inducing an overtime cost. Working one time point worth of overtime on task i costs w_i. The amount of overtime is limited by its availability. Indeed, overtime is assigned to specific time points in the calendar, during which a task can be interrupted or can continue its execution if it is executed in overtime.

To solve their problem, Boudreault et al. [6] do not implement a new constraint in a solver, they rather decompose the calendar constraints into elementary constraints available in any constraint solver and use a meta-heuristic to reach better results.

2.3.3 Other Approaches

In CP Optimizer, tasks in scheduling problems are modeled through *interval* variables. These variables possess a starting time and an ending time, but also a *size*, and a step function, called *intensity*. The size of an interval variable can be interpreted as the work contained in the interval, while the intensity gives the ratio of work that each time point provides. As such, the behavior that tasks do not progress during holidays can directly be treated through the intensity function by having an intensity of 0% during that time. If tasks are not allowed to start or end during holidays, constraints forbidStart and forbidEnd directly model and deal efficiently with this aspect [15].

Beldiceanu [4, 5] introduces a Calendar constraint to model this behavior. This constraint maps, for each calendar, the real-time coordinate system to a virtual one where there are no interruptions. These virtual time coordinates then permit the use of classic propagators that normally cannot deal with calendars. The mapping deals with the problem of changing the length of tasks and of making sure none starts nor ends during a holiday.

3 Calendar Constraints With Overtime

This section presents the new constraints we introduce. The next section presents how they can be decomposed while the following section describes their filtering algorithms.

The constraints we introduce are motivated by the calendar constraints used by Boudreault et al. [6]. We generalize the calendars they use by allowing arbitrary calendars, i.e., non-periodic calendars with sporadic holidays. Every task must follow a specific calendar. The *elapsed time* of a task is the difference between its end time and its start time. Without calendars, the elapsed time of a task is simply its processing time.

Given a horizon representing all the time points at which a task can be processed, our calendars are arbitrary sequences of the symbols $\mathbb{r}$, $\mathbb{c}$, or $\mathbb{o}$ where the t-th symbol represents the nature of the t-th time point, i.e., the t-th hour in our context. The symbol $\mathbb{r}$ indicates that the time point is regular (the classic scheduling problem would correspond to a calendar with only $\mathbb{r}$'s). The symbol $\mathbb{c}$ indicates that the time point is closed, that is, tasks are suspended when they are in process at that time. As for $\mathbb{o}$, it indicates that the time point is an overtime period that can behave as a regular or a closed time point whether it is worked or not. The duration that a task is worked in overtime is the number of time points of type

$\mathbb{o}$ that behave as type $\mathbb{r}$. The time point of the start and the one preceding the end of a task must not be closed. If either of these points is an overtime period, the amount of worked overtime must allow to work them. The time worked in the execution window of a task must be exactly its processing time.

▶ **Example 1.** Let i be a task with processing time $p_i = 3$ following the calendar $\mathbb{coroorrc}$. This task cannot start at time 0 since it is closed. It can execute at times 1, 2, and 3. It could also execute at times 1, 2, and 4. In that case, it finishes later and is idle at time 3. The task cannot start at time 1 while ending at time 7 because the regular times 2, 5, and 6 are mandatory, leaving no work to perform at time 1.

3.1 The CalendarOvertime Constraint

We define a new constraint to model calendars with overtime. The CalendarOvertime constraint, for a starting time variable S, an elapsed time variable E (with $\text{dom}(E) \subseteq [p, t^{\max}]$), an overtime variable O (with $\text{dom}(O) \subseteq [0, p]$), a processing time p, and a calendar Cal, asserts that:

- The first and last time points of the execution window $[S, S+E)$ are not closed.
- There are enough, but not too many, worked time points in the window to complete the task of processing time p with the overtime prescribed by O.
- There are enough overtime periods in $[S, S+E)$ for the overtime prescribed by O.
- The first and last time points in $[S, S+E)$ can actually be worked if they are of type $\mathbb{o}$.

In a more mathematical way, the CalendarOvertime constraint is defined as follows:

$$\begin{aligned}
&\textsc{CalendarOvertime}(S, E, O, p, \text{Cal}) \overset{def}{\iff} \text{Cal}[S] \neq \mathbb{c} \wedge \text{Cal}[S+E-1] \neq \mathbb{c} \\
&\qquad \wedge O = p - |\{t \in [S, S+E) \mid \text{Cal}[t] = \mathbb{r}\}| \\
&\qquad \wedge O \leq |\{t \in [S, S+E) \mid \text{Cal}[t] = \mathbb{o}\}| \\
&\qquad \wedge |\{t \in \{S, S+E-1\} \mid \text{Cal}[t] = \mathbb{o}\}| \leq O
\end{aligned}$$

This constraint does not deal with the concept of resource consumption. It simply maintains consistency between the variables S, E, and O given a processing time p and a calendar Cal.

3.2 The CumulativeOvertime Constraint

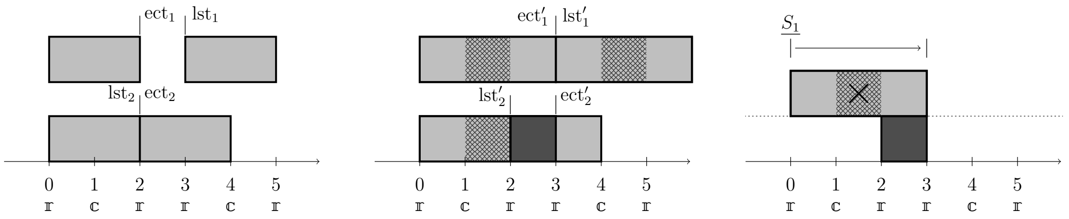
Let $\mathbf{E}$ be $[E_i \mid i \in \mathcal{I}]$, $\mathbf{O}$ be $[O_i \mid i \in \mathcal{I}]$, and $\mathbf{Cal}$ be $[\text{Cal}_i \mid i \in \mathcal{I}]$. We define the new CumulativeOvertime constraint as follows:

$$\begin{aligned}
&\textsc{CumulativeOvertime}(\mathbf{S}, \mathbf{E}, \mathbf{O}, \mathbf{p}, \mathbf{Cal}, \mathbf{h}, h^{\max}) \\
&\overset{def}{\iff} \textsc{Cumulative}(\mathbf{S}, \mathbf{E}, \mathbf{h}, h^{\max}) \\
&\qquad \wedge \bigwedge_{i \in \mathcal{I}} \textsc{CalendarOvertime}(S_i, E_i, O_i, p_i, \text{Cal}_i)
\end{aligned}$$

In words, we define the new CumulativeOvertime constraint as a conjunction of a Cumulative constraint and the CalendarOvertime constraints associated to the tasks consuming the resource. Importantly, we consider that tasks continue to use the resource while they are suspended, be it because of unworked overtime or closed time. This last aspect is reasonable if we consider that some resources may relate to small spaces where moving machinery should be avoided, such as during ship refitting. In that case, releasing the resource necessitates unwanted work that would overcomplicate the planning.

Although the propagator for CalendarOvertime (described at Section 5.1) maintains bounds consistency on S, E, and O given processing time p and calendar Cal, the bounds found on E are often not sufficient to allow the Cumulative constraint to perform a good propagation. This is because the Time-Tabling rule filtering the Cumulative constraint only uses $\underline{E}$ in its reasoning and does not take into account the calendars.

▶ **Example 2.** Consider a task 1 of processing time $p_1 = 2$ following the calendar $\mathtt{rcrrcr}$ with $\mathrm{dom}(S_1) = [0, 3]$ and $\mathrm{dom}(E_1) = [2, 3]$. Let task 2 follow the same calendar, with $p_2 = 2$, $\mathrm{dom}(S_2) = [0, 2]$ and $\mathrm{dom}(E_2) = [2, 3]$. In this case, $O_1 = O_2 = 0$. It can be seen that the domains of the variables of both tasks are bounds consistent with respect to the CalendarOvertime constraints and that they do not induce any compulsory part. Let both tasks consume 1 unit of a resource of capacity 1 (i.e., a disjunctive resource). Then, the Time-Tabling rule is not able to deduce that S_1 should be fixed to 3. Indeed, not only does it fail to detect that task 2 must be executing at time 2 (meaning its compulsory part in Figure 1a should not be empty), it also considers that task 1 could start at time 0 and end at time 2 (excluded), leading to no propagation. Should the calendar considerations be included in the rule, the propagation would be deduced, as visible in Figures 1b and 1c.



(a) When not considering the calendar, the time windows that end the earliest or start the latest have no intersection. The profile is empty and no propagation is possible.

(b) When taking the calendar into account, the tasks execution windows become longer, allowing the detection of compulsory parts that would otherwise be missed.

(c) With this augmented profile, calendar considerations can again lengthen a task's execution window and permit the detection of propagations that would be missed otherwise.

Figure 1 Comparison of the propagation done by the Time-tabling rule without and with calendar considerations in the case described by Example 2. The meaning of ect'_i and lst'_i is defined at section 5.1.

4 Decomposition of the New Constraints

To evaluate the usefulness of the new constraints defined in the previous section, we need to compare them with their decomposition into elementary constraints.

4.1 Decomposition of the CalendarOvertime Constraint

Suppose we want to decompose the constraint CalendarOvertime$(S, E, O, p, \mathrm{Cal})$. If Cal is the trivial calendar, i.e., the calendar with only regular time points without closed time or overtime, the constraint is trivially decomposed as follows:

$$E = p \land O = 0 \tag{8}$$

If Cal is not the trivial calendar, more work is necessary. Let the *compiled calendars* $C^{\mathbb{c}}$, $C^{\mathbb{r}}$, and $C^{\mathbb{o}}$ respectively count how many closed, regular, and overtime periods are encountered in calendar Cal before a given time point in the horizon $[0, t^{\max}]$. The number of closed time points in the time interval $[a, b)$ is simply given by $C^{\mathbb{c}}[b] - C^{\mathbb{c}}[a]$. These arrays can be precomputed.

$$C^{\mathbb{x}}[t] = |\{j \in [0, t) \mid \mathrm{Cal}[j] = \mathbb{x}\}| \qquad \forall \mathbb{x} \in \{\mathbb{c}, \mathbb{r}, \mathbb{o}\} \tag{9}$$

The following variables are added to the decomposition: I, for the idle time i.e., the number of time points in $[S, S+E)$ that are not worked, as well as $\mathrm{N}^{\mathbb{c}}$, $\mathrm{N}^{\mathbb{r}}$, and $\mathrm{N}^{\mathbb{o}}$ respectively for the number of closed, regular, and overtime time points within the execution window $[S, S+E)$.

First, variables $\mathrm{N}^{\mathbb{c}}$, $\mathrm{N}^{\mathbb{r}}$, and $\mathrm{N}^{\mathbb{o}}$ must count the time points of each type in the execution window.

$$\mathrm{N}^{\mathbb{x}} = C^{\mathbb{x}}[S+E] - C^{\mathbb{x}}[S] \qquad \forall \mathbb{x} \in \{\mathbb{c}, \mathbb{r}, \mathbb{o}\} \tag{10}$$

The number of regular time points in the execution window must be equal to the regular time worked. There must be enough overtime periods in the execution window to work the overtime prescribed by O. The idle time is not only the closed time points, but also the unworked overtime periods. Since overtime periods appear directly in the calendar, the elapsed time is simply the processing time plus the idle time. The overtime is nonnegative and at most equal to the processing time.

$$\mathrm{N}^{\mathbb{r}} = p - O \tag{11}$$

$$\mathrm{N}^{\mathbb{o}} \geq O \tag{12}$$

$$I = \mathrm{N}^{\mathbb{c}} + \mathrm{N}^{\mathbb{o}} - O \tag{13}$$

$$E = p + I \tag{14}$$

$$0 \leq O \leq p \tag{15}$$

The starting time and the time preceding the ending time of a task must be able to be worked, even if they are overtime periods. Let $\mathbb{1}(x)$ be the function that returns 1 if x is true and 0 otherwise.

$$\mathrm{Cal}[S] \neq \mathbb{c} \tag{16}$$

$$\mathrm{Cal}[S+E-1] \neq \mathbb{c} \tag{17}$$

$$E > 1 \implies O \geq \mathbb{1}(\mathrm{Cal}[S] = \mathbb{o}) + \mathbb{1}(\mathrm{Cal}[S+E-1] = \mathbb{o}) \tag{18}$$

$$E = 1 \implies O \geq \mathbb{1}(\mathrm{Cal}[S] = \mathbb{o}) \tag{19}$$

4.2 Decomposition of the CumulativeOvertime Constraint

A decomposition of the constraint CUMULATIVEOVERTIME$(\mathbf{S}, \mathbf{E}, \mathbf{O}, \mathbf{p}, \mathbf{Cal}, \mathbf{h}, h^{\max})$ can simply consist of the constraint CUMULATIVE$(\mathbf{S}, \mathbf{E}, \mathbf{h}, h^{\max})$ along with the decomposition of CALENDAROVERTIME$(S_i, E_i, O_i, p_i, \mathrm{Cal}_i)$ for each $i \in \mathcal{I}$ given by constraints (8) to (19).

5 Filtering Algorithms for the New Constraints

The strength of the new constraints over their decomposition, aside from the modeling simplification they bring, is the stronger propagation they permit. This is possible thanks to the filtering rules and algorithms presented in this section.

5.1 Propagation of the CalendarOvertime Constraints

For each task i, let $V_i(s, e, o)$ be a predicate satisfied if task i can start at time s for a duration of e with overtime o given the calendar Cal_i.

$$V_i(s, e, o) \overset{def}{\iff} s + e \leq \text{horizon} \wedge \textsc{CalendarOvertime}(s, e, o, p_i, \mathrm{Cal}_i) \tag{20}$$

Since Cal_i affects task i, est_i, lst_i, ect_i, and lct_i are redefined as follows:

$$\mathrm{est}'_i \overset{def}{=} \min\left\{s \in \left[\underline{S_i}, \overline{S_i}\right] \mid \exists e \in \left[\underline{E_i}, \overline{E_i}\right], \exists o \in \left[\underline{O_i}, \overline{O_i}\right], V_i(s, e, o)\right\} \tag{21}$$

$$\mathrm{lst}'_i \overset{def}{=} \max\left\{s \in \left[\underline{S_i}, \overline{S_i}\right] \mid \exists e \in \left[\underline{E_i}, \overline{E_i}\right], \exists o \in \left[\underline{O_i}, \overline{O_i}\right], V_i(s, e, o)\right\} \tag{22}$$

$$\mathrm{ect}'_i \overset{def}{=} \min\left\{s + e \mid s \in \left[\underline{S_i}, \overline{S_i}\right], e \in \left[\underline{E_i}, \overline{E_i}\right], o \in \left[\underline{O_i}, \overline{O_i}\right], V_i(s, e, o)\right\} \tag{23}$$

$$\mathrm{lct}'_i \overset{def}{=} \max\left\{s + e \mid s \in \left[\underline{S_i}, \overline{S_i}\right], e \in \left[\underline{E_i}, \overline{E_i}\right], o \in \left[\underline{O_i}, \overline{O_i}\right], V_i(s, e, o)\right\} \tag{24}$$

We consider that $\min(\emptyset) = \infty$ and $\max(\emptyset) = -\infty$.

Using definition (21), the checking and filtering rules for $\underline{S_i}$ in the CalendarOvertime propagator are:

$$\mathrm{est}'_i = \infty \implies \text{conflict} \tag{25}$$

$$\underline{S_i} < \mathrm{est}'_i \implies \underline{S'_i} = \mathrm{est}'_i \tag{26}$$

The filtering algorithm for the constraint CalendarOvertime is based on four precomputed vectors: Let $k \subseteq \{\mathbb{r}, \mathbb{o}\}$, be the types of time points the vectors consider. $C_i^k[t]$ is the number of time points of type in k that come before time t in Cal_i, and $Y_i^k[j]$ is the index of the j-th time point of type in k in Cal_i. With these vectors, we define helper functions that execute in constant time. For simplicity's sake, we only present sketches that ignore boundary conditions at the beginning or the end of the scheduling horizon. The function $\mathrm{count}_k(a, b) := C_i^k[b] - C_i^k[a]$ returns the number of time points in the time window $[a, b)$ with a type in k. $\mathrm{previous}_k(t) := Y_i[C_i^k[t+1] - 1]$ returns the latest time point with a type in k that is not later than t. $\mathrm{next}_k(t) := Y_i[C_i^k[t]]$ returns the earliest time point with type in k that is not earlier than t. $\mathrm{get\_end}_k(t, \Delta) := Y_i[C_i^k[t] + \Delta]$ returns the end of the smallest time window beginning at t and containing Δ time points with a type in k. Finally,

$$\begin{aligned}\mathrm{verify\_head\_tail}(s, e) :=& \mathbb{1}(\mathrm{Cal}_i[s] = \mathbb{o}) + \mathbb{1}(e > 1)\mathbb{1}(\mathrm{Cal}_i[s + e - 1] = \mathbb{o}) \\ &\leq p_i - \mathrm{count}_{\{\mathbb{r}\}}(s, s + e)\end{aligned}$$

is true if and only if the time worked regularly in $[s, s + e)$ permits enough overtime to work in overtime on the first and last time points.

This constraint requires a constant number of variables per task. Because the vectors C_i^k and Y_i^k must be precomputed, the space complexity of the filtering algorithm is linear with respect to the horizon, and the initialization (performed once when instantiating the model) is also linear. Algorithm 1 computes in constant time a candidate value for ect'_i, as redefined by (23), given a fixed starting time s for a task i subject to a calendar. Algorithm 2 verifies the value given by Algorithm 1, and filters the lower bound of S_i according to the CalendarOvertime constraint. This algorithm iterates on $\mathrm{dom}(S_i)$, computing a minimal completion time for each candidate start time. The first start time leading to a finite completion time is the new lower bound. Even though the running time complexity is in $\mathcal{O}(|\mathrm{dom}(S_i)|)$, it is technically linear w.r.t. the number of filtered-out unclosed time points. As such, the algorithm runs in constant time if it filters nothing and it runs in linear time if it filters many values. The upper bound of S_i and the other variables are processed similarly.

Algorithm 1 Computing ect'_i given calendar Cal_i, and $S_i = s$.

```
1  Function compute_completion_time($i$, $s$):
2      if $\mathrm{Cal}_i[s] = \mathbb{c}$ then return $\infty$ ;
       // The execution window contains at least $p_i$ unclosed periods.
3      $\mathrm{end} \leftarrow \mathrm{get\_end}_{\{\mathbb{r},\mathbb{o}\}}(s, p_i)$
4      if $\mathrm{end} - s < \underline{E_i}$ then
           // The associated elapsed time must be at least $\underline{E_i}$.
5          $\mathrm{end} \leftarrow \mathrm{next}_{\{\mathbb{r},\mathbb{o}\}}(s + \underline{E_i} - 1) + 1$
6      $\mathrm{worked\_regular\_time} \leftarrow \mathrm{count}_{\{\mathbb{r}\}}(s, \mathrm{end})$
7      $\mathrm{min\_worked\_regular\_time} \leftarrow p_i - \overline{O_i}$
8      if $\mathrm{worked\_regular\_time} < \mathrm{min\_worked\_regular\_time}$ then
           // At least $p_i - \overline{O_i}$ regular time must be worked.
9          $\mathrm{end} \leftarrow \mathrm{get\_end}_{\{\mathbb{r}\}}(s, \mathrm{min\_worked\_regular\_time})$
10     if $\mathrm{Cal}_i[\mathrm{end} - 1] = \mathbb{o} \land$ not $\mathrm{verify\_head\_tail}(s, \mathrm{end})$ then
11         if $\mathrm{next}_{\{\mathbb{r}\}}(\mathrm{end} - 1) + 1 \leq \mathrm{horizon}$ then
               // Adding a regular time point fixes the tail problem.
12             $\mathrm{end} \leftarrow \mathrm{next}_{\{\mathbb{r}\}}(\mathrm{end} - 1) + 1$
       /* The ending time is minimal.  Constraints on $\overline{E_i}$, $\underline{O_i}$ or head and
          tail cannot be made right if they are not already.                */
13     if $\mathrm{end} \leq \mathrm{horizon} \land \mathrm{end} - s \leq \overline{E_i} \land p_i - \mathrm{count}_{\{\mathbb{r}\}}(s, \mathrm{end}) \geq \underline{O_i}$
        $\land \mathrm{verify\_head\_tail}(s, \mathrm{end})$ then
14         return end
15     return $\infty$
```

Algorithm 2 iterates using naive unit leaps (see line 7). By analyzing the cause of why Algorithm 1 returns infinity, these leaps can be extended. For example, if the current s is a time point of type $\mathbb{r}$ and the failure is due to "end-s" at line 13 in Algorithm 1 being greater than $\overline{E}$ by k, then the "+1" in the leap could be replaced by a "+k". We have tested such enhancements but found no improvement on the performance. As such, the simpler version presented is the one used for the experimentations presented in Section 6.

5.1.1 Explaining the Propagation

In a solver with lazy clause generation, we explain propagations by rules (25) and (26) naively, respectively by (27) → False and (28) → $\llbracket \mathrm{est}'_i \leq S_i \rrbracket$.

$$\llbracket \underline{S_i} \leq S_i \rrbracket \land \llbracket S_i \leq \overline{S_i} \rrbracket \land \llbracket \underline{E_i} \leq E_i \rrbracket \land \llbracket E_i \leq \overline{E_i} \rrbracket \land \llbracket \underline{O_i} \leq O_i \rrbracket \land \llbracket O_i \leq \overline{O_i} \rrbracket \tag{27}$$

$$\llbracket \underline{S_i} \leq S_i \rrbracket \land \llbracket \underline{E_i} \leq E_i \rrbracket \land \llbracket E_i \leq \overline{E_i} \rrbracket \land \llbracket \underline{O_i} \leq O_i \rrbracket \land \llbracket O_i \leq \overline{O_i} \rrbracket \tag{28}$$

Our previous attempts indicate that computing more general explanations is of little interest for this propagator compared to using the naive ones.

Algorithm 2 Filtering $\underline{S_i}$ given a calendar.

```
Input: Variables $S_i$, $E_i$, and $O_i$.
$s \leftarrow \text{next}_{\{\texttt{r},\texttt{o}\}}(\underline{S_i})$
while $s \leq \overline{S_i}$ do
    end $\leftarrow$ compute_completion_time($i$, $s$)
    // We only need to verify that it is a valid value for $\text{ect}'_i$.
    if end $\neq \infty$ then
        $\underline{S_i} \leftarrow s$
        return Success
    $s \leftarrow \text{next}_{\{\texttt{r},\texttt{o}\}}(s+1)$
return Conflict
```

5.2 Propagation of the CumulativeOvertime Constraints

The basis of the CumulativeOvertime propagator is that of a Cumulative propagator applying the classic Time-Tabling rule. The main difference is that it uses the definitions (21) to (24), rather than (1) to (4), to compute the profile with (5) and apply the Time-Tabling rules (6) and (7). Thus, $f'(\Omega, t) = \sum_{\{i \in \Omega | t \in [\text{lst}'_i, \text{ect}'_i)\}} h_i$ and the new checking and filtering rules are as follows:

$$\exists\, t, f'(\mathcal{I}, t) > h^{\max} \implies \text{conflict} \tag{29}$$

$$\text{ect}'_i > t \land h^{\max} < h_i + f'(\mathcal{I} \setminus \{i\}, t) \implies \underline{S'_i} > t \tag{30}$$

For that, the algorithm that enforces the Time-Tabling can compute the value ect'_i by calling Algorithm 1 (and verifying the value returned) with increasing values of $s \in \text{dom}(S_i)$. The first valid value returned is the ect'_i. The lst'_i is computed symmetrically. Most propagators applying the Time-Tabling rule can be adapted for the CumulativeOvertime propagator. Because of the computing time caused by the new definitions, the complexity of the CumulativeOvertime propagator is that of its base Cumulative propagator multiplied by the size of the largest domain of the starting time variables. Since we chose to adapt the propagator by Schutt et al. [20] that has a complexity of $\mathcal{O}(n^2)$, we obtain a propagator in $\mathcal{O}(kn^2)$, where $k = \max_{i \in \mathcal{I}} |\text{dom}(S_i)|$. Under the assumption that this constraint is used alongside CalendarOvertime constraints, the size of the scope of the constraint is the same as for the Cumulative constraint (here linear in the number of tasks).

This global propagator is used in combination with the propagators for the CalendarOvertime constraints. This is done because filtering the calendar constraints solely through this global propagator specialized for resource consumption would be inefficient.

5.2.1 Explaining the Propagation

In a solver using lazy clause generation, the propagation needs to be explained. First, should the propagator fail to find a valid ect'_i at some point in its execution, it means that the CalendarOvertime constraint cannot be satisfied. The CumulativeOvertime propagator directly reports a conflict that it naively explains with (27). As such, the rest of this section considers that est'_i, lst'_i, ect'_i, and lct'_i are valid. Let $t \in [\text{lst}'_i, \text{ect}'_i)$ be a time point in the calendar-corrected compulsory part of task i. The expression profile_expl is used to construct the explanation.

$$\text{profile\_expl}(i,t) \overset{def}{=} \begin{cases} [\![S_i \leq t]\!] \wedge [\![t+1-\underline{E_i} \leq S_i]\!] \\ \qquad \wedge [\![\underline{E_i} \leq E_i]\!] & \text{if } t \in \left[\overline{S_i}, \underline{S_i}+\underline{E_i}\right) \\ \qquad (27) & \text{otherwise} \end{cases} \tag{31}$$

The expression profile_expl(i,t) depends on whether the redefinitions (21) to (24) are necessary to detect t as part of the compulsory part of task i. If the original definitions are sufficient, the explanation for t being in the compulsory part of task i is the same as presented by Schutt et al. [20], but with a variable duration. Otherwise, the explanation cannot be as general and we simply reuse the naive one presented previously.

Suppose that rule (29) finds a conflict at time t. We define $B_t \subseteq \mathcal{I}$ the set of tasks for which t is in their corrected compulsory part, i.e., $B_t = \{i \in \mathcal{I} \mid t \in [\text{lst}'_i, \text{ect}'_i)\}$. Let $B_t^* \subseteq B_t$ be a minimal set (in terms of number of elements) such that $\sum_{i \in B_t^*} h_i > h^{\max}$. Let t^+ be the smallest ect'_i or lst'_i greater than t and let t^- be the greatest ect'_i or lst'_i smaller than t. This means that every time point in the interval $[t^-, t^+]$ have the same set of tasks that have a compulsory part overlapping it, i.e., $B_t = B_{t'}$ for all $t' \in [t^-, t^+]$. As such, explaining based on any point in this interval is valid. Then, the propagator explains the conflict by:

$$\bigwedge_{i \in B_t^*} \text{profile\_expl}\left(i, \left\lfloor \frac{t^- + t^+}{2} \right\rfloor\right) \rightarrow \text{False}. \tag{32}$$

This corresponds to saying that the conflict is caused by a minimal number of tasks all having a compulsory part that includes the time point in the middle of the profile rectangle that contains t. If the calendar corrections (the new definitions (21) to (24)) are never needed, this explanation is the same as the *pointwise* explanation from Schutt et al.

For a task i and a time $t \in \left[\underline{S_i}, \text{ect}'_i\right)$, we define task_expl$(i,t)$ as follows:

$$\text{task\_expl}(i,t) \overset{def}{=} \begin{cases} [\![t+1-\underline{E_i} \leq S_i]\!] \wedge [\![\underline{E_i} \leq E_i]\!] & \text{if } t \in \left[\underline{S_i}, \underline{S_i}+\underline{E_i}\right) \\ \qquad (28) & \text{otherwise} \end{cases}. \tag{33}$$

The logical expression task_expl(i,t) depends on whether calendar corrections are needed to detect that task i, when starting at a time not earlier than time $\underline{S_i}$, is not finished by time t. If so, we use a naive explanation like for the CalendarOvertime constraint. Otherwise, we reuse the expression from Schutt et al. [20].

Suppose that rule (30) pushes $\underline{S_i}$ to time $t+1$ and that t is the earliest time for which the rule applies. Let $B_t^* \subseteq B_t \setminus \{i\}$ be a minimal set such that $\sum_{k \in B_t^*} h_k > h^{\max} - h_i$. Then, the propagator instead filters $\underline{S_i}$ to $t^* = \min\{\text{ect}'_i, t^+\}$ and explains it by:

$$\text{task\_expl}(i,\ t^*-1) \wedge \bigwedge_{k \in B_t^*} \text{profile\_expl}\,(k,\ t^*-1) \rightarrow [\![t^* \leq S_i]\!]. \tag{34}$$

Rule (30) is reapplied until it no longer filters. This cuts the propagation from rule (30) into sub-propagations permitting, according to Schutt et al. [20], more general explanations. If the calendar corrections are never needed, these explanations are the same as the ones presented by Schutt et al.

6 Experimentation

To compare the value of our new propagators with the decomposition, we solve the following RCPSP model augmented with calendars and overtime.

6.1 Experimentation Model

The model has initial constraints on the time window of each task, task precedence constraints, resources that tasks need, and calendars that tasks follow. Let $\mathcal{R}$ be the set of resources and $\mathcal{I}$ a set of tasks. Each task $i \in \mathcal{I}$ has to start in a window $[\text{minStart}_i, \text{maxStart}_i]$ and end in a window $[\text{minEnd}_i, \text{maxEnd}_i]$. These windows encode release times and deadlines. Let $\mathcal{P} \subseteq \mathcal{I} \times \mathcal{I}$ contain the precedence relationships. For each $(i, j) \in \mathcal{P}$, the task i must end before the task j may start. The release times and deadlines provide the initial domains of the variables through the propagation of the following constraints:

$$\text{minStart}_i \leq S_i \leq \text{maxStart}_i \qquad \forall i \in \mathcal{I} \tag{35}$$

$$\text{minEnd}_i \leq S_i + E_i \leq \text{maxEnd}_i \qquad \forall i \in \mathcal{I} \tag{36}$$

The following constraints enforce the precedence relationships:

$$S_i + E_i \leq S_j \qquad \forall (i, j) \in \mathcal{P} \tag{37}$$

Finally, CumulativeOvertime constraints prevent the overload of the resources.

$$\textsc{CumulativeOvertime}(\mathbf{S}, \mathbf{E}, \mathbf{O}, \mathbf{p}, \mathbf{Cal}, \mathbf{h}_j, h_j^{\max}) \qquad \forall j \in \mathcal{R} \tag{38}$$

We either minimize the makespan (39) or the overtime costs (40):

$$\max_{i \in \mathcal{I}} \{S_i + E_i\} \tag{39}$$

$$\sum_{i \in \mathcal{I}} w_i O_i \tag{40}$$

We optimize these objective functions separately, i.e., optimizing only one function or the other. When minimizing the makespan, all overtime is forbidden. Otherwise, it would also maximize the overtime, which makes little sense for an applied project, since it leads to cost maximization.

By modifying how constraint (38) is implemented, we define three equivalent models:

- The CumulativeOvertime model implements constraint (38) directly with our global CumulativeOvertime constraint.
- The CalendarOvertime model decomposes constraint (38) with a classic Cumulative constraint, and a CalendarOvertime constraint for each task.
- The decomposition model decomposes constraint (38) as described in section 4.2.

6.2 Experimentation Details

We implement[1] the CalendarOvertime and CumulativeOvertime constraints in C++ in the solver Chuffed 0.13.0[2] [8], and write our models in MiniZinc [16]. To keep the comparison with the CumulativeOvertime model fair, the propagator that filters the Cumulative constraints in the CalendarOvertime and decomposition models only uses the Time-Tabling check and filtering already implemented in Chuffed. We run all experiments with a timeout of 10 minutes on a machine with a 32-core Intel Xeon 4110 CPU @ 2.10 GHz and 32 Gb of memory. We run four executions simultaneously, which may affect the precision of the runtimes.

[1] Available at: `https://github.com/Samclou/chuffed/releases/tag/Calendars-cp2024`

[2] Available at: `https://github.com/chuffed/chuffed/releases/tag/0.13.0`

We use the instances j30, j60, j90, and j120 from the PSPLIB [11] benchmark, the instances bl20 and bl25 from the BL set [3], and the PACK [7] instances, all adapted with randomly generated calendars where time points represent hours. The instances use calendars similar to those of Boudreault et al. [6], where days have 8 regular hours, followed by 4 hours of overtime. Some calendars have weekends off, and some do not have overtime. We add for each day a 5% chance for it to be a holiday. There is a calendar where weekends and holidays are composed of 12 overtime hours. These 2135 augmented instances and the models (as well as the execution logs) are accessible in the code repository.

For makespan minimization, we extend the horizons from the original instances by a factor of 5 to prevent the addition of closed hours from leading to trivial unsatisfiable instances. In these executions, we forbid overtime. For overtime costs minimization, we must use a smaller horizon to prevent having too many instances where the best value of 0 overtime is trivial to find, but it should not be reduced so much that we get easy unsatisfiable instances. To fix the horizon, we solve the instances twice to minimize the makespan: once by forbidding overtime and a second time by allowing overtime. We fix the horizon to the mean makespan. This gives a horizon for which there is always a solution, which is often not trivial and leaves room to optimize the overtime costs. The computation time required to compute these horizons is not taken into account in our results as they are used to construct the instances rather than solving the problem.

7 Results

Comparisons are made between the decomposition and the CalendarOvertime models, and between the CalendarOvertime and the CumulativeOvertime models.

7.1 Comparing the Decomposition and CalendarOvertime Models

Figure 2 shows graphs comparing the runtimes of our models on instances for which the solver proved the optimality. Compared to the decomposition model, the CalendarOvertime model represents an average speedup of 13.8 for makespan optimization and 2.9 for overtime optimization, respectively, on these 1625 and 1373 instances.

This speedup for the makespan optimization is larger than the one for the overtime optimization. We surmise that this important discrepancy is due to the size of the horizon in the makespan optimization instances. Indeed, their horizon is often very high compared to the optimal makespan or the horizon of the overtime optimization instances. It so happens that the Element constraints present in the decomposition of the CalendarOvertime constraint are susceptible to the size of the horizon. For example, in the solver used, constraint (16) becomes a collection of clauses that may each be as long as the horizon while constraint (10) is filtered by a propagator that is linear in the size of the horizon (as long as S and E are not fixed). This leads to both weak and slow filtering which must degrade the performances of the decomposition. We can see that the performances of the decomposition model become more competitive in the context of overtime minimization, which uses a tighter horizon.

Regarding the instances not solved optimally by both models, there are no instances where the decomposition model is able to prove optimality or find a solution better than the CalendarOvertime model. The CalendarOvertime model proves optimality on 17% of the 510 makespan instances and 8% of 762 overtime instances. It finds better solutions in 81% (79%) of makespan (overtime) instances. There are 21 instances for which the decomposition model fails to find any solution while the CalendarOvertime model is able to.

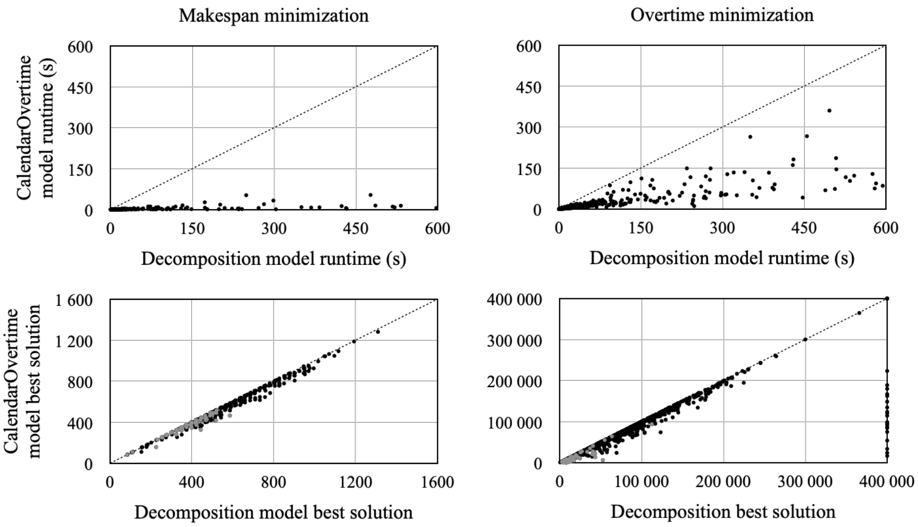


Figure 2 Comparison, between the decomposition and CalendarOvertime models, of the runtime on the instances solved by both models (1^{st} row) and the best solution found for the remaining instances (2^{nd} row) for makespan (1^{st} column) and overtime (2^{nd} column) minimization. On the 2^{nd} row, gray dots are instances solved by the CalendarOvertime model and black dots are for when all models timeout.

7.2 Comparing the CalendarOvertime and CumulativeOvertime Models

Figure 3 shows that the CumulativeOvertime model has an average speedup of 1.14 over the CalendarOvertime model for makespan optimization and 1.24 for overtime optimization, respectively, on the 1712 and 1436 instances solved optimally by both models. When comparing the best solutions found on the remaining instances, we see that, for makespan minimization, the CalendarOvertime model never proves optimality or finds a better solution than the CumulativeOvertime model. The CumulativeOvertime model proves optimality on respectively 1.6% and 1.4% of both these 423 makespan instances and the 699 overtime instances. It finds better solutions in 31% (44%) of makespan (overtime) instances. However, here, there are 5 overtime instances for which the CalendarOvertime model finds a better solution, and 1 where it proves optimality.

Thus, the CalendarOvertime constraint is a notable enhancement over the decomposition and is further improved by the CumulativeOvertime constraint.

8 Conclusion

We propose two new constraints to solve the cumulative scheduling problem with calendars and overtime. The CalendarOvertime constraint uses a precomputed substructure to enforce bounds consistency on the S_i, E_i, and O_i variables in $\mathcal{O}(|\text{dom}(S_i)|)$. The CumulativeOvertime constraint adapts the Time-Tabling rule to take calendars into account. Experiments on PSPLIB, BL, and PACK instances augmented with calendars show that the models using the specialized propagators of the new constraints outperform a model using a decomposition, the CumulativeOvertime constraints being a further enhancement over the CalendarOvertime constraints. These new constraints could also help solve the resource investment problem, the multi-mode resource-constraint project scheduling problem or even disjunctive problems such as job shop when they are augmented with calendars and overtime.

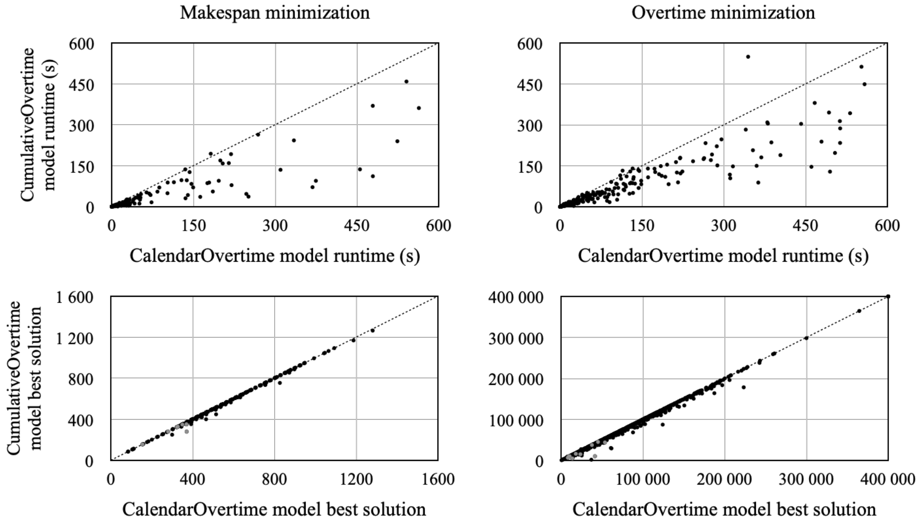


Figure 3 Comparison, between the CALENDAROVERTIME and CUMULATIVEOVERTIME models, of the runtime on the instances solved by both models (1^{st} row) and the best solution found for the remaining instances (2^{nd} row) for makespan (1^{st} column) and overtime (2^{nd} column) minimization. On the 2^{nd} row, gray dots are instances solved by the CALENDAROVERTIME model and black dots are for when all models timeout.

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. doi:10.1016/0895-7177(93)90068-A.

2 Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Springer, 2001.

3 Philippe Baptiste and Claude Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, 5(1):119–139, 2000. doi:10.1023/A:1009822502231.

4 Nicolas Beldiceanu. Global constraints as graph properties on structured network of elementary constraints of the same type. *SICS Technical report T2000-01*, 2000.

5 Global constraint catalog: Calendar. https://sofdem.github.io/gccat/gccat/Ccalendar.html#uid15664. Accessed: 2024-04-04.

6 Raphaël Boudreault, Vanessa Simard, Daniel Lafond, and Claude-Guy Quimper. A constraint programming approach to ship refit project scheduling. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, volume 235, pages 10:1–10:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CP.2022.10.

7 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, 2003. doi:10.1016/S0377-2217(02)00763-4.

8 Geoffrey Chu. *Improving Combinatorial Optimization*. PhD thesis, The University of Melbourne, 2011.

9 Michael R. Garey and David S. Johnson. *Computers and intractability : a guide to the theory of NP-Completeness*. A Series of books in the mathematical sciences. W.H. Freeman, 1979.

10 Steven Gay, Renaud Hartert, and Pierre Schaus. Simple and scalable time-table filtering for the cumulative constraint. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP 2015)*, pages 149–157. Springer International Publishing, 2015. doi:10.1007/978-3-319-23219-5_11.

11 Rainer Kolisch and Arno Sprecher. Psplib - a project scheduling problem library: Or software - orsep operations research software exchange program. *European Journal of Operational Research*, 96(1):205–216, 1997. doi:10.1016/S0377-2217(96)00170-1.

12 Stefan Kreter, Julia Rieck, and Jürgen Zimmermann. Models and solution procedures for the resource-constrained project scheduling problem with general temporal constraints and calendars. *European Journal of Operational Research*, 251(2):387–403, 2016. doi:10.1016/j.ejor.2015.11.021.

13 Stefan Kreter, Andreas Schutt, and Peter J Stuckey. Using constraint programming for solving RCPSP/max-cal. *Constraints*, 22(3):432–462, 2017. doi:10.1007/s10601-016-9266-6.

14 Stefan Kreter, Andreas Schutt, Peter J. Stuckey, and Jürgen Zimmermann. Mixed-integer linear programming and constraint programming formulations for solving resource availability cost problems. *European Journal of Operational Research*, 266(2):472–486, 2018. doi:10.1016/j.ejor.2017.10.014.

15 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling. *Constraints*, 23(2):210–250, 2018. doi:10.1007/s10601-018-9281-x.

16 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, pages 529–543. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-74970-7_38.

17 Wilhelmus Petronella Maria Nuijten. *Time and resource constrained scheduling: a constraint satisfaction approach*. PhD thesis, Technische Universiteit Eindhoven, 1994.

18 Olga Ohrimenko, Peter J Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009. doi:10.1007/s10601-008-9064-x.

19 Pierre Ouellet and Claude-Guy Quimper. Time-table extended-edge-finding for the cumulative constraint. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP 2013)*, pages 562–577. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-40627-0_42.

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

Slide&Drill, a New Approach for Multi-Objective Combinatorial Optimization

João Cortes
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Inês Lynce
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Vasco Manquinho
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Abstract

Following the successful use of Propositional Satisfiability (SAT) algorithms in Boolean optimization (e.g., Maximum Satisfiability), several SAT-based algorithms have been proposed for Multi-Objective Combinatorial Optimization (MOCO). However, these new algorithms either provide a small subset of the *Pareto front* or follow a more exploratory search procedure and the solutions found are usually distant from the Pareto front.

We extend the state of the art with a new SAT-based MOCO solver, Slide and Drill (Slide&Drill), that hones an *upper bound set* of the exact solution. Moreover, we show that Slide&Drill neatly complements proposed UNSAT-SAT algorithms for MOCO. These algorithms can work in tandem over the same shared "blackboard" formula, in order to enable a faster convergence.

Experimental results in several sets of benchmark instances show that Slide&Drill can outperform other SAT-based algorithms for MOCO, in particular when paired with previously proposed UNSAT-SAT algorithms.

2012 ACM Subject Classification Computing methodologies → Optimization algorithms

Keywords and phrases Multi-Objective Combinatorial Optimization, Satisfiability Algorithms

Digital Object Identifier 10.4230/LIPIcs.CP.2024.8

Funding This work was supported by Portuguese national funds through FCT, under projects UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020), PTDC/CCI-COM/2156/2021 (DOI: 10.54499/-PTDC/CCI-COM/2156/2021) and 2022.03537.PTDC (DOI: 10.54499/2022.03537.PTDC).

1 Introduction

In real-world problems it is common to have several objective functions to optimize [17, 19, 29]. For instance, when updating a system such as a Linux installation [13], one can try to maximize the number of packages to be updated from the current version to the most recent one, while at the same time minimizing the number of software packages from the current installation to be removed in the update process. It is usually the case that the objective functions are conflicting, i.e., decreasing one objective function results in having to increase the value of another objective function. Hence, in Multi-Objective Combinatorial Optimization (MOCO), the goal is to try to find all Pareto-optimal solutions, i.e., all solutions for which one cannot improve the value of a function without worsening the value of another one. The set of all Pareto-optimal solutions is known as the Pareto front.

Following the success of Propositional Satisfiability (SAT) algorithms in Boolean optimization problems such as Maximum Satisfiability (MaxSAT) [2] or Pseudo-Boolean Optimization (PBO) [24], several algorithms for MOCO have been proposed based on iterative calls to a satisfiability solver [10, 22, 28, 26]. For instance, the Guided-Improvement Algorithm (GIA) [22] starts with a feasible solution and iteratively checks if there is some other solution

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 8; pp. 8:1–8:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that is better on all objective functions. When the iterative process ends, the algorithm has found a Pareto-optimal solution and new constraints are added so that only assignments that improve on at least one objective function are feasible (i.e., solutions that are worse on all objectives are blocked). More recently, the notion of P-minimal models [26] was introduced, where this blocking is done using a propositional clause.

The issue with GIA and P-minimal algorithms is that the search process is focused on iteratively improving upon one solution until a Pareto-optimal solution is found. Considering that the set of solutions in the Pareto front can be large, in many instances, these algorithms are only able to find a very small subset of the Pareto front within a given time limit. Moreover, it can be the case that the Pareto-optimal solutions are skewed to optimize some objective function and do not provide a broad representation of solutions in the Pareto front.

This paper proposes `Slide&Drill`, a new exact [9], generic algorithm for MOCO that maintains an *upper cover* of the Pareto front, made of feasible of solutions. `Slide&Drill` repeatedly selects a point from the cover to improve upon. This improvement starts with a *drill* operation followed by a series of *slide* operations that generate another upper cover that is closer to the Pareto front. Hence, at any point of time, one can obtain a diversified set of solutions that approximate the Pareto front. Experimental results on representative sets of MOCO instances show that `Slide&Drill` provides better approximations of the Pareto front than previous SAT-based MOCO solvers since `Slide&Drill` is able to find a more diverse set of solutions for the end user.

The paper is organized as follows. Section 2 formally defines the MOCO problem and provides an overview of previous SAT-based MOCO algorithms. Section 3 defines lower and upper bound sets. Section 4 introduces the new `Slide&Drill` algorithm for MOCO based on iterative refinement of an upper bound set. Additionally, Section 4 also proves the correctness of the `Slide&Drill` algorithm and shows how to pair it in tandem with other MOCO algorithms. Section 5 explores different configurations of the `Slide&Drill` algorithm and compares it against other state-of-the-art SAT-based MOCO solvers using three different metrics. Finally, the paper concludes in Section 6.

2 Preliminaries

We start with the definitions that fall under SAT's domain. Next, we introduce the definitions specific to MOCO. Moreover, we briefly review previous approaches to solving MOCO.

2.1 Boolean Satisfiability

▶ **Definition 1** (CNF Formula). *Let $V = \{x_1, \dots, x_n\}$ denote a set of n Boolean variables. A literal is either a variable $x_i \in V$ or its negation $\bar{x}_i$. A clause is a disjunction of literals. A formula in Conjunctive Normal Form (CNF) ϕ is a conjunction of clauses.*

An *assignment* or *model* ν defines a truth value for all variables. Let $\nu(x_i)$ denote the truth value of variable x_i and let $\nu(l_i)$ denote the truth value of a literal l_i. We have $\nu(l_i) = \top$ if $l_i = x_i$ and $\nu(x_i) = \top$, or if $l_i = \bar{x}_i$ and $\nu(x_i) = \bot$. Otherwise, we have $\nu(l_i) = \bot$. A clause c is satisfied if at least one of its literals is true. An assignment ν is said to satisfy a formula ϕ if it satisfies all its clauses. We extend the notation of assignments to define the truth value of a clause c and a CNF formula ϕ as $\nu(c)$ and $\nu(\phi)$, respectively. In the remainder of the paper, we use the set notation for formulas (set of clauses, meaning its conjunction) and clauses (set of literals, meaning its disjunction).

▶ **Definition 2** (Boolean Satisfiability (SAT)). *Given a CNF formula ϕ, the Boolean Satisfiability (SAT) problem is to decide if there is any assignment ν to the variables in ϕ that satisfies it or prove that no such assignment exists.*

Let ϕ be a CNF formula and α a set of unitary clauses. A SAT solver call is denoted by $\phi\text{-}\mathtt{SAT}(\alpha)$, and its value decides the satisfiability of $\phi \cup \alpha$, i.e., it checks the satisfiability of ϕ assuming all literals in α are true. Note that if $\alpha = \emptyset$, then the solver checks the satisfiability of ϕ. If the query is satisfiable, then the call returns a satisfiable model. Otherwise, it returns a null value, written as $\emptyset$.

2.2 Single and Multi-Objective Combinatorial Optimization

▶ **Definition 3** (Linear Pseudo-Boolean function and Pseudo-Boolean formulas). *A linear*[1] pseudo-Boolean (PB) function $f : \{0,1\}^n \to \mathbb{N}$ *computes a weighted sum of its literals,*

$$f(\boldsymbol{x}) = f(x_1 \dots x_n) = \sum_{i=1}^{n} w_i l_i \quad , w_i \in \mathbb{N},\ x_i \in V,\ l_i \in \{x_i, \bar{x}_i\}. \tag{1}$$

Pseudo-Boolean constraints *generalize propositional clauses, and can be written as $f(\boldsymbol{x}) \bowtie k$, $\bowtie \in \{\leq, \geq, =\}$. A PB formula is a conjunction of PB constraints.*

▶ **Definition 4** (Pseudo-Boolean Optimization (PBO)). *Given a PB formula ϕ, an assignment ν is said* (ϕ-)feasible *if it satisfies all constraints in ϕ. Given a PB formula ϕ and a PB function f to minimize, the goal of Pseudo-Boolean Optimization (PBO) is to find an assignment ν that satisfies ϕ and minimizes the value of $f(\boldsymbol{x})$, where $\boldsymbol{x} \equiv (\nu(x_1), \dots, \nu(x_n))$.*

Next, we generalize PBO to the multi-objective case. Multi-objective optimization builds upon a criterion of comparison (or order) of tuples of numbers. This paper uses the *Pareto order or dominance.*

▶ **Definition 5** (Pareto partial order ($\prec$)). *Let Y be some subset of $\mathbb{N}^n$. For any $\boldsymbol{y}, \boldsymbol{y}' \in Y$,*

$$\begin{aligned} \boldsymbol{y} \preceq \boldsymbol{y}' &\iff \forall i, \boldsymbol{y}_i \leqslant \boldsymbol{y}'_i, \\ \boldsymbol{y} \prec \boldsymbol{y}' &\iff \boldsymbol{y} \preceq \boldsymbol{y}' \wedge \boldsymbol{y} \neq \boldsymbol{y}'. \end{aligned}$$

We say $\boldsymbol{y}$ dominates *$\boldsymbol{y}'$ iff $\boldsymbol{y} \preceq \boldsymbol{y}'$. We say $\boldsymbol{y}$* strictly-dominates *$\boldsymbol{y}'$ iff $\boldsymbol{y} \prec \boldsymbol{y}'$.*

Given a tuple of objective functions sharing a common domain X, we can compare two elements $\boldsymbol{x}, \boldsymbol{x}' \in X$ by comparing the corresponding tuples in the objective space. We use the term *multi-objective function* to denote an array of functions.

▶ **Definition 6** (Pareto Dominance ($\prec$)). *Let $F : X \to Y \subseteq \mathbb{N}^n$ be a* multi-objective function, *mapping the* decision space *X into the* objective space *Y. For any $\boldsymbol{x}, \boldsymbol{x}' \in X$,*

$$\begin{aligned} \boldsymbol{x} \prec \boldsymbol{x}' &\iff F(\boldsymbol{x}) \prec F(\boldsymbol{x}'), \\ \boldsymbol{x} \preceq \boldsymbol{x}' &\iff F(\boldsymbol{x}) \preceq F(\boldsymbol{x}'). \end{aligned}$$

We say $\boldsymbol{x}$ dominates *$\boldsymbol{x}'$ iff $\boldsymbol{x} \preceq \boldsymbol{x}'$. We say $\boldsymbol{x}$* strictly-dominates *$\boldsymbol{x}'$ iff $\boldsymbol{x} \prec \boldsymbol{x}'$.*

[1] Note We will drop the *linear* qualifier hereafter, as we will only work with linear functions and constraints.

As a consequence of this comparison criterion, different *optimal* solutions may be mapped to different points in the objective space, which does not happen in the single objective case. Therefore, the solution to the problem is actually a set, called *Pareto front*. These solutions are optimal in the sense that for each, there is no other feasible solution that strictly dominates them.

▶ **Definition 7** (Pareto front). *Given a a multi-objective function $F : X \to Y$ and a* feasible space *$Z \subseteq X$, the* Pareto front *of Z is a subset $P \subseteq Z$ containing all elements that are not strictly-dominated,*

$$P = \{\, \boldsymbol{x} \in Z : \nexists \boldsymbol{x}' \in Z : \boldsymbol{x}' \prec \boldsymbol{x} \,\} .$$

Let the image front of Z, *or simply* front of Z, *be the unique subset $\overline{Y} \subseteq Y$ that is the image of P under F,*

$$\overline{Y} \equiv \operatorname{front}_Z F = \{\, \boldsymbol{y} \in Y : \exists \boldsymbol{x} \in P : \boldsymbol{y} = F(\boldsymbol{x}) \,\} .$$

Finally, let argument front of Z,*denoted by* $\operatorname{arg\,front}_Z$, *be any subset $\overline{Z}$ of the Pareto Front P that is mapped under F into $\overline{Y}$ in a one-to-one fashion.*

▶ **Definition 8** (Multi-Objective Combinatorial Optimization (MOCO)). *Let $F : X \to Y \subseteq \mathbb{N}^n$ be a* multi-objective PB function, *mapping the* decision space $X \subseteq \{\,0, 1\,\}^n$ *into the* objective space Y. *Let $Z \subseteq X$ be the image under $\nu \mapsto \boldsymbol{x} = \nu(V) \equiv (\nu(x_1), \dots, \nu(x_n))$ of the feasible space of a PB formula ϕ, with variables in V.*

The goal of MOCO is to find a $\operatorname{front}_\phi F \equiv \operatorname{front}_{Z(\phi)} F$, i.e., the complete set of non-dominated objective points $\boldsymbol{y} \in Y$ whose preimage under F is ϕ-feasible. A MOCO instance will be denoted by the triple $\langle \phi, V, F \rangle$.

A remark: most applications require the production of $\operatorname{arg\,front}_Z F$, which is one of the preimages under F of $\operatorname{front}_Z F$. Our non-standard choice was made bearing in mind the clarity of the discussion and of the algorithms's presentation. In any case, the implementation of the algorithms returns an $\operatorname{arg\,front}_{Z(\phi)} F$, as usual. The pseudo-code can be adapted to do the same, but it will get significantly clobbered without adding much in the way of ideas.

▶ **Example 9.** Let $\langle \phi, V, F \rangle$ denote a MOCO instance defined over $V = \{x_1, x_2, x_3\}$, with two objective functions to minimize $F = (f_1, f_2)$ where $f_1(\boldsymbol{x}) = 2x_1 + x_2$, $f_2(\boldsymbol{x}) = 2\bar{x}_2 + 2x_3$ and $\phi = \{x_1 + x_2 + x_3 \geq 2\}$. In this case, there are two Pareto-optimal solutions: $\nu_1 = \{(x_1, 0), (x_2, 1), (x_3, 1)\}$ with costs $(1, 2)$ and $\nu_2 = \{(x_1, 1), (x_2, 1), (x_3, 0)\}$ with costs $(3, 0)$. Note that ν_1 provides a better value for f_1, while ν_2 is able to improve on f_2. All other satisfiable assignments to ϕ are dominated by either $\nu_1(V)$ or $\nu_2(V)$.

2.3 Encoding of Pseudo-Boolean Functions

In several SAT-based optimization algorithms, PB objective functions are encoded into CNF [8, 24]. In MOCO, we are interested in blocking feasible solutions that are dominated by some other feasible solution. In order to achieve this goal, one can use *unary counter* [3, 15, 16] encodings.

▶ **Definition 10** (Unary Counter). *Let $f_i : \{\,0, 1\,\}^n \to \mathbb{N}$ be a PB function and set V be an ordered set of variables that parametrize the domain of f_i,*

$$V = \{\, x_1, \dots, x_n \,\}, f_i(\boldsymbol{x}) = f_i(x_1, \dots, x_n) \qquad (2)$$

Algorithm 1 `P-Minimal` Algorithm.

```
   Input  : $\langle \phi, V, F \rangle$                                // MOCO instance
   Output : $\text{front}_\phi\, F$                                    // one img-front
1  $(\widetilde{\phi}, O) \leftarrow$ EncodeCNF$(F, V)$               // build unary counters
2  $\phi \leftarrow \phi \cup \widetilde{\phi}$
3  $I \leftarrow \emptyset$
4  $\nu' \leftarrow \phi$-SAT$(\emptyset)$                             // find first feasible model
5  while $\nu' \neq \emptyset$ do
6      while $\nu' \neq \emptyset$ do
7          $\boldsymbol{x} \leftarrow \nu'(V)$, $\boldsymbol{y} \leftarrow F(\boldsymbol{x})$
8          $\alpha \leftarrow \left\{ \left\{ \bar{o}_{i,\boldsymbol{y}_i+1} \right\}, 1 \leqslant i \leqslant m \right\}$
9          $c \leftarrow \left\{ \bar{o}_{i,\boldsymbol{y}_i}, 1 \leqslant i \leqslant m \right\}$
10         $\phi \leftarrow \phi \cup \{ c \}$                         // block region dominated by $\boldsymbol{y}$
11         $\nu' \leftarrow \phi$-SAT$(\alpha)$                        // look for $\boldsymbol{y}'$ that dominates $\boldsymbol{y}$
12     end
13     $I \leftarrow I \cup \{ \boldsymbol{y} \}$                      // save optimal solution $\boldsymbol{y}$
14     $\nu' \leftarrow \phi$-SAT$(\emptyset)$                         // find new non-dominated solution
15 end
16 return $I$
```

Consider the CNF formula $\widetilde{\phi}$ with variables $V \cup O$, where $V \cap O = \emptyset$ and O contains one variable $o_{i,k}$ for each value $k \in \mathbb{N} : \exists \boldsymbol{x} : k = f_i(\boldsymbol{x})$. The elements of O are the order variables. *We call the tuple $\left\langle f_i, V, O, \widetilde{\phi} \right\rangle$ an unary counter of f_i iff all feasible models ν of $\widetilde{\phi}$ satisfy*

$$f_i(\boldsymbol{x}) \geqslant k \Leftrightarrow o_{i,k}, \quad \boldsymbol{x} = \nu(V). \tag{3}$$

2.4 SAT-based algorithms for MOCO

One approach for solving MOCO is through Minimal Correction Subset (MCS) enumeration since all Pareto-optimal solutions are MCSs of the MOCO formula [28]. After enumerating the formula's MCSs, one can filter out the non-optimal solutions. The main advantage of the MCS enumeration is that it is not necessary to encode the objective functions into CNF since, in some cases, the encoding of objective functions can dominate the size of the resulting CNF formula [8].

Soh et al. [26] show that with a unary representation of the objective functions (see section 2.3), it is possible to establish a one-to-one correspondence between the P-minimal models and Pareto-optimal solutions of a MOCO instance.

Algorithm 1 illustrates the `P-Minimal` algorithm. It starts by finding any feasible solution (line 4). Next, it iteratively improves that solution until a Pareto-optimal solution is found (lines 6-12). Each time a new solution is found, all dominated solutions are blocked using a single clause (line 10). Afterwards, the process repeats if there are other non-dominated solutions (line 14). Otherwise, the algorithm ends and returns the Pareto front (line 16).

The `P-Minimal` algorithm can be seen as a particular case of the Guided-Improvement Algorithm (GIA) [22]. The algorithm structure is the same, but `P-Minimal` uses a single clause to block dominated solutions instead of a disjunction of PB constraints. Recently, new UNSAT-SAT and Hitting Set-based algorithms have also been proposed [5] and can be seen as a generalization of core-guided Maximum Satisfiability (MaxSAT) algorithms for MOCO. Other adaptations of MaxSAT techniques have been proposed for MOCO [14, 12], including preprocessing techniques [11].

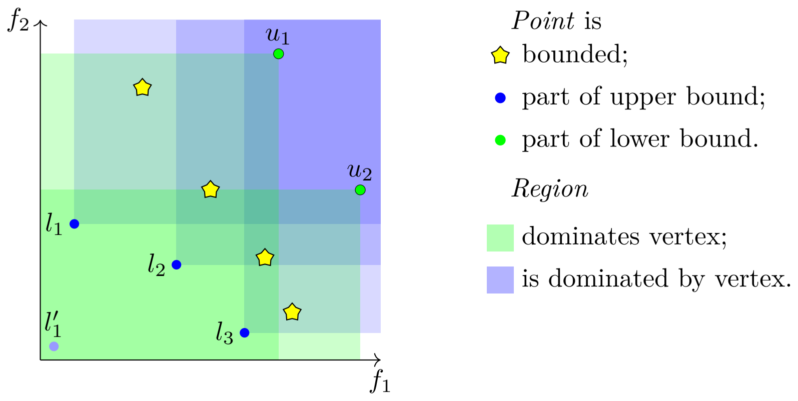


Figure 1 Bound sets (Definition 13) of some starred set $\overline{Y}$. The points $\{ l_1, l_2, l_3 \}$ form a *lower bound set* L. Dropping l_1 breaks *coverage*. This lower bound set is also *thin*, and adding l'_1 would make it "thick". The singleton set $\{ l'_1 \}$ is also a thin, lower bound set, not only of $\overline{Y}$ but also of L. The points $U = \{ u_1, u_2 \}$ form a thin, *upper bound set* of $\overline{Y}$. Note how it implies that U is an upper set of L too. And, by the same token, of $\{ l'_1 \}$. The set $\overline{Y}$ could be the image front of some MOCO instance.

3 Upper and Lower Bound Sets

Given that the Pareto order is just a *partial order* in the mathematical sense, there is no warrant to expect the existence of a *least element* of the feasible objective space. At the same time, the Pareto order reduces to the canonical *total order* of the integers when there is only one objective. The generalization of the order requires a generalization of the concept of "bounds". In particular, it is useful to deal in *bound sets* (Definition 13) that can contain more than one element.

We consider two different comparison predicates over sets. Let A and B be any two sets of points in the objective space. Then, 1) is A a *lower/upper cover* of B?, and 2) is A *a lower/upper bound set of* B?

▶ **Definition 11** (*Lower* and *upper covers*). *Let A and B be subsets of some decision space X, equipped with a multi-objective function F. Then, A* covers B from below*, or A* is a lower cover *of B, iff every element of B is dominated by some element of A,*

$$\forall b \in B, \exists a \in A : a \preceq b.$$

A strictly covers *B, or A* is a strict lower-cover *of B, iff*

$$\forall b \in B, \exists a \in A : a \prec b.$$

Also, we define an upper cover *analogously. In particular, B is an* upper cover *of A iff for every element of A there is some element of B that is dominated,*

$$\forall a \in A, \exists b \in B : a \preceq b.$$

The strict version trivially follows.

▶ **Definition 12** (Thin/thick sets). *A set A is* thin *if it does not contain distinct comparable elements,*

$$\neg\exists a_1, a_2 \in A : a_1 \neq a_2 \wedge a_1 \preceq a_2 \tag{4}$$

Otherwise, A is thick.

▶ **Definition 13** (Lower and upper bound sets). *Let L, U and Z be subsets of some decision space X*[2]*, equipped with a multi-objective function F. $L \subseteq X$ is a* (strictly) lower bound set of *$Z \subseteq X$ iff L (strictly) covers Z from below and L is thin. If L is a lower bound set of Z, we say $L \preceq Z$. If it is a strictly lower bound set, we say $L \prec Z$.*

$U \subseteq X$ is a (strictly) upper bound set of *$Z \subseteq X$ iff U (strictly) covers Z from above and U is thin. If U is an upper bound set of Z, we say $U \succ Z$. If it is a strictly upper bound set, we say $U \succeq Z$.*

Figure 1 provides examples of lower and upper bound sets. Let the starred points correspond to the optimal front in the objective space. Any optimal element in the front will be dominated by at least one element of any lower bound set (e.g., $\{l_1, l_2, l_3\}$). Similarly, any element of the front dominates at least one element of the upper bound set (e.g., $\{u_1, u_2\}$).

Let u_{max} be the *maximal point*, that is, the point whose coordinates are the largest values of each objective. Then, the singleton set $\{u_{max}\}$ is clearly an upper bound set, although not necessarily satisfiable. Analogously, the singleton set containing the origin is a lower bound set.

By computing a *satisfiable* upper bound set, we get an approximated view of the real front. If we improve this upper bound set slowly but surely, we will eventually stop, given a sufficient amount of time. At that point, the upper bound set coincides with the front.

4 Slide&Drill, an Upper-Bound Set Improver

We propose a new algorithm for MOCO, named Slide&Drill (Algorithm 2). Like P-Minimal, it is a SAT-UNSAT algorithm backed by a SAT oracle. By design, P-Minimal drills down the objective space, following a "greedy" path to optimal solutions. In contrast, Slide&Drill is a comprehensive algorithm that interleaves the drilling phase with a sliding one that diversifies [25] the flushed-out solutions.

4.1 Algorithm Description

We will go over the details of Slide&Drill (Algorithm 2). There is an illustration of the intuition behind the algorithm's dynamic in Figure 2.

P-Minimal (Algorithm 1) attempts to get to optimal solutions quickly by always moving to a dominator of the current point, and so it tries to go "down" towards the origin, so to speak. It assumes good approximations of the sought-after front should, above all, contain optimal solutions as soon as possible and that by diving in this fashion, it will flush them out quicker. But that may not be the case for every problem and application domain. And even if it is true that Pareto-optimal solutions can be found sooner, it may be more important to have a broad, diverse approximation with solutions that are feasible but not necessarily optimal.

In comparison, Slide&Drill moves less eagerly and more comprehensively in the direction of the front. It interleaves two mechanisms, *drill* and *slide*, that communicate through a *waiting list* of points and move the incumbent set down until it matches the exact front. The union of the incumbent set and the waiting list will contain an upper bound set of the exact result whenever a new drill is started.

[2] Although we define bound sets as part of the decision space, we will use their image in the objective space as a proxy throughout the description of the algorithms.

Algorithm 2 Slide&Drill, Slide and Drill MOCO solver.

```
    Input  : $\langle \phi, V, F \rangle$                          // MOCO instance
    Output : $\text{front}_\phi\, F$                               // one img-front
 1  $(\widetilde{\phi}, O) \leftarrow$ EncodeCNF$(F, V)$           // build unary counters
 2  $\phi \leftarrow \phi \cup \widetilde{\phi}$
 3  $W \leftarrow \{ u_{max} \}$                                   // maximal point
 4  $I \leftarrow \emptyset$
 5  while $W \neq \emptyset$ do                                    // drill
 6      $\boldsymbol{\omega} \leftarrow \langle \textit{select and remove} \rangle (W)$
 7      $\alpha \leftarrow \{ \{ \bar{o}_{i,\boldsymbol{\omega}_i+1} \} : i \in 1 \ldots m \}$      // set up drill at $\boldsymbol{\omega}$
 8      $\alpha' \leftarrow \emptyset$
 9      $\nu \leftarrow \phi$-SAT$(\alpha)$
10      while $\nu \neq \emptyset$ do                              // slide
11          $\boldsymbol{x} \leftarrow \nu(V)$, $\boldsymbol{y} \leftarrow F(\boldsymbol{x})$
12          $\phi \leftarrow \phi \cup \{ \{ \bar{o}_{i,\boldsymbol{y}_i} : i \in 1 \ldots m \} \}$   // block $\boldsymbol{y}$ dominated region
13          $I \leftarrow I \backslash \{ \boldsymbol{y}' \in I : \boldsymbol{y} \preceq \boldsymbol{y}' \} \cup \{ \boldsymbol{y} \}$   // update incumbent set
14          $W \leftarrow W \cup \{ \boldsymbol{y} \}$              // update waiting list
15          $\alpha' \leftarrow \alpha' \cup \{ \{ o_{i,\boldsymbol{y}_i} : i \in 1 \ldots m \} \}$   // temp. focus non-dominating
16          $\nu \leftarrow \phi$-SAT$(\alpha \cup \alpha')$
17      end
18  end
19  return $I$
```

After the initialization and the encoding of the unary counters, the external *drill* loop (line 5) hones the incumbent set I, as long as it is possible to do so. When we drill at site $\boldsymbol{\omega}$ (line 6), we look for points that dominate $\boldsymbol{\omega}$ (i.e., solutions "below" $\boldsymbol{\omega}$). This is accomplished by line 7 and the semantics of the unary counters. The first drill site is the maximal point u_{max}, (line 3), i.e., the point whose coordinates in the objective space are the maximal values of the objective functions. The $\langle$*select and remove*$\rangle$ procedure fetches an element of W while removing it and can be implemented using different strategies. When the waiting list is depleted, the drill loop stops. At that point, the incumbent set I is the complete solution, and the algorithm returns. The waiting list is expanded by the inner *slide* loop (lines 10-16). The *waiting list* W takes in freshly found solutions that will eventually be used to start another drill. Besides, the solutions are also placed into the incumbent set I that represents the best approximation of the front so far. The incumbent set will be reported if the solver cannot finish under the resource limits.

This slide loop is the main distinction between Slide&Drill and P-Minimal. Instead of drilling until striking an optimal solution, as the P-Minimal algorithm does, we steer the oracle so as to slide across the objective space, collecting solutions that do not dominate each other. This is accomplished by building the auxiliary formula α' while accruing the waiting list. The formula α' contains one clause per point found since the start of the last slide loop (line 15) and blocks the region under the known solutions. As soon as the solver fails to find an extra point the slide loop is complete and the implicit upper bound set contained in the union of the waiting list and the incumbent set was made whole again. Figure 2 provides a small example of the execution of the Slide&Drill algorithm.

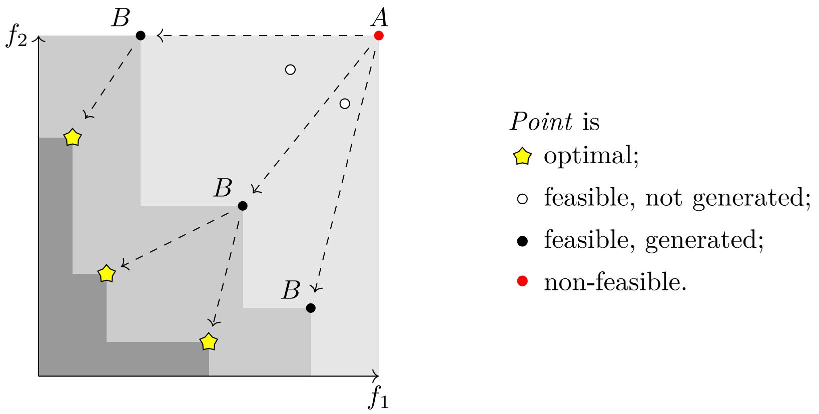


Figure 2 Illustration of a run of the `Slide&Drill` (Algorithm 2). Three upper bound sets are produced, marked by A, B and the star. The first drill site is the maximal point. We drill and find one of the elements marked with B. The remaining ones are generated by the slide loop. Note this is our first satisfiable upper bound set. Assume the next drill site, chosen by $\langle$*select and remove*$\rangle$, is B's midpoint. As we drill again, either of the two optimal solution is found, and the slide generates the other. The uppermost B point is chosen next, and the missing optimal solution is found during the subsequent drill. There are 4 remaining drill sites to consider: the three optimal solutions and the lowermost B point. Neither will produce new solutions, and the algorithm terminates after four more "blank" drills. All B elements are dominated, and they are pushed out of I by the addition of the optimal solutions. The shading levels vary as the number of upper bound sets that dominate the region. The lighter tone is painted by A only, while the darker is painted by all three.

The waiting list can be backed by different containers. We consider both a *stack* (i.e., FIFO container) and a *queue* (i.e., LIFO container). Different containers result in different implementations of $\langle$*select and remove*$\rangle$ (line 6), and hence a different concrete `Slide&Drill`.

- If a *stack* is used the algorithm resembles `P-Minimal`, but it is not quite the same. It is safer because it will perform a slide step, and hence diversify the incumbent set before drilling further. If the computation results in timeout, the pool of solutions will differ from what `P-Minimal` would have found. There is a trade-off between the number of optimal points (probably larger with `P-Minimal`) and the diversity of the points obtained;
- If a *queue* is used, the algorithm is substantially different from `P-Minimal`. We expect less optimal solutions but more robust approximations. This is a more extreme approach than the one resulting from using a stack. It will further tilt the scale in the favor of diverse but suboptimal points.

4.2 Algorithm Properties

Let us prove `Slide&Drill` (Algorithm 2) is *sound* and *complete* (Lemma 17)

▶ **Lemma 14.** *Any optimal point that dominates the drill site* $\boldsymbol{\omega}$ *will dominate at least one of the points generated by the associated slide loop (line 10).*

Proof. Assume that Lemma 14 is not true. Then, there must exist an optimal point $\boldsymbol{y}$ that dominates $\boldsymbol{\omega}$ but fails to dominate any of the generated points. In that case, the temporary constraints added at line 15 do not render $\boldsymbol{y}$ unsatisfiable, and because $\boldsymbol{y}$ is optimal, neither do the permanent constraints added at line 12. And therefore, $\boldsymbol{y}$ must have been generated. And that contradicts the assumption because $\boldsymbol{y}$ dominates itself. ◀

▶ **Lemma 15.** *At the start of the outer loop (line 5), the union of the optimal points in the incumbent set I with the waiting list W contains an upper bound set of the front $\overline{Y} = \mathrm{front}_Z\, F$.*

Proof. This is true for the first run because the waiting list contains the maximal point.

Assume Lemma 15 true at the start of iteration i, and let U be an upper bound set contained in $I \cap \overline{Y} \cup W$. We want to prove that an upper bound set U' is contained in $I' \cap \overline{Y} \cup W'$, where I' and W' are the incumbent set and waiting list at the start of iteration $i+1$.

Let $W' = W \setminus \{\,\boldsymbol{\omega}\,\} \cup \Delta W$, where ΔW is the set accrued by the successive executions of line 14. We will prove that $C = U \setminus \{\,\boldsymbol{\omega}\,\} \cup \Delta W$ is an upper-cover of $\overline{Y}$. All solutions $\boldsymbol{y} \in \overline{Y}$ that do not dominate $\boldsymbol{\omega}$ are covered by elements in U. Solutions $\boldsymbol{y}$ that do dominate $\boldsymbol{\omega}$ are covered by elements in ΔW, by Lemma 14.

If the upper cover C is thin, then $U' = C$. Otherwise, for any pair of comparable elements $\boldsymbol{y} \preceq \boldsymbol{y}' \in C$, drop $\boldsymbol{y}$. The obtained set is a cover because any point dominating $\boldsymbol{y}$ dominates $\boldsymbol{y}'$ too. The remaining elements of C are incomparable and are collected into U' so that $U' \subseteq C$.

To see that $U' \subseteq I' \cap \overline{Y} \cup W'$,

$$U' \subseteq C = U \setminus \{\,\boldsymbol{\omega}\,\} \cup \Delta W \implies \tag{5}$$

$$U' \subseteq (I \cap \overline{Y} \cup W) \setminus \{\,\boldsymbol{\omega}\,\} \cup \Delta W \implies \tag{6}$$

$$U' \subseteq (I' \cap \overline{Y} \cup W) \setminus \{\,\boldsymbol{\omega}\,\} \cup \Delta W \implies \tag{7}$$

$$U' \subseteq (I' \cap \overline{Y}) \setminus \{\,\boldsymbol{\omega}\,\} \cup W \setminus \{\,\boldsymbol{\omega}\,\} \cup \Delta W \implies \tag{8}$$

$$U' \subseteq (I' \cap \overline{Y}) \setminus \{\,\boldsymbol{\omega}\,\} \cup W' = (I' \cap \overline{Y} \cup W') \setminus \{\,\boldsymbol{\omega}\,\} \subseteq I' \cap \overline{Y} \cup W', \tag{9}$$

where Equation (7) follows because only dominated solutions can be removed from I, and Equation (9) follows because $\boldsymbol{\omega}$ does not belong to W'. ◀

▶ **Lemma 16.** *At the start of the outer loop (line 5), any point in I that does not belong to W is optimal.*

Proof. All points are added to both I and W. If some point $\boldsymbol{\omega}$ does not belong to W, then it must have been removed by line 6. After that, the query will return an empty model iff $\boldsymbol{\omega}$ is optimal because the restrictions in ϕ block only dominated regions, and the assumptions focus the search over the region dominating $\boldsymbol{\omega}$. If $\boldsymbol{\omega}$ is not optimal, the query at line 16 will generate a point that dominates it, and that point will push off $\boldsymbol{\omega}$ from I at line 13. ◀

▶ **Proposition 17.** *Algorithm 2 is sound and complete.*

Proof. Let us prove soundness first. If the algorithm returns, W is empty. By Lemma 15, I contains an upper bound set. By Lemma 16, all its elements are optimal. Every element of the front dominates at least one element of I. Assume $\boldsymbol{y}$ is optimal and is not part of I. It must be dominated by some element of I, but an optimal point is dominated only by itself. Hence, $\boldsymbol{y}$ cannot be absent from I.

Let us move on to show the algorithm is complete. The clauses added by line 12 block at least one feasible model each, as they block the dominated region, including its defining vertex. Because no blocking clause is ever dropped, the number of satisfiable queries is bounded by the number of satisfiable models, which is finite.

After entering the slide loop at line 10, it will fail to return iff there is an infinite number of satisfiable queries, which cannot happen, given the former argument.

Therefore, every operation occurring in the drill loop (line 5) ends successfully in a finite amount of time. Therefore, the loop exits iff W becomes empty.

Note that the waiting list receives new elements only at line 14. Based on the argument above, the number of inserted elements must always be finite. Each iteration of the drill loop takes one element out. Assume this loop never ends. Eventually, the number of removals would catch up to the number of insertions, and the waiting list would be empty. But then, the loop would end, which contradicts the hypothesis. ◀

4.3 Tandem `Slide&Drill`

Two different solvers working together will most likely produce better results than any of them would by themselves.

Suppose we have two different approximated fronts A and B of a MOCO instance, produced respectively by solvers a and b. Consider also the combined solution $A\frown B$, built from $A \cup B$ by weeding out any dominated point from the union. Most likely, $A\frown B$ is a better approximation of the front than any of the solutions A and B by themselves. And it cannot be worse. Even more, had they shared the incrementally built approximations on the fly, the workers would have guided each other and avoided regions of the objective space that were already branded as dominated by some feasible solution produced by the other contributor.

Because `Slide&Drill` is a SAT-UNSAT solver, it makes sense to consider for its companion an UNSAT-SAT solver. We chose a previously proposed UNSAT-SAT algorithm named `Core-Guided` [5]. The workers (i.e., `Slide&Drill` and `Core-Guided`) will share a single, *incrementally* built formula. Note that the unary counters representing the objective functions are shared, as is the SAT oracle.

In order to synchronize their work, there is a *conflict budget*. The solvers will work in turn: as soon as the assigned budget is fully depleted on SAT calls the current worker stops, and the other contributor kicks in with a restored budget.

For the `Slide&Drill` algorithm, we simply reinsert the last drill site into the waiting list and proceed. For the `Core-Guided` algorithm, we keep track of the current upper-fence and bootstrap the next search session by setting the upper fence to the backed-up value.

5 Results and Analysis

5.1 Benchmark Sets and Experimental Setup

In order to evaluate our MOCO algorithms against other state-of-the-art MOCO solvers, we consider two publicly available benchmark sets of MOCO instances that have already been used in previous research works.

The Development Assurance Level (DAL) [4] benchmark set [3] is composed of 95 instances encoding different levels of rigour in the development of a software or hardware component of an aircraft. The development assurance level defines the assurance activities aimed at eliminating design and coding errors that could affect the safety of an aircraft. The goal is to allocate the smallest DAL to functions to decrease the development costs.

The Package Upgradeability (PU) benchmark set is composed of 687 instances from the Mancoosi International Solver Competition [18]. Each instance encodes the upgradeability of packages in an open-source system. The `packup` tool [13] was used to generate variants containing between two and five objectives to optimize. This results in 3570 instances.

[3] `https://www.lifl.fr/LION9/challenge.html`.

All of the experiments were conducted on a computer with Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz running Linux Debian 10.2. Each problem instance was executed for each MOCO solver with a memory limit of 32 GB and a CPU timeout of 10 minutes (600 seconds) imposed using the `runsolver` [23] tool.

5.2 Evaluated Algorithms

We evaluate our algorithms against several SAT-based MOCO solvers. The `ParetoMCS` algorithm [4] is based on the enumeration of MCSs of the MOCO instance [28] and the `P-Minimal` algorithm implements the SAT-UNSAT approach presented in Algorithm 1 [26]. Additionally, the `Core-Guided` algorithm implements a complementary UNSAT-SAT approach [5].

The `Slide&Drill` algorithm implements our new approach proposed in Algorithm 2. Furthermore, both the `Slide&Drill` and `P-Minimal` approaches are combined with the `Core-Guided` algorithm, (as described in section 4.3).

All algorithms are implemented using the publicly available codebase [5] from the authors of the `Core-Guided` algorithm [5]. Hence, all algorithms use the selection delimiter encoding [16] to represent the objective functions. Furthermore, the underlying SAT solver is also the same and used incrementally [7, 21, 1]. As a result, the observed differences in performance are mainly from the algorithmic techniques employed and a more fair comparison is achieved.

5.3 Evaluation Metrics

Finding the Pareto front of MOCO instances is computationally harder than solving single-objective optimization problems. In most cases, given an acceptable time limit, solvers can only provide an approximation of the Pareto front.

Let $\mathcal{A}$ denote a set of algorithms and variants to be evaluated and let $\mathcal{I}$ denote the set of instances. Let $Y_{i,j}$ denote the approximation of the Pareto front provided by algorithm A_i ($A_i \in \mathcal{A}$) for instance I_j ($I_j \in \mathcal{I}$). Let R_j denote the reference set for instance I_j defined as $R_j = \cup_{A_i \in \mathcal{A}} Y_{i,j}$, where only the incomparable elements are kept, i.e., all dominated solutions are filtered out of R_j. Hence, the reference set R_j contains only the best solutions found by any of the evaluated algorithms in $\mathcal{A}$.

To evaluate the quality of the approximations provided by each tool, we use three different metrics. The first metric is the *Contribution indicator* that measures the contribution of a given algorithm to the reference set. Hence, the contribution indicator of algorithm $A_i \in \mathcal{A}$ in a MOCO instance I_j is defined as $\frac{|Y_{i,j} \cap R_j|}{|R_j|}$. Clearly, *larger* values are preferable since the metric is maximized when the algorithm is able to identify all solutions in the reference set.

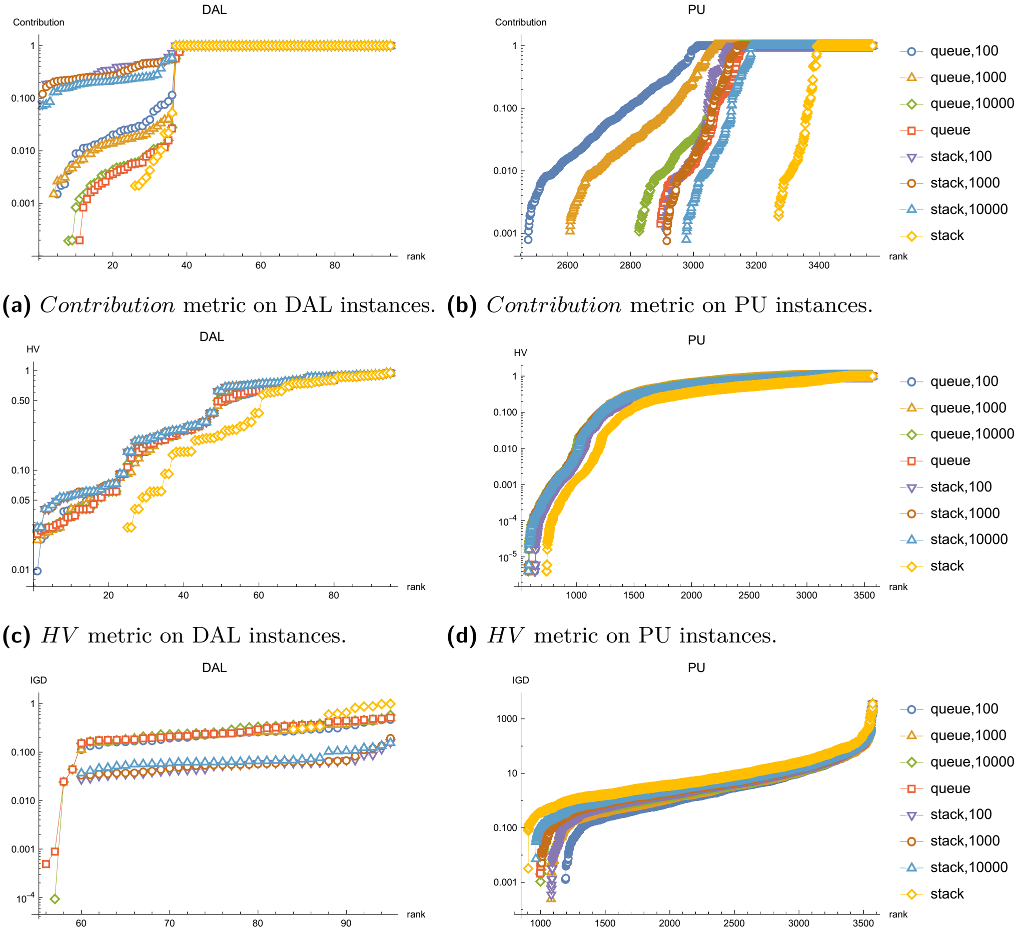
The second metric is the Hypervolume (HV) indicator [31]. This indicator measures the volume of the objective space between the set of nondominated solutions $Y_{i,j}$ and a given reference point u_r. The reference point depends on the benchmark. For a given instance I_j, the reference point is set to the largest possible objective values in the reference set R_j [6]. As in the previous indicator, *larger* values of HV are preferable since the volume of the dominated objective space is maximized at the Pareto front.

Finally, the third metric is the Inverted Generational Distance (IGD) indicator [30, 6]. IGD measures the average Euclidean distance, in the objective space, between the reference set R_j and the solution set $Y_{i,j}$ returned by the algorithm. In this case, *smaller* values of IGD are preferable, meaning that the solution set $Y_{i,j}$ is closer to the reference set R_j.

[4] https://gitlab.ow2.org/sat4j/moco

[5] https://gitlab.inesc-id.pt/u001810/moco

[6] If the reference set R_j is the Pareto front, then the reference point u_r is the Nadir point [20].



(a) *Contribution* metric on DAL instances. **(b)** *Contribution* metric on PU instances.

(c) *HV* metric on DAL instances. **(d)** *HV* metric on PU instances.

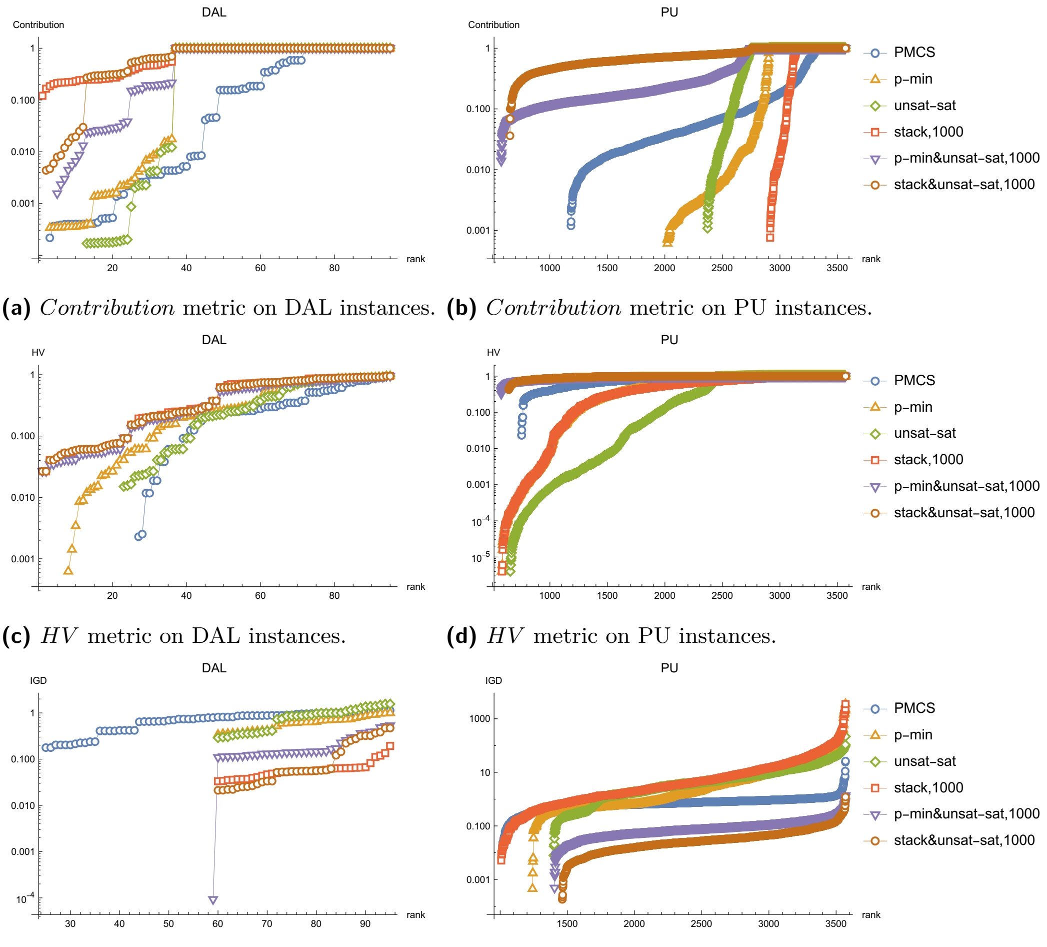
(e) *IGD* metric on DAL instances. **(f)** *IGD* metric on PU instances.

Figure 3 Comparison of the *Contribution*, *IGD* and *HV* results for each set of instances. `Slide&Drill` variants only. Each series is sorted independently, smaller values first. Vertical scale is logarithmical. Each series is labelled by the type of waiting list and the value of the conflict budget.

5.4 Slide and Drill Variants

The `Slide&Drill` algorithm (Algorithm 2) can be configured in different ways. In this section we focus on the management of the waiting list and the SAT solver call. As mentioned in section 4.1, the waiting list can be managed as a stack or as a queue and this results in exploring the search space in different ways. Additionally, one can set the SAT solver call with a limited budget of conflicts in order for `Slide&Drill` not to get "stuck". Setting up a conflict budget will not violate neither soundness nor completeness, since the site of the unfinished drill goes back into the waiting list and all SAT calls are done in an incremental fashion (i.e., the same SAT solver instance is always used and no learned clause is ever removed).

Figure 3 shows the results of several variants of the `Slide&Drill` algorithm for the three metrics defined in section 5.3 for both the DAL (left) and PU (right) benchmark sets. The `stack` and `queue` variants denote that the waiting list is managed as a stack and queue, respectively. Moreover, whenever the `stack` and `queue` variants are followed by a number C,



(a) *Contribution* metric on DAL instances. **(b)** *Contribution* metric on PU instances.

(c) *HV* metric on DAL instances. **(d)** *HV* metric on PU instances.

(e) *IGD* metric on DAL instances. **(f)** *IGD* metric on PU instances.

Figure 4 Comparison of the *Contribution*, *IGD* and *HV* results for each set of instances. Each series is sorted independently, smaller values first. Vertical scale is logarithmical.

then C denotes the conflict limit in the SAT call. Whenever the conflict limit is reached, the SAT call ends and the Slide&Drill algorithm retrieves a new starting point from the waiting list. Otherwise, no limit is imposed on the SAT call.

The experimental results in these benchmark sets show that the algorithm performs better when a conflict limit is imposed. This occurs for all metrics in both benchmark sets. The budgeted SAT call allows the algorithm to choose a new element of the waiting list, allowing it to find a wider variety of solutions that better approximates the Pareto front.

We obtained mixed results regarding the waiting list's management. While the stack variants perform better for the DAL benchmark set, the queue variants perform better on PU instances. This assay is based on the contribution metric, as the overall values for HV and IGD are similar.

5.5 Comparison with Other MOCO Solvers

We compare the stack, 1000 variant of the Slide&Drill algorithm (stack strategy for management of the waiting list and $C = 1000$ for the conflict limit on the SAT solver) against other state-of-the-art MOCO solvers. We chose this variant of the Slide&Drill algorithm since it seems to be the most balanced one, considering the results from the previous section.

The results on DAL (left) and PU (right) benchmarks considering the three metrics are available in Figure 4. For the DAL benchmarks, the new `Slide&Drill` algorithm is able to outperform the `ParetoMCS` (`PMCS`), `Core-Guided` (`unsat-sat`) and `P-Minimal` (`p-min`) algorithms on all metrics. The approximation of the Pareto front provided by `Slide&Drill` on these instances is clearly better than the ones produced by all other algorithms. Due to the newly proposed strategy, `Slide&Drill` is able to find a more diverse set of solutions and, thus, a more accurate approximation of the Pareto front. Furthermore, even when `Slide&Drill` and `Core-Guided` work in tandem (`stack&unsat-sat`), there are only very slight improvements to the contribution metric.

On the PU benchmarks, the `Slide&Drill` (`stack, 1000`) algorithm is able to find solutions close to the `P-Minimal` (`p-min`) algorithm considering both the HV and IGD metrics. Moreover, it is able to outperform the `Core-Guided` (`unsat-sat`) algorithm. However, the `ParetoMCS` (`PMCS`) is the best standalone algorithm in terms of HV and IGD. Nevertheless, when `Slide&Drill` is paired with `Core-Guided` in tandem (`stack&unsat-sat`), then this approach is clearly better on all metrics on the PU benchmark set. This is due to the high complementarity of these algorithms when applied on the PU instances. Observe that the `P-Minimal`, when paired with `Core-Guided` in tandem (`p-min&unsat-sat`), also improves its performance. However, the tandem `Slide&Drill` and `Core-Guided` still performs better on all metrics due to the higher diversification of solutions provided by our new `Slide&Drill` algorithm.

6 Conclusions and Future Work

This paper introduces the Slide and Drill approach for solving MOCO problems. The proposed `Slide&Drill` algorithm is a SAT-based algorithm with a strategy to diversify the set of solutions found such that a better approximation of the Pareto front can be found. Previously proposed algorithms either disregard the objective function representation (e.g., through the enumeration of MCS) or have too much focus on proving that a given solution is Pareto-optimal, resulting in being able to identify only a small set of the Pareto front.

Experimental results on two representative sets of benchmarks show that the new `Slide&Drill` algorithm outperforms previous SAT-based MOCO solvers on three different metrics. Moreover, the performance of the `Slide&Drill` algorithm can be additionally boosted when paired with a complementary `Core-Guided` approach. Hence, the newly proposed algorithms further enhance the usage of SAT-based approaches for MOCO.

The Slide and Drill approach introduced in this paper can be configured using different techniques to diversify the exploration of the search space. In this paper we exploit several strategies to choose elements of a waiting list that correspond to areas of the search space still to explore. In future work, we propose to manage the waiting list as a priority queue using a performance metric such as the Hypervolume as the selection criterion. Although this criterion has already been used in other algorithmic contexts [27], using it in a tandem algorithm with both `Slide&Drill` and `Core-Guided` approaches poses new additional challenges.

References

1 Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In Matti Järvisalo and Allen Van Gelder, editors, *16th International Conference on Theory and Applications of Satisfiability Testing, SAT 2013, Helsinki, Finland, July 8-12, 2013*, volume 7962 of *Lecture Notes in Computer Science*, pages 309–317. Springer, 2013. doi:10.1007/978-3-642-39071-5_23.

2 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 929–991. IOS Press, 2021. doi:10.3233/FAIA201008.

3 Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In Francesca Rossi, editor, *International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003. doi:10.1007/978-3-540-45193-8_8.

4 Pierre Bieber, Remi Delmas, and Christel Seguin. Dalculus - theory and tool for development assurance level allocation. In Francesco Flammini, Sandro Bologna, and Valeria Vittorini, editors, *Computer Safety, Reliability, and Security - 30th International Conference, SAFECOMP 2011, Naples, Italy, September 19-22, 2011. Proceedings*, volume 6894 of *Lecture Notes in Computer Science*, pages 43–56. Springer, 2011. doi:10.1007/978-3-642-24270-0_4.

5 João Cortes, Inês Lynce, and Vasco M. 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 2023, 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.

6 Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints. *IEEE Trans. Evol. Comput.*, 18(4):577–601, 2014. doi:10.1109/TEVC.2013.2281535.

7 Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In Ofer Strichman and Armin Biere, editors, *First International Workshop on Bounded Model Checking, BMC@CAV 2003, Boulder, Colorado, USA, July 13, 2003*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 543–560. Elsevier, 2003. doi:10.1016/S1571-0661(05)82542-3.

8 Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006. doi:10.3233/sat190014.

9 Matthias Ehrgott, Xavier Gandibleux, and Anthony Przybylski. Exact methods for multi-objective combinatorial optimisation. In Salvatore Greco, Matthias Ehrgott, and José Rui Figueira, editors, *Multiple Criteria Decision Analysis: State of the Art Surveys*, pages 817–850. Springer New York, New York, NY, 2016. doi:10.1007/978-1-4939-3094-4_19.

10 Marco Gavanelli. An algorithm for multi-criteria optimization in csps. In *European Conference on Artificial Intelligence*, pages 136–140. IOS Press, 2002.

11 Christoph Jabs, Jeremias Berg, Hannes Ihalainen, and Matti Järvisalo. Preprocessing in sat-based multi-objective combinatorial optimization. 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 18:1–18:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CP.2023.18.

12 Christoph Jabs, Jeremias Berg, Andreas Niskanen, and Matti Järvisalo. Maxsat-based bi-objective boolean optimization. In *International Conference on Theory and Applications of Satisfiability Testing*, volume 236 of *LIPIcs*, pages 12:1–12:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.12.

13 Mikolás Janota, Inês Lynce, Vasco M. Manquinho, and João Marques-Silva. Packup: Tools for package upgradability solving. *J. Satisf. Boolean Model. Comput.*, 8(1/2):89–94, 2012. doi:10.3233/sat190090.

14 Mikolás Janota, António Morgado, José Fragoso Santos, and Vasco M. Manquinho. The seesaw algorithm: Function optimization using implicit hitting sets. In *International Conference on Principles and Practice of Constraint Programming*, volume 210 of *LIPIcs*, pages 31:1–31:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.31.

15 Saurabh Joshi, Ruben Martins, and Vasco M. Manquinho. Generalized totalizer encoding for pseudo-boolean constraints. In *International Conference Principles and Practice of Constraint Programming*, volume 9255 of *LNCS*, pages 200–209. Springer, 2015. doi:10.1007/978-3-319-23219-5_15.

16 Michal Karpinski and Marek Piotrów. Encoding cardinality constraints using multiway merge selection networks. *Constraints*, 24(3-4):234–251, 2019. doi:10.1007/s10601-019-09302-0.

17 Rui Li, Qinghua Zheng, Xiuqi Li, and Zheng Yan. Multi-objective optimization for rebalancing virtual machine placement. *Future Gener. Comput. Syst.*, 105:824–842, 2020. doi:10.1016/j.future.2017.08.027.

18 Mancoosi international solver competition 2011. https://www.mancoosi.org/misc-2011/index.html.

19 Rafael Marques, Luís M. S. Russo, and Nuno Roma. Flying tourist problem: Flight time and cost minimization in complex routes. *Expert Syst. Appl.*, 130:172–187, 2019. doi:10.1016/j.eswa.2019.04.024.

20 Kaisa Miettinen. *Nonlinear Multiobjective Optimization*, volume 12. Springer Science & Business Media, 2012.

21 Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In Alessandro Cimatti and Roberto Sebastiani, editors, *15th International Conference on Theory and Applications of Satisfiability Testing - SAT 2012, Trento, Italy, June 17-20, 2012*, volume 7317 of *Lecture Notes in Computer Science*, pages 242–255. Springer, 2012. doi:10.1007/978-3-642-31612-8_19.

22 Derek Rayside, H.-Christian Estler, and Daniel Jackson. The guided improvement algorithm for exact, general-purpose, many-objective combinatorial optimization. Technical Report Technical Report MIT-CSAIL-TR-2009-033, MIT Massachusetts Institute of Technology, 2009.

23 Olivier Roussel. Controlling a Solver Execution with the runsolver Tool: System description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):139–144, November 2011. doi:10.3233/SAT190083.

24 Olivier Roussel and Vasco M. Manquinho. Pseudo-boolean and cardinality constraints. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 695–733. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-695.

25 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.

26 Takehide Soh, Mutsunori Banbara, Naoyuki Tamura, and Daniel Le Berre. Solving multiobjective discrete optimization problems with propositional minimal model generation. In *International Conference Principles and Practice of Constraint Programming*, volume 10416 of *LNCS*, pages 596–614. Springer, 2017. doi:10.1007/978-3-319-66158-2_38.

27 Satya Tamby and Daniel Vanderpooten. Enumeration of the nondominated set of multiobjective discrete optimization problems. *INFORMS J. Comput.*, 33(1):72–85, 2021. doi:10.1287/IJOC.2020.0953.

28 Miguel Terra-Neves, Inês Lynce, and Vasco M. Manquinho. Introducing pareto minimal correction subsets. In *International Conference on Theory and Applications of Satisfiability Testing*, volume 10491 of *LNCS*, pages 195–211. Springer, 2017. doi:10.1007/978-3-319-66263-3_13.

29 Yuan Yuan and Wolfgang Banzhaf. ARJA: automated repair of java programs via multi-objective genetic programming. *IEEE Trans. Software Eng.*, 46(10):1040–1067, 2020. doi:10.1109/TSE.2018.2874648.

30 Qingfu Zhang and Hui Li. MOEA/D: A multiobjective evolutionary algorithm based on decomposition. *IEEE Trans. Evol. Comput.*, 11(6):712–731, 2007. doi:10.1109/TEVC.2007.892759.

31 E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications.* PhD thesis, University of Zurich, Zürich, Switzerland, 1999.

Pseudo-Boolean Reasoning About States and Transitions to Certify Dynamic Programming and Decision Diagram Algorithms

Emir Demirović
TU Delft, The Netherlands

Ciaran McCreesh
University of Glasgow, Scotland

Matthew J. McIlree
University of Glasgow, Scotland

Jakob Nordström
University of Copenhagen, Denmark
Lund University, Sweden

Andy Oertel
Lund University, Sweden
University of Copenhagen, Denmark

Konstantin Sidorov
TU Delft, The Netherlands

Abstract

Pseudo-Boolean proof logging has been used successfully to provide certificates of optimality from a variety of constraint- and satisifability-style solvers that combine reasoning with a backtracking or clause-learning search. Another paradigm, occurring in dynamic programming and decision diagram solving, instead reasons about partial states and possible transitions between them. We describe a framework for generating clean and efficient pseudo-Boolean proofs for these kinds of algorithm, and use it to produce certifying algorithms for knapsack, longest path, and interval scheduling. Because we use a common proof system, we can also reason about hybrid solving algorithms: we demonstrate this by providing proof logging for a dynamic programming based knapsack propagator inside a constraint programming solver.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Discrete optimization

Keywords and phrases Proof logging, dynamic programming, decision diagrams

Digital Object Identifier 10.4230/LIPIcs.CP.2024.9

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.12574620`

Funding *Emir Demirović*: part of the XAIT lab funded by the Delft AI Labs programme.
Ciaran McCreesh: supported by a Royal Academy of Engineering research fellowship, and by the Engineering and Physical Sciences Research Council [grant number EP/X030032/1].
Jakob Nordström: supported by the Swedish Research Council grant 2016-00782 and the Independent Research Fund Denmark grant 9040-00389B.
Andy Oertel: supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.
Konstantin Sidorov: part of the XAIT lab funded by the Delft AI Labs programme.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 9; pp. 9:1–9:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Acknowledgements Part of this work was carried out while taking part in the semester program *Satisfiability: Theory, Practice, and Beyond* in 2021 at the Simons Institute for the Theory of Computing at UC Berkeley, and in the extended reunion of this semester program in the spring of 2023. This work has also benefited greatly from discussions during the Dagstuhl Seminars 22411 *Theory and Practice of SAT and Combinatorial Solving* and 23261 *SAT Encodings and Beyond*.

1 Introduction

It is sometimes vital that combinatorial solving algorithm implementations can be trusted to give correct answers. To this end, when claiming that a problem has no solution, Boolean satisfiability (SAT) solvers do not just assert unsatisfiability, but also provide an independently verifiable proof of this fact, in one of several standard formats such as *DRAT* [20, 19, 35], *LRAT* [10], or *VeriPB* [13]. The proof can then be inspected by a formally verified proof checker to assert its correctness. This means the algorithm is *certifying* [28]: while we still cannot trust that the implementation is correct, this does guarantee that if it ever gives an incorrect answer, then we can detect it.

Of the above proof formats, *VeriPB* is the most general-purpose: as well as supporting advanced SAT-solving techniques such as parity reasoning [18], symmetry and dominance breaking [4], and MaxSAT optimisation [1], it has also been used for subgraph-finding algorithms [16, 14, 15] and for constraint programming with a variety of global constraints [17, 29]. In these latter settings, a *VeriPB* proof resembles a description of a backtracking search tree, interleaved with justifications of facts obtained from inference algorithms or constraint propagation. However, the *VeriPB* proof format has no direct notion of a search tree. Instead, its underlying proof system is powerful enough to express implicational reasoning. In particular, constraints may be reified and dereified, and if some fact can be derived, it can also be derived under a sequence of guesses with (almost) no additional effort. This is in contrast to, e.g., the VIPR proof format [8], which was designed specifically for mixed integer programming and which has explicit notions of assumptions and closing branches that function independently from other proof rules. An advantage of a sufficiently powerful proof system that does not have a direct notion of search is that techniques like restarts [16] and autotabulation [17] can be encoded without needing additions to the proof system.

However, there are non-search-based ways of solving hard problems. Both dynamic programming and decision diagram algorithms can be viewed as working with partial states, and transitions between those states [22, 3]. In this work, we show that *VeriPB* can also be used for efficient proof logging for algorithms that work with states and transitions, rather than search, regardless of whether the algorithm uses memoisation, a matrix, or a layer-by-layer construction. This is primarily because the pseudo-Boolean constraints and extended cutting planes proof system underlying *VeriPB* makes it very clean to work with implications.

Using a common system, rather than inventing a new proof system for dynamic programming proofs, has several benefits: it allows us to reason about hybrid or nested solving strategies that use more than one kind of algorithm, it avoids the need to reinvent proof logging for various kinds of constraint and dominance reasoning, and it gives us immediate access to a suite of proof checking tools which would otherwise be expensive to recreate. To illustrate this, we have implemented proof logging for a knapsack constraint inside a constraint programming solver, whose propagator involves reasoning about paths through a dynamic programming table or decision diagram to detect loss of support for values in constraint programming variables [34].

2 Background

Before we can talk about proofs for dynamic programming problems, we give a brief overview of the *VeriPB* proof system, and outline how it has been used to generate proofs for backtracking search algorithms.

2.1 Pseudo-Boolean Preliminaries

Although designed to support many different kinds of solvers, the foundations of the *VeriPB* proof system are Boolean variables and pseudo-Boolean constraints. Let x_i be a set of Boolean variables ranging over 0 (false) and 1 (true). We write $\overline{x}_i$ to mean $1 - x_i$ (i.e. not x), and refer to x_i and $\overline{x}_i$ as *literals*. A pseudo-Boolean (PB) constraint over literals ℓ_i is an inequality in the form $\sum_i c_i\ell_i \bowtie A$, where $\bowtie$ is either $\geq$ or $\leq$ and c_i and A are integer constants. A PB constraint can always be rewritten in *normalised form* $\sum_i c_i\ell_i \geq A$ with all literals over distinct variables and all c_i and A non-negative, and when describing the proof system we will assume constraints are normalised. A PB optimisation problem is a set of PB constraints, together with an objective $\sum_i c_i\ell_i$ to be minimised.

Let $C = \sum_i c_i\ell_i \geq A$ be a PB constraint, and y and y_j be distinct literals. We define $\overline{C}$ to mean $\sum_i c_i\ell_i \leq A{-}1$; $\wedge_j y_j \Rightarrow C$ to mean $\sum_j K\overline{y}_j + \sum_i c_i\ell_i \geq A$ where $K = A - \sum_i \min(c_i, 0)$; and $y \Leftrightarrow C$ to mean the pair of PB constraints $y \Rightarrow C$ and $\overline{y} \Rightarrow \overline{C}$. It is easy to check that the constraints defined in this way have the meaning suggested by the notation used. Note how, unlike for Boolean formulae in conjunctive normal form (CNF), full reification of a pseudo-Boolean constraint by a literal requires only a pair of constraints.

2.2 The *VeriPB* Proof System

In a *VeriPB* proof, we begin with a set of pseudo-Boolean constraints as input – these are assumed, as axioms, and so they must accurately describe the high-level problem being solved. A proof is then a sequence of pseudo-Boolean constraints, where each new constraint follows either obviously or by explicit construction from the input and any other constraints already derived, in such a way that at least one optimal solution is always preserved.

When proof steps consist of explicit constructions, they are given as a sequence of *cutting planes* steps [7], as follows. For any literal ℓ_i, we may freely introduce a constraint $\ell_i \geq 0$. Given two constraints $\sum_i a_i\ell_i \geq A$ and $\sum_i b_i\ell_i \geq B$, we may add them together to derive $\sum_i(a_i + b_i)\ell_i \geq A + B$. We may also multiply by a positive integer constant c, to get $\sum_i ca_i\ell_i \geq cA$, or (assuming normalised form) divide to get $\sum_i \left\lceil \frac{a_i}{c} \right\rceil \ell_i \geq \left\lceil \frac{A}{c} \right\rceil$. Finally, we can *saturate*, turning (again assuming normalised form) $\sum_i a_i\ell_i \geq A$ into $\sum_i \min(a_i, A)\, \ell_i \geq A$.

A clausal constraint, or *clause*, is one of the form $\sum_i \ell_i \geq 1$. This corresponds naturally to a Boolean clause in CNF. By *resolution*, we mean deriving $\sum_i x_i + \sum_j y_j \geq 1$ from the clauses $r + \sum_i x_i \geq 1$ and $\overline{r} + \sum_j y_j \geq 1$; this may be achieved by adding the constraints and then saturating [21]. In particular, resolution allows us to take the clauses $r \Rightarrow \sum_i x_i \geq 1$ and $r + \sum_j y_j \geq 1$ and derive $\sum_i x_i + \sum_j y_j \geq 1$. Proof steps such as this that involve implications are generally straightforward in cutting planes: for example, if we have both $r \Rightarrow \sum_i a_i x_i \geq A$ and $s \Rightarrow r$, we may easily derive that $s \Rightarrow \sum_i a_i x_i \geq A$ by multiplication and then addition. As a special case of this, if we have established that the left hand side of an implication must be true, then we can dereify the implication and derive its right hand side unconditionally. Another useful fact, which we use repeatedly throughout this work, is that if we have a process for deriving a constraint D from a set of constraints C_i, then we can reuse this process to derive a reified version of D if we are given a set of reified constraints C'_i; we explain this in detail in the appendix.

An alternative to cutting planes steps is to allow the proof verifier to add constraints that are obvious enough that they do not require an explicit derivation. A constraint C follows by *reverse unit propagation* (RUP) if adding $\overline{C}$ to the existing set of constraints leads immediately to contradiction upon achieving integer bounds consistency for each constraint individually [9]. Obviously such constraints are implied, and this condition can be verified efficiently, so a RUP constraint may safely be added as a proof step. (The term *unit propagation* is used due to the SAT origins of proof logging [12]; if all constraints are clauses, integer bounds consistency and unit propagation are equivalent.) As with cutting planes proofs, RUP proof procedures can trivially be modified to work subject to reifications.

The *VeriPB* proof system also has a non-implicational *strengthening* rule [4]. We do not use the full generality of the rule in this paper, but will use it as an *extension* rule. An *extension variable* z reifying an arbitrary PB constraint C is a variable which has not previously been used, which is introduced in a proof alongside the pair of constraints $z \Leftrightarrow C$; the strengthening rule can be used to introduce an extension variable in this way. We will also use strengthening to implement *fusion resolution*: given $r \Rightarrow \sum_i a_i x_i \geq A$ and $\overline{r} \Rightarrow \sum_i a_i x_i \geq A'$, strengthening lets us derive that $\sum_i a_i x_i \geq \min(A, A')$.

A proof of unsatisfiability ends by deriving $0 \geq 1$. For an optimisation problem with objective expression $\sum_i c_i \ell_i$, a *VeriPB* proof will conclude by demonstrating that the objective lies between two integer lower and upper bounds – for an exact solution, these will be the same. To do this, a proof step may witness a solution by giving a partial assignment to variables. The proof checker verifies that this assignment unit propagates to a complete feasible assignment to all variables, and then introduces a new objective-improving constraint $\sum_i c_i \ell_i \leq A - 1$ where A is the calculated objective value from the assignment.

Finally, we may also delete derived constraints, under certain conditions. This will lower the amount of memory required to verify the proof, as well as potentially speeding up verification of RUP and strengthening steps. For soundness reasons, there are restrictions on when constraints may be deleted (e.g. to prevent us from deleting every constraint in the input and then claiming an optimal solution with zero cost) [4], but for the techniques used in this paper, the verifier will allow us to delete any constraint we introduce, as well as any extension variable by deleting its two defining constraints.

2.3 A Framework for Proofs for Backtracking Search

For a very simple backtracking search algorithm, a proof could consist of a RUP statement for every backtrack, asserting that at least one of the guessed assignments must be false. Alternatively, if we are using conflict-driven clause learning (CDCL), a proof consists of a RUP step for every learned clause in turn. This applies to proofs using either *DRAT* or *VeriPB*. However, this is only possible if every fact used by the search algorithm follows by integer bounds consistency on the PB representation of the problem (or, for *DRAT*, from unit propagation on the CNF representation). This would suffice, e.g. for conventional DPLL or CDCL SAT solvers, but does not work if we have stronger propagation or inference algorithms such as domain-consistent all-different. In this case, it is necessary to help the proof checker by interleaving additional steps within the proof [17]. The nature of these steps depends upon the inference being performed, and can involve additional RUP steps or (in *VeriPB* proofs only) explicit cutting planes steps. The aim here is to ensure that any fact "known" to the solving algorithm is also visible to the proof checker under unit propagation. Crucially, using PB proofs does *not* mean that the solving algorithm is in any way a PB solver, nor does it need to employ any cutting planes reasoning to be able to write cutting planes proof steps. Instead, most solvers that write *VeriPB* proofs are conventional algorithms that have subsequently been augmented with, effectively, template-based print statements.

Although variations on this technique are suitable for various forms of backtracking search, including with backjumping and restarts, this framework does not extend to being able to cover dynamic programming algorithms, which have a very different notion of a search space. The remainder of this paper explores a different framework, where the structure of *VeriPB* proofs represent how dynamic programming algorithms run.

3 Proofs Involving States and Transitions

The key idea we will use for the proofs in this paper is to introduce an extension variable for each entry in a dynamic programming matrix, or for each node in a memoised recursive search tree or a top-down decision diagram construction. Each of these extension variables will reify the conjunction of several other extension variables, representing different parts of the state. We will then build up implication constraints between these extension variables that reflect the way entries in the matrix are derived, the recursive call structure, or the edges in the decision diagram. We will additionally build up a series of at-least-one constraints, demonstrating that the structure we have created is complete. We finish by using the at-least-one constraint over the final row of the matrix, or the final non-terminal layer of the decision diagram, to prove the conclusion.

So far, this idea is not unique to *VeriPB* proofs. The *DRAT* proof system also has an extension rule, and indeed Sinz and Biere [31], Jussila et al. [23] and Bryant [6] have constructed *DRAT* proofs for binary decision diagram solvers using extension variables in a similar but more restricted way. However, using *VeriPB* has many theoretical and practical benefits when we look at more complex problems. For example, counting problems like pigeonhole have direct proofs in *VeriPB* that scale trivially to arbitrarily large numbers of pigeons, and do not require decision diagram structures for some semblance of efficiency. Similarly, cutting planes allows us to work efficiently with reified integer linear inequalities without requiring complex and inefficient adder and multiplier circuits. *VeriPB* also supports optimisation problems, whereas the *DRAT* proof system only guarantees that satisfiable instances cannot be made unsatisfiable, and would not be sound if used for optimisation problems. Since we are looking to bring proof logging to a broader range of algorithms that solve problems far beyond the reach of SAT solving, we will work exclusively with *VeriPB*.

3.1 Knapsack as a Dynamic Programming Problem

We will first illustrate how to create proofs for simple 0/1 knapsack problems. We are given n items with weights $\boldsymbol{w}_i$ and profits $\boldsymbol{p}_i$, and we want to maximise profit whilst not taking items with a combined weight more than some constant W. For simplicity, we assume that all weights and profits are non-negative integers. We can express this as the PB problem

$$x_i \in \{0,1\} \qquad i \in \{1,\dots,n\} \tag{1}$$

$$\text{minimise} \qquad \sum_{i=1}^{n} -\boldsymbol{p}_i x_i \tag{2}$$

$$\text{subject to} \qquad \sum_{i=1}^{n} \boldsymbol{w}_i x_i \le W, \tag{3}$$

recalling the convention that PB problems have an objective function to be minimised rather than maximised. Note already that this PB representation is extremely straightforward, and does not involve constructing adder and multiplier circuits as it would if we used a CNF encoding.

This problem has a recursive formulation. Letting $P(i, w)$ be the maximum profit obtainable after taking the first i items whilst having weight w still available to use, we have the properties

$$P(0, w) = 0 \tag{4}$$

$$P(i, w) = \max\{ \tag{5}$$

$$\quad P(i-1, w), \tag{6}$$

$$\quad P(i-1, w-\boldsymbol{w}_i) + \boldsymbol{p}_i \text{ if } \boldsymbol{w}_i \leq w\}. \tag{7}$$

Here, Equation (4) gives the initial condition that there is zero profit from taking no items, regardless of weight; Equation (6) describes the option where we do not take item i; Equation (7) describes the option where we do take item i if we are allowed to; and the max operator in Equation (5) says that if we have two partial sums over the first i items both using weight $W - w$ then we need only consider the one which gives us the better profit.

This relation does not directly give us an algorithm. However, there are several standard ways of turning such a recurrence relationship into an algorithm, including dynamic programming via a matrix built iteratively over weights; using recursion with memoisation; or constructing a decision diagram layer by layer from the root downwards [22, 32]. From an algorithm implementation perspective, the choice of methods can be very important; however, for proof logging, the approach we describe works equally well for all three methods. The important points are simply that

1. the algorithm somehow avoids calculating the same partial sums twice;
2. not all partial sums of weights and profits are necessarily calculated; and
3. there is some way of handling "dominated" states, such as the maximum operation in Equation (5).

For ease of explanation, and because it allows the widest range of techniques to be demonstrated, we will assume a layer-by-layer construction, starting by considering whether or not we take the first item, and then building this up to decide what combination of the first two items we will take, and then the first three items, and so on. Within layer i, we will consider every possible partial sum of the first i weights that does not already exceed our bound W, and associate that with the maximum possible partial sum of profits using exactly that weight. We call this information a *state*, no matter whether it is implemented as a node in a decision diagram, a memoised function call, or an entry in a matrix. We call partial sums of either weights or profits *partial* states, and view the full state as being the conjunction of partial weight and profit states.

The idea behind our *VeriPB* proof is that we will introduce an extension variable $S^i_{w,p}$ for each state on layer i with partial sum of weights w and partial sum of profits p. For convenience, we will also introduce these variables for states that will be ignored due to the maximum rule. Recall that an extension variable is introduced by reifying a constraint; in our case, this constraint will be

$$S^i_{w,p} \Leftrightarrow W^i_w + P^i_p \geq 2 \tag{8}$$

where W^i_w and P^i_p are themselves also extension variables,

$$W^i_w \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{w}_j x_j \geq w \text{ and} \tag{9}$$

$$P^i_p \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{p}_j x_j \leq p. \tag{10}$$

In other words, $S^i_{w,p}$ is defined to be true if and only if the sum of the taken weights for the first i items is *at least* w, *and* the sum of the taken profits for the first i items is *at most* p. The reason for this choice of inequalities will become evident when we look at the maximum rule.

Merely introducing extension variables tells us nothing about which states could actually occur. The remainder of the proof consists of deriving implicational relationships between extension variables (which correspond to edges in a decision diagram), and then in proving that each layer is complete (that is, that we have an extension variable for every possible state that has not been eliminated).

The first set of implications that we derive correspond to deciding not to take item x_i. We in turn derive

$$W^{i-1}_w \wedge \overline{x}_i \Rightarrow W^i_w \qquad \text{using a cutting planes addition rule, and then} \tag{11}$$

$$P^{i-1}_p \wedge \overline{x}_i \Rightarrow P^i_p \qquad \text{similarly, and finally} \tag{12}$$

$$S^{i-1}_{w,p} \wedge \overline{x}_i \Rightarrow S^i_{w,p} \qquad \text{follows by RUP.} \tag{13}$$

For the base case, the first part of the conjunction is trivially true and is instead omitted, whilst for subsequent layers we will already have created the earlier extension variables, either due to the algorithm's layer-by-layer construction, or iteration, or recursion.

Next, suppose we *cannot* take item i due to the partial sum of weights exceeding W (recalling that for simplicity, we are forbidding negative weights). If this is the case, we derive

$$W^{i-1}_w \Rightarrow \overline{x}_i \qquad \text{using cutting planes and RUP, and then} \tag{14}$$

$$S^{i-1}_{w,p} \Rightarrow \overline{x}_i \qquad \text{and} \tag{15}$$

$$S^{i-1}_{w,p} \Rightarrow S^i_{w,p} \qquad \text{both follow by RUP.} \tag{16}$$

This cutting planes addition step is between the forward implication constraint defining W^{i-1}_w, and the constraint giving the bound on W that is part of the input axiom. Because none of the remaining weight coefficients are negative, a simple bounds consistency calculation shows that if we have used too much weight already by layer i then there is no way of assigning the remaining x_i variables that will bring our weight sum back to be no more than W.

Finally, suppose we *can* take item i. Letting $w' = w + \boldsymbol{w}_i$ and $p' = p + \boldsymbol{p}_i$ be our new weights and profits respectively, we instead derive

$$W^{i-1}_w \wedge x_i \Rightarrow W^i_{w'} \qquad \text{using cutting planes, and} \tag{17}$$

$$P^{i-1}_p \wedge x_i \Rightarrow P^i_{p'} \qquad \text{similarly, then} \tag{18}$$

$$S^{i-1}_{w,p} \wedge x_i \Rightarrow S^i_{w',p'} \qquad \text{follows by RUP, as does} \tag{19}$$

$$S^{i-1}_{w,p} \Rightarrow S^i_{w,p} + S^i_{w',p'} \geq 1. \tag{20}$$

Until this point, we have been ignoring the maximum rule. If we have two states on the same layer with the same w, and one with profit p and another with profit $p' > p$, we will derive that

$$S^i_{w,p} \Rightarrow S^i_{w,p'}. \tag{21}$$

What this implication means is, "if there is an assignment to the first i x_i variables where the weight sums to at least w and the profit to no more than p, then there is an assignment where the weight sums to at least w and the profit sums to no more than some larger profit p'". This is almost vacuous, and can easily be proved in cutting planes by unwrapping the conjunctions.

In fact, in our proofs we can also do this for a distinct pair of states $S^i_{w,p} \Rightarrow S^i_{w',p'}$ where $w' \leq w$ and $p' \geq p$; this can be detected efficiently in a layer-by-layer algorithm, but not so easily with other approaches.

Now we have described the relationship between states on the same and subsequent layers. The last part of the structure of our proof consists in deriving an at-least-one constraint over the final layer, asserting that our diagram is complete. Again, we make use of an inductive argument, by first deriving at-least-one constraints over the first layer, then the second layer, and so on. This is a simple sequence of resolution steps: given

$$\sum_{(w,p)\text{ on layer } i-1} S^{i-1}_{w,p} \geq 1 \tag{22}$$

we may resolve every variable on

$$\begin{aligned} &S^{i-1}_{w,p} \Rightarrow S^i_{w,p} && \text{from Equation (16), or} \\ &S^{i-1}_{w,p} \Rightarrow S^i_{w,p} + S^i_{w',p'} \geq 1 && \text{from Equation (20)} \end{aligned}$$

to derive the desired

$$\sum_{(w,p)\text{ on layer } i} S^{i}_{w,p} \geq 1. \tag{23}$$

This sets us up to provide a conclusion for our proof. Our algorithm execution will have solved the problem at this point, so we know an optimal assignment with profit $P^\star$ that we can use to obtain a solution-improving constraint $\sum_i -\boldsymbol{p}_i x_i \leq -P^\star - 1$. This in turn contradicts each component of Equation (23), showing unsatisfiability.

To bring this together, we illustrate one way of implementing a proof-logging knapsack solving algorithm in Algorithm 1. We stress, however, that the techniques we have described are not in any way tied to this particular algorithm design. In particular, the same proof framework can be used for matrix-based dynamic programming where each weight is considered in turn, as well as for recursion with memoisation. For a matrix, more states will be created, both in the solving algorithm and in the proof, whilst for recursion the states will be constructed in an order corresponding to the recursive search execution, rather than layer by layer. Similarly, although we chose to apply (a more general version of) the maximum rule as a single pass at the end of constructing each layer, we could instead derive the appropriate implication whenever the maximum rule is used.

Until this point, we have not discussed deletions. To save memory, matrix and decision diagram approaches to dynamic programming sometimes need only keep the current and previous layers (or columns). We can do this in our proof too: when we start building layer $i \geq 3$, we can tell the proof verifier that we promise we will no longer need to access any constraint and extension variable defined in layer $i-2$, and so these constraints may now be deleted. This will help the proof verifier use less memory, and can also speed up verification – proof steps using RUP or that introduce extension variables are not, strictly speaking, of constant complexity to verify in the worst case; we return to this in Section 4. With this caveat aside, the proofs we have written are efficient, in that we write effectively only a constant amount of data in the proof for each computation carried out by the algorithm.

3.2 A General Framework

In the same way that interleaving inference and backtrack constraints gives a general framework for proof logging for backtracking search algorithms, we are now in a position to describe how to generate proofs for dynamic programming and decision diagram algorithms. For a given problem and solving algorithm, we need to be able to do seven things.

Algorithm 1 One way of solving the knapsack problem, with proof logging, using a layer-by-layer decision diagram style construction.

```
$S^0 \leftarrow \{S^0_{0,0}\}$
for $i \leftarrow 1 \ldots n$ do                                          // i.e. for each layer in turn
    for all $S^i_{w,p} \in S^{i-1}$ do                                     // i.e. for each state in the previous layer
        Extend $W^i_w \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{w}_j x_j \geq w$, $P^i_p \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{p}_j x_j \leq p$, and then
            $S^i_{w,p} \Leftrightarrow W^i_w \wedge P^i_p$ if they do not already exist
        // Consider not taking item i
        $S^i \leftarrow S^i \cup \{S^i_{w,p}\}$
        Derive $W^{i-1}_w \wedge \overline{x}_i \Rightarrow W^i_w$ and $P^{i-1}_p \wedge \overline{x}_i \Rightarrow P^i_p$ by cutting planes addition, then
            $S^{i-1}_{w,p} \wedge \overline{x}_i \Rightarrow S^i_{w,p}$ by RUP
        // Now see whether we could take item i
        if $w + \boldsymbol{w}_i > W$ then                                 // We cannot take item i
            Derive $W^{i-1}_w \Rightarrow \overline{x}_i$ by addition, then $S^{i-1} \Rightarrow \overline{x}_i$ and $S^{i-1}_{w,p} \Rightarrow S^i_{w,p}$ by RUP
        else                                                              // We could take item i
            Let $(w', p') = (w + \boldsymbol{w}_i, p + \boldsymbol{p}_i)$
            Extend $W^i_{w'} \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{w}_j x_j \geq w'$, $P^i_{p'} \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{p}_j x_j \leq p'$, and then
                $S^i_{w',p'} \Leftrightarrow W^i_{w'} \wedge P^i_{p'}$ if they do not already exist
            $S^i \leftarrow S^i \cup \{S^i_{w',p'}\}$
            Derive $W^{i-1}_w \wedge x_i \Rightarrow W^i_{w'}$ and $P^{i-1}_p \wedge x_i \Rightarrow P^i_{p'}$ by addition, then
                $S^{i-1}_{w,p} \wedge x_i \Rightarrow S^i_{w',p'}$ and $S^{i-1}_{w,p} \Rightarrow S^i_{w,p} \vee S^i_{w',p'}$ by RUP
    for all $S^i_{w,p} \in S^i$ that is dominated by some other $S^i_{w',p'}$ do
        Derive $S^i_{w,p} \Rightarrow S^i_{w',p'}$ by unwrapping
        $S^i \leftarrow S^i \setminus \{S^i_{w,p}\}$
    Derive $\sum S^i \geq 1$ by resolving on each variable in $\sum S^{i-1} \geq 1$
    Delete every constraint created on layer $S^{i-1}$

if $S^n$ is empty then
    Conclude infeasibility
else
    Log how we obtain the state with the best profit
    Derive that every $S^n_{w,p}$ contradicts the solution-improving constraint
    Conclude optimality
```

1. Represent the problem as a set of PB inequalities and a PB objective to minimise.
2. Generate an extension variable for each new state, as it is encountered (whether that state is a node, a matrix entry, or a memoised recursive call). This is also done for infeasible states.
3. Generate an implication constraint $S' \wedge c \Rightarrow S$ linking each new state S to its predecessor S', showing that if we were in state S' and we choose a given condition c, then we arrive at this new state.
4. For any state S that is infeasible, generate a proof $S \Rightarrow \bot$ that being in this state implies contradiction. (In practice, this can sometimes be combined into the previous step instead, as we did in Equation (16).)
5. For any state S that is dominated, subsumed, or similar by a better state S', generate a proof that $S \Rightarrow S'$.
6. Show that we have considered every feasible state on a layer, or generated a complete column in a matrix, by creating an at-least-one constraint over the extension variables.
7. Derive a conclusion using the at-least-one constraint over the final layer or column.

The first requirement is generally straightforward, since the representation only needs to be correct, not useful for solving purposes. However, note that this means that our starting point is a problem, not an algorithm or a recurrence relation for solving that problem: we are certifying solutions that are found using dynamic programming, rather than specifically certifying the execution of a dynamic program. Ideally, this representation step should generally be carried out independently of how we then decide to go on and find a solution.

For the second requirement, we need to ask what kinds of state can be represented using extension variables in a *VeriPB* proof. For knapsack, the states represented a conjunction of pseudo-Boolean inequalities. However, this technique is much more general. For example, Bergman et al. [2] give an example of a decision diagram solver where states represent sets of vertices from a graph: these can be represented as conjunctions of Boolean variables, using a pair of reified inequalities to express a reified equality constraint. Similarly, we can reuse the encoding described by Gocht et al. [17] to represent anything that could be described in constraint programming terms using integer variables. It is not so obvious how to represent rational or real numbers in *VeriPB*, although in some circumstances these could be handled by scaling.

For the third requirement, if our conditions and states correspond cleanly to sets of Boolean variables then this is trivial: we are simply extending a set of inequalities by adding in additional fixed variables. For the fourth requirement, this may also be trivial, or we may need to reuse the constraint programming techniques of Gocht et al. [17] to show that a given partial state is infeasible. The sixth requirement needs only that we can show that we have indeed considered every possibility moving between layers or columns – for Boolean variables, this is immediate, whilst for encoded integer variables we can make use of the at-least-one constraint over each option. The seventh requirement comes down to showing that, given an optimal full state S and a suboptimal full state S', S' does not beat S – this should follow naturally from the objective function. For each of these requirements, we rely heavily upon the ability to cleanly wrap and unwrap reified constraints, and to reason as if reifications were not present using the technique described in Theorem 1 in the appendix. It is worth stressing that these properties, and the resulting ease of producing this kind of proof, are a specific characteristic of extended cutting planes, and they do not hold for many other proof systems.

This leaves the fifth requirement, being able to reason about dominated states. This potentially requires more creativity – and this should not be surprising, since alongside tracking states, merging states is the other feature which distinguishes dynamic programming style algorithms from backtracking search. Fortunately, the *VeriPB* proof system provides us with a suite of tools for these scenarios. In many cases, fusion resolution under implications (which, given $s \land r \Rightarrow \sum_i a_i x_i \geq A$ and $s \land \overline{r} \Rightarrow \sum_i a_i x_i \geq A'$ lets us infer that $s \Rightarrow \sum_i a_i x_i \geq \min(A, A')$ by resolving away the r) is sufficient, but *VeriPB*'s strengthening rule also allows sophisticated symmetry and dominance arguments [4].

At least so long as we are working with Booleans and integers, we have found this framework to be powerful enough for a wide range of problems. For example, weighted interval scheduling problems [25] have a natural recursive formulation using a maximum operation and sums, and dynamic programming gives a polynomial time solving algorithm. Proof logging for this problem is simpler than knapsack: the states are a simple sum, rather than a conjunction of sums.

Or, suppose we want to find the longest path in a directed acyclic graph. This also has a simple dynamic programming formulation, where nodes are visited in topological order. The longest path ending at a given node is then calculated by looking at each predecessor node and adding its longest path cost to the cost of its edge to our given node, and taking the maximum of these costs. In this case, our proof would use the costs as state variables, and rather than having two options at each transition, would be selecting between one option per incoming edge on the node. Note also that the proof process implicitly checks the correctness of the topological sort: if either the implementation were faulty, or the concept mathematically flawed (e.g. if we tried to do this in a graph with cycles), then the proof process would fail.

Of course, this does not mean that we can provide efficient proof logging for every dynamic programming or decision diagram algorithm that might ever be invented, just as it would not be reasonable to claim that efficient proof logging is definitely possible for every single backtracking search algorithm – for example, we do not yet know whether it is practically feasible to reason about real or floating point numbers in *VeriPB*. Nor does this automate the process of adding proof logging to a solver. However, in the same way that the framework of interleaving RUP backtracking steps with explicit derivations for reasoning has vastly simplified adding proof logging to a wide range of search algorithms, we can say that these techniques will vastly reduce the conceptual and implementation hurdles required to use proof logging for state- and transition-based algorithms.

3.3 Knapsack as a Constraint

We return now to knapsack, but in a more general setting. As well as being an interesting stand-alone problem, knapsack appears as a constraint in some constraint programming toolkits. Trick [34] describes a propagator for a single 0/1 integer linear inequality where the sum is a variable, whilst Fahle and Sellmann [11], Sellmann [30], Katriel et al. [24], Malitsky et al. [27], and Malitsky et al. [26] work on exactly two integer linear equalities that sum to two different variables, and do not restrict to 0/1 variables for the items. MiniZinc also defines the constraint this way [33], whilst XCSP^3 [5] allows for more than two inequalities. In all cases, the multiplier vector(s) are integer constants – sometimes these are required to be non-negative.

Propagators based upon Trick's approach can achieve either bounds or domain consistency on the sum variables, as well as domain consistency on the item variables. This is done by building a decision diagram, and then, by working from the final layer and moving backwards, deleting any nodes and edges that do not lead to a feasible state; what remains is a diagram where every path from the first layer to the final layer corresponds to a solution to the constraint. Once this is built, on some layers there may only be edges corresponding to the layer's item being accepted, or only edges corresponding to the layer's item being rejected; in this case, the associated item variable is forced.

Gocht et al. [17] described a framework for proof logging for constraint programming solvers using *VeriPB*. This framework supports integer variables, and a number of global constraints, including integer linear inequalities. To add a new constraint propagator to this framework, we must have two things. Firstly, we must be able to express the semantics of the constraint in PB form – this is trivial, because integer linear inequalities are already supported. Secondly, we must have a way of justifying all reasoning that can be carried out by its propagator. This will follow a similar pattern to proof logging for a standalone knapsack solver, but with different states and a more complicated conclusion.

For a standalone knapsack solver, recall that our states $S^i_{w,p}$ represented that the partial sum of the first i items has weight at least w, and profit at most p. For a constraint, we instead want to track states that have weight exactly w, and profit exactly p. To do this, we can introduce the four extension variables

$$W{\uparrow}^i_w \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{w}_j x_j \geq w \qquad W{\downarrow}^i_w \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{w}_j x_j \leq w \tag{24}$$

$$P{\uparrow}^i_p \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{p}_j x_j \geq p \qquad P{\downarrow}^i_p \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{p}_j x_j \leq p \tag{25}$$

which allow us to define

$$S^i_{w,p} \Leftrightarrow W{\uparrow}^i_w + W{\downarrow}^i_w + P{\uparrow}^i_p + P{\downarrow}^i_p \geq 4. \tag{26}$$

When building the structure of the proof, there are five differences.

1. We must construct implications for all four partial states, rather than just two.
2. We must bear in mind that we might be inside a backtracking search, and so some of the information we have about variables might be conditional. Fortunately this is not a concern: recall that any RUP or cutting planes proof can trivially and efficiently be extended to operate under assumptions.
3. We might be dealing with constraint programming variables whose domains are not 0/1. This means there may be more than two edges coming out of a state. To derive the implications for partial sums, we follow Gocht et al.'s approach of introducing direct variables as required, and then we use an additional cutting planes multiplication operation. We must also take care when deriving the at-least-one constraint over each layer, because this relies upon exhaustively branching. Again, this is dealt with by Gocht et al.'s framework, which allows us to obtain an at-most-one constraint for any constraint programming variable's values.
4. We may now only merge states with exact matches on weights and profits. This is true both algorithmically and in proof terms – reassuringly, if we were to forget this condition when implementing the propagation algorithm, we would quickly find it impossible to construct the appropriate implication steps in the proof.
5. We cannot delete intermediate layers as we go: we want to reason about the diagram as a whole, so it stands to reason that the structure of the diagram must remain in the proof. However, we can delete every intermediate constraint once the conclusions are derived.

Rather than establishing a proof of optimality, a knapsack propagator's proof aims to show lack of support for some variables' values. By looking at the possible weights and profits on the final layer of the decision diagram, we can recognise that either some bounds or some specific values are unsupported by the constraint; we can derive these facts inside a proof by resolving over the at-least-one constraint on the final layer. This gives us either bounds or domain consistency on the sum variables, as we prefer.

The backwards pass, which shows lack of support on the item values, is also straightforward – since our propagation algorithm works backwards from the final layer, eliminating infeasible nodes, it is sufficient to use RUP steps to show that the corresponding states must be false. Once this has been done, eliminating values from item variables also follows by RUP. This closely resembles the steps used by McIlree and McCreesh [29] to generate proofs from propagations for the regular language membership constraint.

4 Implementations and Evaluation

Before presenting the results of our empirical evaluation, it is important to ask what the purpose of such an evaluation should be. Rather than trying to implement the world's fastest dynamic programming algorithms or propagators, or even to tell you when to use these

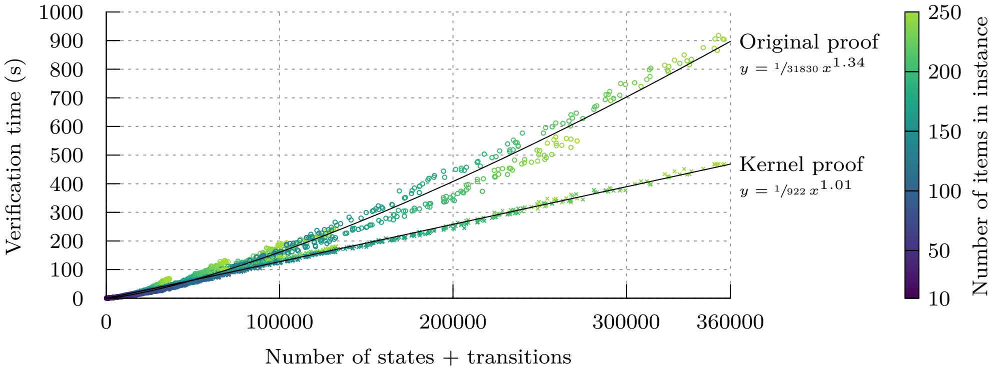


Figure 1 Verification times for knapsack problem instances with between 10 and 250 items (shown using colour). The power law fit lines show the original proof and the rewritten kernel proof times, plotted against the number of states plus transitions required to solve the instance.

techniques, the main aim of this paper is to demonstrate that *if* you choose to use these techniques, then certifying correctness using pseudo-Boolean proof logging is viable. To show this, we have implemented[1] stand-alone solvers for three problems: knapsack, longest path in a directed acyclic graph, and interval scheduling. For knapsack, we implemented both top-down and matrix-based algorithms, whilst for the other two problems we used only a matrix. With the aim of the paper in mind, our key measure of success from these implementations is that we were able to add proof logging to each solver simply by adding in statements to log information that was already present, without needing to extend or change the underlying algorithm. To validate our implementations, we tested them on a large number of randomly generated instances and were able to verify every proof produced.

Our proofs in each case are generated *efficiently*, having cost and length roughly linear in the amount of work done by the solver. However, the constant factor slowdown needed to write these proofs to disk is potentially large. Creating a new entry in a dynamic programming table for a problem such as knapsack can be extremely fast, requiring only a few additions, comparisons, and memory accesses. However, to justify an entry and the transition leading to it, we need to write several lines of text to a file. For an efficiently implemented algorithm, this can easily lead to more than an order of magnitude slowdown. This is much worse than for, e.g. SAT solving, because a CDCL solver does much more computation per proof step than a simple knapsack algorithm.

But what about proof verification time – is that also roughly linear in proof size? This turns out to be a more complex question. When using only explicit cutting planes derivations, we would expect the cost of verifying each proof step to depend only upon the number of operations. However, verifying reverse unit propagation or strengthening steps requires achieving bounds consistency over the active set of inequalities, which is not a constant-time operation. In the top line of Figure 1 we show the verification times required for 1,200 randomly generated knapsack problem instances with between 10 and 250 items, with random weights and profits both between 1 and 10, and a maximum weight of between 50 and 1000, solved using the top-down approach. (These parameters were selected to give instances where dynamic programming is a good choice of solving technique, so that we can measure the scalability of proof verification: we are trying to challenge the proof verifier, not the solver.)

[1] `https://doi.org/10.5281/zenodo.12574620`

We measure verification time as a function of the number of states plus transitions required to solve each instance, since this is in effect "the amount of work" the solver took to solve an instance. The fit line suggests that verification scales worse than linearly, but better than quadratically.

Similarly to how DRAT proofs can be converted to LRAT proofs, *VeriPB* is able to rewrite proofs into a simplified "kernel format" that does not require any propagations to verify: reverse unit propagation steps are rewritten to cutting planes derivations, and strengthening rule applications are also given explicit cutting planes subproofs for each proof goal [15]. Carrying out this simplification is not computationally more expensive than verifying the proof, and introduces only a small additional slowdown for outputting the rewritten proof to disk. In Figure 1 we also plot the time taken to verify these rewritten proofs, achieving the lower line. Now, the power law fit line suggests that verification time scales extremely close to linearly with proof size, with a verification rate of a little below a thousand states and transitions per second (which we expect to vary considerably based upon hardware and disk speeds). In principle, solvers could output these kernel proofs directly, avoiding the need for proof rewriting if an important concern is the initial proof verification time; however, this would require considerably more work from solver authors.

Finally, we have also implemented the knapsack constraint inside the Glasgow Constraint Solver, using a top-down construction. Our implementation supports arbitrarily many simultaneous inequalities, and is not restricted to 0/1 variables. It achieves domain consistency on every variable. Again, we were able to do this without having to restrict or alter the underlying propagation algorithm: *VeriPB* proofs are powerful enough to conveniently express the reasoning we wanted to carry out, and we did not have to design an algorithm specifically to make proof logging possible. To validate the implementation, we used the same system as other constraints in the Glasgow Subgraph Solver, where curated and randomly generated test data is combined with proof checking inside a continuous integration framework; we have successfully verified thousands of proofs in this manner. In terms of performance, any measurements are extremely sensitive to disk write speeds and to details of implementation, to the extent that using shorter variable names inside proofs can have a significant effect upon running times. However, to give indicative figures, verifying knapsack propagation proofs is typically between twenty and fifty times more expensive than producing them; this is somewhat more expensive than for some other propagators [17, 29], likely due to the large number of extension variables used in the proofs.

5 Conclusion

We have shown that the *VeriPB* proof system supports convenient and efficient proofs for a range of dynamic programming algorithms, and that it can do so regardless of whether the algorithms use a matrix, recursion and memoisation, or a top-down construction, and even when we are inside a dynamic programming propagator in a constraint programming toolkit. We saw that the cutting planes proof system makes it both natural and efficient to reason about reified linear inequalities, whilst extension variables give us the power to describe the logical relationships between states.

The knapsack propagation example showed how different conclusions could be inferred, depending upon how states were represented: when solving the knapsack problem directly, we tracked less information, thus allowing more states to be merged, whilst for constraint propagation our states were more expressive. This example could be extended further, e.g. to relaxed and restricted decision diagrams, where we are allowed to violate some constraints and only achieve a lower or upper bound rather than an exact solution. In such a setting, our ability to compose proofs and to run proofs conditional upon assumptions or guesses would be very helpful, since modern decision diagram based solvers can construct many decision diagrams during the solving process.

An interesting open question is how to extend this work to cover problems where we want to count solutions, rather than finding an optimal solution. Once a decision diagram or dynamic programming matrix has been constructed, solution counts are often easily accessible. However, this property does not immediately transfer through to proofs. In the same way that DRAT proofs can only be used to reason "without loss of satisfaction", *VeriPB* proofs establish "without loss of optimality". This means that solutions can be removed, so long it can be shown that another equally-good-or-better solution exists (for example, through symmetry or dominance breaking). We believe it is important to give solver authors the ability to write proofs that correspond precisely to the real-world problem being solved. As such, we would like to see an appropriate theoretical foundation that will allow solvers to produce proofs either for optimality reasoning or for counting, with only minimal changes that reflect the algorithmic differences needed in the two settings. We would also be interested to know whether *VeriPB* can reasonably be used to work with rational or real numbers, either by scaling or more advanced techniques.

References

1 Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided MaxSAT solving. In Brigitte Pientka and Cesare Tinelli, editors, *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2023. doi:10.1007/978-3-031-38499-8_1.

2 David Bergman, André A. Ciré, Ashish Sabharwal, Horst Samulowitz, Vijay A. Saraswat, and Willem Jan van Hoeve. Parallel combinatorial optimization with decision diagrams. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, volume 8451 of *Lecture Notes in Computer Science*, pages 351–367. Springer, 2014. doi:10.1007/978-3-319-07046-9_25.

3 David Bergman, André A. Ciré, Willem-Jan van Hoeve, and John N. Hooker. *Decision Diagrams for Optimization*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2016. doi:10.1007/978-3-319-42849-9.

4 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.

5 Frédéric Boussemart, Christophe Lecoutre, and Cédric Piette. XCSP3: an integrated format for benchmarking combinatorial constrained problems. *CoRR*, abs/1611.03398, 2016. arXiv:1611.03398.

6 Randal E. Bryant. Tbuddy: A proof-generating BDD package. In Alberto Griggio and Neha Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 49–58. IEEE, 2022. doi:10.34727/2022/ISBN.978-3-85448-053-2_10.

7 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.

8 Kevin K. H. Cheung, Ambros M. Gleixner, and Daniel E. Steffy. Verifying integer programming results. In Friedrich Eisenbrand and Jochen Könemann, editors, *Integer Programming and Combinatorial Optimization - 19th International Conference, IPCO 2017, Waterloo, ON, Canada, June 26-28, 2017, Proceedings*, volume 10328 of *Lecture Notes in Computer Science*, pages 148–160. Springer, 2017. doi:10.1007/978-3-319-59250-3_13.

9 Chiu Wo Choi, Warwick Harvey, J. H. M. Lee, and Peter J. Stuckey. Finite domain bounds consistency revisited. In *AI 2006: Advances in Artificial Intelligence, 19th Australian Joint Conference on Artificial Intelligence, Hobart, Australia, December 4-8, 2006, Proceedings*, pages 49–58, 2006. doi:10.1007/11941439_9.

10 Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017. doi:10.1007/978-3-319-63046-5_14.

11 Torsten Fahle and Meinolf Sellmann. Cost based filtering for the constrained knapsack problem. *Ann. Oper. Res.*, 115(1-4):73–93, 2002. doi:10.1023/A:1021193019522.

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

13 Stephan Gocht. *Certifying Correctness for Combinatorial Algorithms: by Using Pseudo-Boolean Reasoning*. PhD thesis, Lund University, Sweden, 2022. URL: https://lup.lub.lu.se/record/3550cb96-83d5-4fc7-9e62-190083a3c10a.

14 Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, 2020. doi:10.1007/978-3-030-58475-7_20.

15 Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph 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 8038–8047. AAAI Press, 2024. doi:10.1609/AAAI.V38I8.28642.

16 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1134–1140. ijcai.org, 2020. doi:10.24963/ijcai.2020/158.

17 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. 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 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CP.2022.25.

18 Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-boolean proofs. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3768–3777. AAAI Press, 2021. doi:10.1609/AAAI.V35I5.16494.

19 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013. URL: https://ieeexplore.ieee.org/document/6679408/.

20 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2013. doi:10.1007/978-3-642-38574-2_24.

21 John N. Hooker. Generalized resolution for 0-1 linear inequalities. *Ann. Math. Artif. Intell.*, 6(1-3):271–286, 1992. doi:10.1007/BF01531033.

22 John N. Hooker. Decision diagrams and dynamic programming. In Carla P. Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 94–110. Springer, 2013. doi:10.1007/978-3-642-38171-3_7.

23 Toni Jussila, Carsten Sinz, and Armin Biere. Extended resolution proofs for symbolic SAT solving with quantification. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 54–60. Springer, 2006. doi:10.1007/11814948_8.

24 Irit Katriel, Meinolf Sellmann, Eli Upfal, and Pascal Van Hentenryck. Propagating knapsack constraints in sublinear time. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 231–236. AAAI Press, 2007. URL: http://www.aaai.org/Library/AAAI/2007/aaai07-035.php.

25 Antoon W.J. Kolen, Jan Karel Lenstra, Christos H. Papadimitriou, and Frits C.R. Spieksma. Interval scheduling: A survey. *Naval Research Logistics (NRL)*, 54(5):530–543, 2007. doi:10.1002/nav.20231.

26 Yuri Malitsky, Meinolf Sellmann, and Radoslaw Szymanek. Filtering bounded knapsack constraints in expected sublinear time. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, pages 141–146. AAAI Press, 2010. doi:10.1609/AAAI.V24I1.7560.

27 Yuri Malitsky, Meinolf Sellmann, and Willem Jan van Hoeve. Length-lex bounds consistency for knapsack constraints. 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 266–281. Springer, 2008. doi:10.1007/978-3-540-85958-1_18.

28 Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Comput. Sci. Rev.*, 5(2):119–161, 2011. doi:10.1016/J.COSREV.2010.09.009.

29 Matthew J. McIlree and Ciaran McCreesh. Proof logging for smart extensional constraints. 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 26:1–26:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CP.2023.26.

30 Meinolf Sellmann. Approximated consistency for knapsack constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 679–693. Springer, 2003. doi:10.1007/978-3-540-45193-8_46.

31 Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining bdds. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *Computer Science - Theory and Applications, First International Symposium on Computer Science in Russia, CSR 2006, St. Petersburg, Russia, June 8-12, 2006, Proceedings*, volume 3967 of *Lecture Notes in Computer Science*, pages 600–611. Springer, 2006. doi:10.1007/11753728_60.

32 Steven Skiena. *The Algorithm Design Manual, Third Edition.* Texts in Computer Science. Springer, 2020. doi:10.1007/978-3-030-54256-6.

33 Peter J. Stuckey, Kim Marriott, and Guido Tack. The MiniZinc handbook section 4.2.1: Global constraints, 2023. URL: `https://www.minizinc.org/doc-2.5.3/en/lib-globals.html`.

34 Michael A. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Ann. Oper. Res.*, 118(1-4):73–84, 2003. doi:10.1023/A:1021801522545.

35 Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. doi:10.1007/978-3-319-09284-3_31.

A Proofs Under Implications

In various pseudo-Boolean (PB) proof logging projects, it has been useful to rely on the assumption that if we have an efficient proof procedure for deriving a constraint D from a set of constraints F, then we can convert this into an efficient procedure for deriving $R \Rightarrow D$ from the set of constraints $\{R \Rightarrow C : C \in F\}$ for some conjunction of literals R. In this appendix we formalise and generalise this property, showing that efficient cutting-planes proofs can be "unrestricted" to construct analogous efficient proofs where the premises and conclusion are subject to (potentially different) conditions using reification.

A.1 Notation

A *(partial) assignment* is a (partial) function from variables to $\{0, 1\}$; we extend an assignment ρ from variables to literals in the natural way by respecting the meaning of negation, and for literals ℓ over variables x not in the domain of ρ, denoted $x \notin \mathrm{dom}(\rho)$, we use the convention $\rho(\ell) = \ell$. For notational convenience, we can also view ρ as the set of literals $\{\ell : \rho(\ell) = 1\}$ assigned true by ρ. Applying ρ to a constraint $C = \sum_i a_i \ell_i \geq K$ yields

$$C\restriction_\rho \doteq \sum_{\ell_i : \rho(\ell_i) = \ell_i} a_i \ell_i \geq K - \sum_{\ell_j \in \rho(\ell_j) = 1} a_j \tag{27}$$

substituting literals as specified by ρ. We extend this notation to applying assignments to F in the natural way $F\restriction_\rho = \bigcup_{C \in F} C\restriction_\rho$.

We will write $\mathit{Vars}(C)$, $\mathit{Vars}(F)$, $\mathit{Lits}(C)$ and $\mathit{Lits}(F)$ to denote the sets of variables or literals appearing in a PB constraint C or formula F.

A.2 Constructing Proofs Under Implications

We can now state our main result in its general form.

▶ **Theorem 1.** *Let F be a PB formula over n variables, ρ be a partial assignment, and suppose that from $F\restriction_\rho$ we can derive a constraint D using a cutting planes and RUP derivation of length L. Then we can construct a derivation of length $\mathrm{O}(n \cdot L)$ from F of the constraint*

$$\bigwedge_{\ell \in \rho} \ell \Rightarrow D. \tag{28}$$

In what follows, we assume all constraints are normalised. We will first show the following.

▶ **Lemma 2.** *For any PB constraint C and partial assignment ρ, we can always derive $\bigwedge_{\ell \in \rho} \ell \Rightarrow C\restriction_\rho$ from C using a cutting planes derivation of length $\mathrm{O}(|\mathit{Vars}(C)|)$.*

Proof. First, let us write C as

$$\sum_{\substack{\ell_i \in \mathit{Lits}(C):\\ \rho(\ell_i)=\ell}} a_i\ell_i + \sum_{\substack{\ell_j \in \mathit{Lits}(C):\\ \rho(\ell_j)=1}} b_j\ell_j + \sum_{\substack{\ell_k \in \mathit{Lits}(C):\\ \rho(\ell_k)=\ell}} c_k\ell_k \geq K. \tag{29}$$

Then, if we let $B = \sum_{\substack{\ell_j \in \mathit{Lits}(C):\\ \rho(\ell_j)=1}} b_j$, we note that $C{\restriction_\rho}$ is the constraint

$$\sum_{\substack{\ell_i \in \mathit{Lits}(C):\\ \rho(\ell_i)=\ell}} a_i\ell_i \geq K - B \tag{30}$$

and $\bigwedge_{\ell\in\rho} \ell \Rightarrow C{\restriction_\rho}$ is the constraint

$$\sum_{\substack{\ell_j \in \mathit{Lits}(C):\\ \rho(\ell_j)=1}} (K-B)\ell_j + \sum_{\substack{\ell_k \in \mathit{Lits}(C):\\ \rho(\ell_k)=0}} (K-B)\ell_k + \sum_{\substack{\ell_i \in \mathit{Lits}(C):\\ \rho(\ell_i)=\ell}} a_i\ell_i \geq K - B \tag{31}$$

To derive Equation (31) from Equation (29) we can proceed as follows.

1. For all j, add the literal axioms amounting to $b_j\bar{\ell}_j \geq 0$ to Equation (29) yielding

$$\sum_{\substack{\ell_k \in \mathit{Lits}(C):\\ \rho(\ell_k)=\ell}} c_k\ell_k + \sum_{\substack{\ell_i \in \mathit{Lits}(C):\\ \rho(\ell_i)=\ell}} a_i\ell_i \geq K - B \tag{32}$$

2. Saturate to ensure that for all k, $c_k \leq K - B$.
3. Add literal axioms $\ell_k \geq 0$ and $\bar{\ell}_j \geq 0$ as needed to obtain Equation (31).

This amounts to at most one weakening step per variable appearing in C, along with one saturation step, and hence has length $\mathrm{O}(|\mathit{Vars}(C)|)$. ◀

We are now able to prove the main result.

Proof. Let $\pi = (D_1, \ldots, D_L = D)$ be the derivation of D from $F{\restriction_\rho}$, and denote by π_s the set $\{D_1, \ldots, D_{s-1}\}$ of constraints prior to derivation step s. Each D_s is one of the following:

- An axiom (constraint in $F{\restriction_\rho}$).
- A literal axiom.
- The result of a cutting planes operation, with antecedents in π_s.
- A RUP constraint with respect to $F{\restriction_\rho} \cup \pi_s$.

We will proceed by structural induction on π and show that for any D_s we can construct a length $O(n \cdot s)$ derivation that $\bigwedge_{\ell\in\rho} \ell \Rightarrow D_s$ from F.

For the base cases, we consider an axiom $D_a \in F{\restriction_\rho}$. We must have some constraint $C \in F$ such that $C{\restriction_\rho} = D_a$. Hence we can derive C as an axiom, and then by Lemma 2 we can derive $\bigwedge_{\ell\in\rho} \ell \Rightarrow C{\restriction_\rho}$, i.e. $\bigwedge_{\ell\in\rho} \ell \Rightarrow D_a$, in $O(|\mathit{Vars}(C)|) \subseteq O(n)$ steps. Note that if D_a is instead a literal axiom then $\bigwedge_{\ell\in\rho} \ell \Rightarrow D_a$ is also a literal axiom, because the reification coefficients will all be zero.

Now assume for any non-axiom constraint D_s we have already constructed a derivation of length $O(n \cdot (s-1))$ deriving all the constraints in $\pi'_s = \left\{\bigwedge_{\ell\in\rho} \ell \Rightarrow D_i : D_i \in \pi_s\right\}$. We now consider different cases depending on how D_s was derived in π.

Case 1: D_s is the result of adding two constraints $D_i, D_j \in \pi_s$.

Then by assumption $\bigwedge_{\ell\in\rho} \ell \Rightarrow D_i$, and $\bigwedge_{\ell\in\rho} \ell \Rightarrow D_j$ have already been derived. If we let K_i and K_j be the degrees of D_i and D_j respectively, we can write these in the form

$$\sum_{\ell\in\rho} K_i\bar{\ell} + D_i \tag{33}$$

and

$$\sum_{\ell \in \rho} K_j \bar{\ell} + D_j, \tag{34}$$

and so adding these together yields

$$\sum_{\ell \in \rho} (K_i + K_j)\bar{\ell} + D_s. \tag{35}$$

If K_s is the degree of D_s, note that we must have $K_s \leq K_i + K_j$, since cancellation of matching literals when adding D_i and D_j can only reduce the degree of their sum. Hence if we apply saturation to Equation (35) we obtain $\sum_{\ell \in \rho} K_s \bar{\ell} + D_s$, i.e. $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$, as required.

Case 2: D_s is result of multiplying a constraint $D_i \in \pi_s$ by a scalar λ.
Then by assumption $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_i$ has already been derived, and again we can write this as

$$\sum_{\ell \in \rho} K_i \bar{\ell} + D_i \tag{36}$$

where K_i is the degree of K_i. If we multiply this by λ we obtain

$$\sum_{\ell \in \rho} \lambda K_i \bar{\ell} + \lambda D_i \tag{37}$$

which is precisely $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$, as required.

Case 3: D_s is the result of dividing a constraint $D_i \in \pi_s$ by a scalar λ.
Then again by assumption $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_i$ has already been derived, and this time we will write this in full as

$$\sum_{\ell \in \rho} K_i \bar{\ell} + \sum_j a_j \ell_j \geq K_i. \tag{38}$$

If we divide this by λ we obtain

$$\sum_{\ell \in \rho} \lceil (K_i/\lambda) \rceil \bar{\ell} + \sum_j \lceil a_j/\lambda \rceil \ell_j \geq \lceil (K_i/\lambda) \rceil, \tag{39}$$

which is precisely $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$, as required.

Case 4: D_s is the result of applying saturation to a constraint $D_i \in \pi_s$.
Once again by assumption $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_i$ has already been derived, and we can write this in full as above in Equation (38). After applying saturation to this we obtain

$$\sum_{\ell \in \rho} \min(K_i, K_i)\bar{\ell} + \sum_j \min(a_j, K_i)\ell_j \geq K_i. \tag{40}$$

which is precisely $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$, as required.

Case 5: D_s is the result of applying weakening (adding literal axioms) to a constraint $D_i \in \pi_s$.
In this case we can view the added literal axioms as another degree-0 constraint D_j, which we can always derive, and so the fact we can obtain $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$ follows immediately from Case 1.

Case 6: D_s is a RUP constraint.

Write $D_s = \sum_i a_i \ell_i \geq K$ and let $A = \sum_i a_i$. Then $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$ is the constraint

$$\sum_{\ell \in \rho} K\bar{\ell} + \sum_i a_i \ell_i \geq K, \tag{41}$$

and its negation is

$$\sum_{\ell \in \rho} K\ell + \sum_i a_i \bar{\ell}_i \geq A + 1 + (|\rho| - 1)K. \tag{42}$$

We can see that for Equation (42) to be satisfied, all the reification literals $\ell \in \rho$ must be set to true. Recalling that all constraints in $\pi'_s = \left\{\bigwedge_{\ell \in \rho} \ell \Rightarrow D_i : D_i \in \pi_s\right\}$ are all assumed to have been previously derived, we can see that performing unit propagation will reduce constraints in $F \cup \pi'_s \cup \neg(\bigwedge_{\ell \in \rho} \ell \Rightarrow D)$ to be precisely the constraints in $F{\restriction_\rho} \cup \pi_s \cup \neg D$. Since by assumption deriving D_s from $F{\restriction_\rho} \cup \pi_s$ by RUP was a legitimate derivation step, continued unit propagation on the constraint database must result in a contradiction. Hence we can derive $\bigwedge_{\ell \in \rho} \ell \Rightarrow D$ from $F \cup \pi'_s$ as a single RUP step.

In all of these cases, we only need a constant number (at most two) proof steps, to derive $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$, from what was assumed to already be derived, and so by starting from the axioms and applying induction we can construct a derivation which includes all of the constraints in $\pi'_L = \left\{\bigwedge_{\ell \in \rho} \ell \Rightarrow D_i : D_i \in \pi\right\}$ and in particular our desired $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_L$.

Since each of the L constraints in π'_L requires at most $O(n)$ intermediate derivation steps, our constructed derivation has length at most $O(n \cdot L)$. ◀

With Theorem 1 established we easily obtain the following useful corollary.

▶ **Corollary 3.** *Let F be a PB formula over n variables and let R be a set of literals over distinct variables not appearing in F (i.e. for any $\ell \in R$, $\bar{\ell} \notin R$ and $\ell \notin \mathit{Lits}(F)$). Then let $R(F)$ be a set of reified constraints $\{R_C \Rightarrow C : C \in F\}$, where each reifying term R_C is a conjunction of literals in R.*

Then, if we can derive a constraint D from F using a cutting planes and RUP derivation of length L, we can construct a derivation of length $O(L \cdot n)$ of the constraint $\bigwedge_{C \in F} R_C \Rightarrow D$ from $R(F)$.

Proof. Take the partial assignment ρ setting $\ell = 1$ for each $\ell \in R$ and apply Theorem 1. ◀

Finally, we conclude with a closer look at when the $O(n \cdot L)$ worst case in Theorem 1 will actually occur.

▶ **Observation 4.** *In practice, we can often consider the length of the constructed derivation in Theorem 1 to be $O(L)$ rather than $O(n \cdot L)$. This is because the $O(n)$ overhead occurs only in the base case when transforming an axiom from the initial formula to the required form by adding literal axioms (n in the worst case) and saturating as described in Lemma 2. We can achieve the same transformation in $O(1)$ steps when a syntactic implication rule is implemented, as is the case for the VeriPB proof checker. This automatically checks that literal axioms can be added to a previously derived constraint to obtain a specified constraint.*

Anytime Weighted Model Counting with Approximation Guarantees for Probabilistic Inference

Alexandre Dubray
Institute of Information and Communication Technologies, Electonics and Applied Mathematics (ICTEAM), UCLouvain, Belgium

Pierre Schaus
Institute of Information and Communication Technologies, Electonics and Applied Mathematics (ICTEAM), UCLouvain, Belgium

Siegfried Nijssen
Institute of Information and Communication Technologies, Electonics and Applied Mathematics (ICTEAM), UCLouvain, Belgium

Abstract

Weighted model counting (WMC) plays a central role in probabilistic reasoning. Given that this problem is $\#P-$hard, harder instances can generally only be addressed using approximate techniques based on sampling, which provide statistical convergence guarantees: the longer a sampling process runs, the more accurate the WMC is likely to be. In this work, we propose a deterministic search-based approach that can also be stopped at any time and provides hard lower- and upper-bound guarantees on the true WMC. This approach uses a value heuristic that guides exploration first towards models with a high weight and leverages *Limited Discrepancy Search* to make the bounds converge faster. The validity, scalability, and convergence of our approach are tested and compared with state-of-the-art baseline methods on the problem of computing marginal probabilities in Bayesian networks and reliability estimation in probabilistic graphs.

2012 ACM Subject Classification Mathematics of computing → Probabilistic inference problems; Theory of computation → Probabilistic computation; Mathematics of computing → Approximation

Keywords and phrases Projected Weighted Model Counting, Limited Discrepancy Search, Approximate Method, Probabilistic Inference

Digital Object Identifier 10.4230/LIPIcs.CP.2024.10

Supplementary Material
Software (Source Code): `https://github.com/aia-uclouvain/schlandals` [8]
archived at `swh:1:dir:3ffca0d07dbd88cffbbdfda6e3f0ae09a9e77ac0`
Dataset: `https://github.com/AlexandreDubray/bn-benchmarks`
archived at `swh:1:dir:e4644057447aa2f4b9579b41744a6dede462da6a`
Dataset: `https://github.com/AlexandreDubray/probabilistic-graph-benchmarks`
archived at `swh:1:dir:fc1586f609a54f32b09e4d71eda2eb6457f00d20`

1 Introduction

Model counting, the problem of counting the number of satisfying assignments of a propositional formula, is a canonical $\#P-$hard problem with many applications. In particular, (projected) weighted model counting ((P)WMC) has been extensively used for probabilistic reasoning problems, such as computing the marginals in Bayesian Networks [4, 5, 20, 9] or reliability estimation in probabilistic graphs [10, 9]. While model counters are becoming increasingly efficient, computing an exact probability is not always possible for large probabilistic models. Methods computing an approximation of the probability offer better

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 10; pp. 10:1–10:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

scalability, but their guarantees are often statistical [3, 1, 12, 11, 13, 18]. Such methods are often based on sampling, meaning they can be stopped anytime. Moreover, the more the sampling process runs, the more accurate the estimated probability will likely be. For example, model counters providing so-called $(\varepsilon, \delta)-$guarantees return an approximation whose relative error is bounded by a parameter ε with probability $1 - \delta$ [3, 1]. Some methods offer weaker guarantees; they ensure that the probability of returning a wrong lower bound on the probability decreases with the number of iterations [13, 18].

This work proposes a new search-based method for solving PWMC problems. It can be stopped at any time and returns hard bounds on the true count. We show that a DPLL-style model counter can maintain a lower and an upper bound on the true count at each search tree node. This modified version of the classical DPLL-style algorithm can be stopped at any time during its execution, providing deterministic guarantees on the true count. Then, we observe that in the specific context of *weighted* model counting, *Limited Discrepancy Search* [14] (LDS) offers advantages compared to classical depth-first-search methods. Based on this observation, we propose a simple value-selection heuristic that favors the most likely model and use it in an LDS-based search. Our final contribution is to link the lower and upper bound on the true count to the well-known ε-guarantee. We show that bounds can be used to compute an approximate probability that satisfies an ε requirement and to determine the quality of an approximate model count returned by our algorithm when interrupted by a time-out. Using this theoretical result and the LDS-based search, we provide the first anytime method that can return, during its execution, approximate weighted counts with minimal ε-guarantees. We implemented this method, named `Schlandals-LDS-`ε, in `Schlandals` [9], a recently proposed projected weighted model counter specialized for probabilistic inference. We compare our method against state-of-the-art solvers on two major probabilistic reasoning tasks: computing marginal probabilities in Bayesian networks and reliability estimation in probabilistic graphs. Our experiments show that `Schlandals-LDS-`ε performs better on both tasks when looking at the number of solved instances and bound convergence, validating the effectiveness of our bound calculation and LDS-based weighted model counting.

The rest of this paper is organized as follows. We review the technical background for model counting and `Schlandals` in Section 2, followed by a review of the related work in Section 3. In Section 4, we present `Schlandals-LDS-`ε We first introduce how to compute bounds in Section 4.1 and then the search based on Limited Discrepancy Search in Section 4.2. We show the link between the bounds and ε-guarantees in Section 4.3. We compare our work with existing methods and evaluate the convergence of our bounds in Section 5 before concluding in Section 6.

2 Technical Background

Let F be a boolean formula over a set of variables $\mathcal{V}$. An *interpretation* I for a formula F is a complete assignment to all variables in F, and a *partial* interpretation is an interpretation on a subset of all the variables in F. We use $I = \{a = \top, b = \bot\}$ to denote an interpretation in which a is true ($\top$) and b is false ($\bot$). By $F[I]$, we denote the evaluation of F under I, using the standard way of interpreting logical formulas. If $F[I] = \top$, I is a *model* of F. Let $\mathcal{M}_F = \{I \in \{0,1\}^{|\mathcal{V}|} \mid F[I] = \top\}$ be the set of models of F. The goal of a model counter is to compute $|\mathcal{M}_F|$, the number of models of F. In weighted model counting, each variable $v \in \mathcal{V}$ has two weights $\omega(v^+)$ and $\omega(v^-)$; the weight of an interpretation is defined as $\omega(I) = \prod_{v \in I | v = \top} \omega(v^+) \prod_{v \in I | v = \bot} \omega(v^-)$. A weighted model counter computes $\sum_{I \in \mathcal{M}_F} \omega(I)$, the weighted sum of F's models. Let $\mathcal{P} \subseteq \mathcal{V}$ be a subset of the variables and $\pi_\mathcal{P} : \{0,1\}^{|\mathcal{V}|} \mapsto \{0,1\}^{|\mathcal{P}|}$ the function that keeps, from an assignment to all variables of F, only the assignment to the variables in $\mathcal{P}$. A projected model counter computes the number of projected models, defined by $|\pi_\mathcal{P}(\mathcal{M}_F)| = |\{\pi(I) \mid I \in \mathcal{M}_F\}|$. For example, let $F = (a \lor \neg b) \land (b \lor c \lor d) \land (\neg a \lor d)$. This formula has 7 models, but if restricted to $\mathcal{P} = \{a, b\}$, then it only has 3 models. The interpretations $I_1 = \{a = \top, b = \top, c = \top, d = \top\}$ and $I_2 = \{a = \top, b = \top, c = \bot, d = \top\}$ are both models of F but, when restricted on the variables $\{a, b\}$, they are both equal: $\pi_\mathcal{P}(I_1) = \pi_\mathcal{P}(I_2) = \{a = \top, b = \top\}$. A projected weighted model counter computes $PWMC(F) = \sum_{I \in \pi_\mathcal{P}(\mathcal{M}_F)} \omega(I)$. Notice that in such a case, a weight only needs to be defined for an interpretation projected on the variables $\mathcal{P}$.

An approximate model counter returns an ε-approximation ($\varepsilon > 0$) if, given a true count $c^\star$, it returns a count $\hat{c}$ and the following inequality holds:

$$\frac{c^\star}{1+\varepsilon} \leq \hat{c} \leq c^\star(1+\varepsilon), \tag{1}$$

that is, the relative error of the approximation is bounded by a factor $1 + \varepsilon$, and we say that such model counters provide ε-guarantees. Approximate model counters providing (ε, δ)-guarantees return an approximation that respects Equation (1) with a probability $1 - \delta$.

`Schlandals` is a search-based projected weighted model counter specialized for probabilistic inference problems [9]. Its input language has some specific features. It requires, in particular, that the projected variables are partitioned into discrete probability distributions $D_1, \ldots, D_n$.

▶ **Example 1** (Example of an input for Schlandals)**.** Below is an example of a formula F in Schlandals, where all variables v are in $\mathcal{P}$:

$$\begin{array}{rrr} d_2^1 \land d_2^2 \Rightarrow d_1^1 & d_1^1 \land d_2^1 \Rightarrow d_4^1 & d_1^1 \land d_3^2 \land d_3^3 \Rightarrow d_5^2 \\ d_4^1 \Rightarrow d_2^2 & d_1^1 \land d_3^1 \Rightarrow d_5^1 & d_1^1 \land d_5^1 \Rightarrow d_3^2 \end{array}$$

with distributions $D_1 = \{d_1^1, d_1^2\}, D_2 = \{d_2^1, d_2^2\}, D_3 = \{d_3^1, d_3^2, d_3^3\}, D_4 = \{d_4^1, d_4^2\}$, and $D_5 = \{d_5^1, d_5^2\}$.

We denote by $dom_F(D_i)$ the domain of a distribution D_i (i.e., the set of probabilistic variables in its partition). For each variable $v \in dom_F(D_i)$, *one* weight $P(v)$ needs to be specified, in such a manner that $\sum_{v \in dom_F(D_i)} P(v) = 1$. When determining which interpretations are models in `Schlandals`, an implicit constraint is added: if an interpretation I does not fix *exactly* one variable $v \in dom_F(D_i)$ to $\top$ in each distribution D_i, then we have $F[I] = \bot$.

▶ **Example 2.** Continuing our example, an interpretation I setting $d_1^1 = \top$ and $d_1^2 = \top$ is not a model as two variables of the same distribution are set to $\top$. On the contrary, $I = \{d_1^2, d_2^2, d_3^1, d_4^1, d_5^1\}$ is a model of the formula. The weight of this model is obtained by calculating $P(d_1^2) \times P(d_2^2) \times P(d_3^1) \times P(d_4^1) \times P(d_5^1)$, reflecting the probabilities of the choices made for each of the distributions in the interpretation. For conciseness, we describe a partial interpretation I over the weighted variables by the choices (variables set to $\top$) in the distributions.

The importance of the `Schlandals` language is that it allows several probabilistic inference problems to be modeled in a concise manner, among which inference problems in Bayesian Networks and probabilistic graphs; here, the ability to perform *projected* model counting is critical for efficient encoding of inference problems on probabilistic graphs [27, 10]. The fact that weighted variables are part of *distributions* ensures that weighted model counts always correspond to probabilities.

Algorithm 1 PWMC as done in Schlandals [9].

```
1  Function PWMCr($F$)
       input  : A boolean formula $F$ with distributions $\mathcal{D}_F$
       output : $P[F]$ the weighted model count of $F$
2      if $|\mathcal{D}_F| = 0$ then return 1 // $F$ is SAT, return 1. See [9]
3      if $F$ in cache with value $p$ then return $p$
4      $P[F] \leftarrow 0$
5      $D_i \leftarrow$ choose_distribution()                // For heuristics, see [9]
6      foreach $v \in dom_F(D_i)$ do
7          if $TIMEOUT$ then break
8          if $F[v] = \bot$ then continue // Applies propagation
           // $prop(F[v])$ are the choices, for some distributions, forced by
              propagation
9          $p \leftarrow \prod_{v' \in prop(F[v])} P(v')$
10         $C \leftarrow$ components of $F[v]$          // Independent components
11         foreach $F_c \in C$ do $p^c \leftarrow$ PWMCr($F_c$)
12         $P[F] \mathrel{+}= p \times \prod_c p^c$
13     end
14     Adds $F \mapsto p$ in cache and return $p$
15 return PWMCr($F$)
```

The Schlandals language, as presented in [9], also requires that all clauses are Horn. This allows Schlandals to solve satisfaction problems involving only non-weighted variables in polynomial time by means of propagation. While this restriction may seem limiting, it was shown that many problems can be modeled even under this restriction; moreover, the ideas presented below can also be extended to solvers for formulas without this restriction.

The Schlandals solver implements a variant of component caching DPLL search, presented in Algorithm 1. Given how a model's weight is defined, Schlandals does not branch like classical model counters. It starts from a formula F and selects a distribution not yet assigned (line 5). The variable selection heuristic is replaced by a *distribution* selection heuristic, and the value selection heuristic selects which variable in the distribution must be set to true. Then, for each value of its domain (lines 6-13), it sets it to true, applies propagation, and gets the residual formula $F[v]$ (line 8). Here Schlandals performs traditional forms of propagation, such as unit propagation, but also a specific form of propagation for Horn clauses. If a formula remains after propagation, it is decomposed into independent components (line 9) that are solved independently (line 11). The counts of the independent components are multiplied with each other and added to the count of the formula (line 12), multiplied by the probability of the distributions assigned during propagation (computed at line 9).

3 Related Work

Most other approximate model counters are sampling-based; these model counters provide statistical guarantees at best. *Hashing-based* approximate model counters use special classes of hash functions to sample partitions of the search space. They count the models in these small parts and estimate the overall count from these parts. Usually, such solvers provide (ε, δ)−guarantees: they return an approximation whose relative (to the true count) error is bounded by a factor ε, as specified by Equation (1), with a probability of $1 - \delta$. ApproxMC [3, 24, 23] is one such solver, designed for unweighted (projected) model counting. Based on the same idea, WeightMC [1] targets weighted model counting and sampling.

Another class of methods is that of the bounding counters. SampleCount [13] uses a sampler to estimate which variables, after being assigned, divide the solution space in half. When the formula is small enough, it counts exactly its number of models and multiplies the result by a constant factor to obtain an estimated count. This procedure obtains a lower bound with probabilistic guarantees when repeated multiple times. PartialKC [18] is another sampling-based model counter based on partial compilation. Both of these approaches are designed for unweighted model counting. Note that such sampling-based model counters can usually be turned into anytime model counters by estimating model counts even before the predefined approximation guarantee is ensured. In particular, the partial representations produced by PartialKC can be used to compute lower and upper bounds on the true count. However, given that PartialKC works on unweighted formulas, their computations differ from ours. Moreover, PartialKC's compilation process does not aim to fasten the bounds convergence.

Model counting problems are related to several other problems. One such problem is calculating the partition function, Z, of discrete stochastic graphical models, such as Markov Random Fields, Markov Networks, or Cost Function Networks. An algorithm, called Z^*_ε, for calculating an estimate of $\hat{Z}$ with $\frac{Z}{1+\varepsilon} \leq \hat{Z} \leq Z$ guarantees was proposed by Viricel et al. [26] and is implemented in the Toulbar2 [21] system. Also Z^*_ε calculates upper and lower bounds on the final model count, pruning the search based on ε. However, there are several differences between our algorithm and this algorithm: 1) our algorithm is defined over a different form of model that includes projected variables; 2) our algorithm is any time and operates such that an upper-bound on the final model count can be calculated during the search; 3) our upper-bound calculation is optimized for the type of model we work with: it relies on the fact that only variables have weights, integrates the bound calculation with domain propagation, and can be integrated into any DPLL-style algorithm with little overhead; 4) it does not have the behavior stated by Viricel et al. [26] that a larger ε can lead to less pruning; 5) we combine our method with LDS. We will demonstrate the benefits of our approach experimentally.

Finally, weighted model counting has been used as a probabilistic inference mechanism in probabilistic programming languages such as Problog [7, 27]. In particular, anytime methods have been developed specifically for Problog [28], where a TP-compilation incrementally calculates model counts by combining SDD diagrams [6].

4 Anytime Projected Weighted Model Counting with Bounds

This section describes Schlandals-LDS-ε, our anytime search-based approach for computing bounds on $P[F]$. First, we describe how a simple modification of a DPLL search can compute bounds at each search tree node. Then, we present how Limited Discrepancy Search (LDS) can be applied to weighted model counting. Finally, we show the relationship between the computed bounds and ε-approximations, and we show that LDS-based search can be used to compute approximate weighted model counts with deterministic guarantees.

4.1 Computing Bounds During the Search

Our lower and upper bounds are based on calculating which interpretations are (not) models, accounting for how propagation reduces the domain of the distributions. Let F be the (sub-)formula being solved and $\mathcal{D}_F = \{D_1, \ldots, D_n\}$ the distributions in F. We denote by

$dom_F(D_i) = \{d_i^1, \ldots, d_i^{k_i}\}$ the domain of D_i. During the search propagation might have previously removed some variables from $dom_F(D_i)$; hence, $\sum_{v \in dom_F(D_i)} P(v) \leq 1$ generally for subformulas considered during the search.

Lower bound

Our lower bound is relatively simple and amounts to maintaining a sum for models seen till a certain moment. Without loss of generality, let us assume that the solver decides to branch on D_1. We denote by $F[v]$ the formula that remains from F after deciding that, for D_1, variable $v \in dom_F(D_i)$ is true, and after propagation has been performed. Let $prop(F[v])$ be the set of probabilistic variables that are set to true during this propagation. Essentially our algorithm calculates the probability $P[F]$ recursively by exploring all possibilities for D_1, using:

$$P[F] = \sum_{v \in dom_F(D_1)} \left(P[F[v]] \times \prod_{v' \in prop(F[v])} P(v') \right)$$

A lower bound is obtained by executing this sum over a subset of $dom_F(D_1)$; hence, we can incrementally maintain a lower bound while considering the possibilities in $dom_F(D_1)$.

Upper bound

Maintaining an upper bound is conceptually more complex, but we propose an upper bound that can be calculated with relatively little overhead in a DPLL-based solver. A first naïve upper bound, which we will improve afterwards, is $P(\mathcal{D}_F)$, defined as follows:

$$P(\mathcal{D}_F) = \prod_{D_i \in \mathcal{D}_F} \left(\sum_{v \in dom_F(D_i)} P(v) \right) \tag{2}$$

i.e., $P(\mathcal{D}_F)$ represents the maximum probability that F can obtain if all remaining interpretations are models. The intuition behind this formula is that an upper bound can be obtained by calculating a weighted sum over the interpretations that are in the cartesian product of the remaining domains of the remaining distributions. The product calculates this sum efficiently. As illustrated in the following example, this upper bound can be tightened during the search when the domains of the distributions are pruned by propagation.

▶ **Example 3.** Continuing Example 1, let us consider the case when, at the root, the solver decides to branch on D_4 and assigns $d_4^1 = \top$. In that case, the clause $d_4^1 \implies d_2^2$ becomes $\top \implies d_2^2$, forcing $d_2^2 = \top$. Since exactly one variable must be true in each distribution, we have that $d_2^1 = \bot$. This means that no interpretation containing $d_4^1 = \top$ and $d_2^1 = \top$ can be a model of F, and the upper bound for the formula F can be improved. Assuming that no other variable is set to $\bot$ during that propagation, the weight of the removed interpretations is equal to $P(d_4^1) \times P(d_2^1)$, and the upper bound for F can be decreased to $1.0 - P(d_4^1) \times P(d_2^1)$.

This improved upper bound calculation is formalized in Algorithm 2, which is an adaptation of Algorithm 1 to compute bounds on $P[F]$. Intuitively, this algorithm computes, for any search tree node with sub-formula F', a sum of probabilities of models of F', denoted p_{in}, and a sum of probabilities of interpretations that are not models, denoted p_{out}. From these two values, the lower and upper bounds can be inferred. The structure of Algorithm 2 is the same as Algorithm 1. The search stops when there are no more distributions in F (line 2) or

Algorithm 2 PWMC with Bounds Computaton.

```
1  Function PWMC-Bounds($F$)
       input  : A boolean formula $F$ with distributions $\mathcal{D}_F$
       output : The probability of the considered models ($p_{in}$) and non-models ($p_{out}$)
2      if $|\mathcal{D}_F| = 0$ then return $(1, 0)$
3      if $F$ in cache with values $(p'_{in}, p'_{out})$ then return $(p'_{in}, p'_{out})$
4      $p_{in} \leftarrow 0$; $p_{out} \leftarrow 0$
5      $D_i \leftarrow$ choose_distribution()                    // For heuristics, see [9]
6      foreach $v \in dom_F(D_i)$ do
7          if TIMEOUT then break
8          if $F[v] = \bot$ then
9              $p_{out}$ += $P(v) \times P(\mathcal{D}_F \backslash \{D_i\})$                // Uses Equation (2)
10         else
11             $p \leftarrow \prod_{v' \in prop(F[v])} P(v')$
12             $p_{out}$ += $P(v) \times (P(\mathcal{D}_F \backslash \{D_i\}) - P(\mathcal{D}_{F[v]}))$   // Uses Equation (2)
13             $C \leftarrow$ components of $F[v]$                   // Independent components
14             foreach $F_c \in C$ do
15                 $(p^c_{in}, p^c_{out}) \leftarrow$ PWMC-Bounds($F_c$)
16             end
17             $p_{in}$ += $p \times \prod_c p^c_{in}$
18             $p_{out}$ += $p \times (P(\mathcal{D}_{F[v]}) - \prod_c (P(\mathcal{D}_{F_c}) - p^c_{out}))$   // Uses Equation (2)
19         end
20     end
21     Add $F \mapsto (p_{in}, p_{out})$ in cache and return $(p_{in}, p_{out})$
22 $(p_{in}, p_{out}) \leftarrow$ PWMC-Bounds($F$, $\varepsilon$)
23 return $(p_{in}, 1 - p_{out})$
```

the formula is in the cache (line 3). Otherwise, it selects a distribution to branch on (line 5). Then, it iterates over its domain (lines 6-20), applies Schlandals' propagation (line 8), and recursively explores the independent components (lines 13-16). Moreover, the computation of p_{in}, the sum of F models' probability (line 17), is the same as the computation of $P[F]$ in Algorithm 1.

Hence, the main difference between Algorithm 2 and Algorithm 1 is the computation of p_{out}. When exploring sub-problems of F (i.e., branching for some value v of a distribution), $F[v]$, the formula obtained after applying Schlandals' propagation, might turn out to be unsatisfiable. Then, all interpretations containing $v = \top$ cannot be models of F, and p_{out} is increased (line 8). This rule also applies after propagation: propagation may remove some variables from a distribution's domain, making some interpretations non-models. The probability of the interpretations setting one of these variables to $\top$ can be added to p_{out} (line 10). This computation is based on the difference in the maximum probability of the distributions before ($P(\mathcal{D}_F \setminus \{D_i\})$) and after the propagation ($P(\mathcal{D}_{F[V]})$). Finally, the probabilities of the non-models of each independent component are combined and added to p_{out} (line 18). The intuition for this formula is that if one partial interpretation in one of the components evaluates to false, then the whole interpretation evaluates to false regardless of the partial interpretation in the other components. Hence, if U_c represents the random event that the c-th component evaluates to $\bot$ (with probability p^c_{out}), we wish to compute $P(\vee_{i=1}^{|C|} U_i) = P(\neg(\wedge_{i=1}^{|C|} \neg U_i))$; this gives the indicated formula as for each component the probability that it is satisfiable is given by $P(\mathcal{D}_{F_c}) - p^c_{out}$.

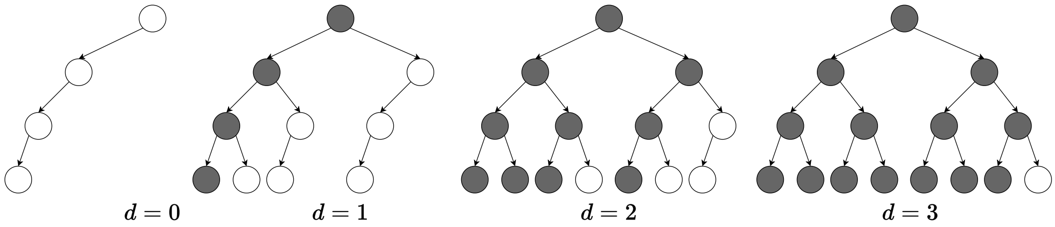


Figure 1 Example of iterative (from left to right) exploration of the search space with Limited Discrepancy Search. At each iteration, d is the current discrepancy, starting from 0. The white nodes represent newly explored nodes, while the grey ones represent nodes explored in previous iterations.

4.2 Limited Discrepancy Search for Weighted Model Counting

Limited Discrepancy Search [14] is a search procedure initially designed to solve constraint satisfaction problems in situations where it is reasonable to assume that a branching heuristic will often be correct. An essential intuition behind LDS is that if a branching heuristic is always correct, a solution can be found by following the leftmost branch of the search tree, that is, the branch preferred by the heuristic. If the number of times the heuristic is wrong is limited, and the lefthand branch of a node does not provide a solution, then the heuristic should be incorrect less often in the righthand subtree. Limited Discrepancy Search builds on this intuition; it iteratively explores the search space, deviating more and more from the heuristic, as illustrated in Figure 1. Initially, it follows exactly the branching heuristic, exploring the leftmost branch of the search tree. If no solution is found, it allows deviating from the heuristic at one node per branch: the discrepancy is 1. This process continues, incrementing the maximum number of discrepancies each time until a solution is found or the whole search space is explored. This intuition naturally extends to optimization problems: a good heuristic will guide the search toward a good solution and provide tight bounds. Hence, large parts of the search space can be pruned in subsequent iterations of the LDS. However, it is not obvious that counting problems benefit from such a search scheme; the whole search space must be explored to calculate the exact model count.

However, we argue that LDS offers advantages compared to Algorithm 2 in the specific setting of *anytime weighted* model counting, where the interest is to converge the bounds as rapidly as possible. Indeed, the interpretations of a formula F are unlikely to have uniform weights. If LDS' assumptions are true, then the most likely interpretations can be found using a small discrepancy, which is very fast for LDS, as it ignores large parts of the search space. On the other hand, methods based on depth-first-search, as presented in Algorithm 1 must explore the whole sub-tree, including unlikely interpretations, before switching to a more promising part of the search space; moreover, when decomposing a formula in components, DFS-based algorithms face the problem that they cannot raise the lower-bound before all components have been considered. Hence, for the hardest problems, DFS-based methods might even time out before exploring likely interpretations and considering all components. Limited Discrepancy Search has the potential not to suffer from these problems, if the branching heuristic is reasonable.

Algorithm 3 gives the modified procedure to perform an LDS-based DPLL search in `Schlandals`. The structure of the algorithm is the same as Algorithm 2: a recursive procedure (lines 1-18) explores the search space by selecting a distribution (line 5), exploring its values (lines 6-17) and recursively solving the sub-problems (line 12-14) while maintaining the p_{in} and p_{out} counts (lines 11,15-16). However, there are some key differences, which we will describe next.

Algorithm 3 PWMC with Bounds Computation and Limited Discrepancy Search.

```
1  Function Schlandals-LDS($F$, $d$)
       input  : A boolean formula $F$ with distributions $\mathcal{D}_F$ and a discrepancy budget $d$
       output : The probability of the considered models ($p_{in}$) and non-models ($p_{out}$)
2      if $|\mathcal{D}_F| = 0$ then return $(1, 0)$
3      if $F$ in cache with values $(p_{in}, p_{out}, d')$ with $d' \leq d$ or $p_{in} + p_{out} = P(\mathcal{D}_F)$ then
        return $(p_{in}, p_{out})$
4      $p_{in} \leftarrow 0$; $p_{out} \leftarrow 0$
5      $D_i \leftarrow$ choose_distribution()                    // For heuristics, see [9]
6      for $k \leftarrow 1$ to $\min(d+1, |dom_F(D_i)|)$ do
7          if $TIMEOUT$ then break
8          $v \leftarrow k-$th values of $dom_F(D_i)$          // Has the $k-$th highest weight
9          if $F[v] = \bot$ then $p_{out}$ += $P(v) \times P(\mathcal{D}_F \backslash \{D_i\})$; continue // Equation (2)
10         $p \leftarrow \prod_{v' \in prop(F[v])} P(v')$
11         $p_{out}$ += $P(v) \times (P(\mathcal{D}_F \backslash \{D_i\}) - P(\mathcal{D}_{F[v]}))$
12         $C \leftarrow$ components of $F[v]$
13         foreach $F_c \in C$ do
               // Discrepancy not decrement for first child
               $(p_{in}^c, p_{out}^c) \leftarrow$ Schlandals-LDS$(F_c, d - (k-1))$
14         end
15         $p_{in}$ += $p \times \prod_c p_{in}^c$
16         $p_{out}$ += $p \times (P(\mathcal{D}_F) - \prod_c (P(\mathcal{D}_{F_c}) - p_{out}^c))$
17     end
18     Add $(p_{in}, p_{out}, d)$ in cache and return $(p_{in}, p_{out})$
19 $d \leftarrow 0$; $lb \leftarrow 0$; $ub \leftarrow 0$
20 while not $TIMEOUT$ do
21     $(p_{in}, p_{out}) \leftarrow$ Schlandals-LDS($F$, $d$)
22     $lb \leftarrow \max(p_{in}, lb)$, $ub \leftarrow \min(1 - p_{out}, ub)$
23     output $(lb, ub)$
24     $d$ += 1
25 end
26 return $(lb, ub)$
```

First, when a formula is found in the cache (line 3), its result is not automatically returned. Indeed, if a node in the search tree is only partially explored when first encountered, its bounds are not tight, and $p_{in} + p_{out} \neq P(\mathcal{D}_F)$. Such a formula must still be explored, but only if the discrepancy budget is higher than the one stored in the cache; otherwise, no new interpretation can be found.

Then, when a distribution D_i is selected for branching, the iteration on its domain (line 6) differs from classical DPLL-search: it is limited by the discrepancy budget. Moreover, the effectiveness of LDS heavily depends on the value selection heuristic. We propose a simple heuristic that favors the most likely interpretation first. Let $dom_F(D_i) = \{v_1, \ldots, v_n\}$ be the domain of D_i. Let us assume that the domain is iterated from v_1 to v_n; given a discrepancy budget d, the values $v_1, \ldots, v_d$ are explored. We propose to order the values of the domain such that $P(v_1) \geq P(v_2) \geq \ldots \geq P(v_n)$. The rationale behind this heuristic is the following. The weight of an interpretation of F is computed as $\prod_{D_i \in \mathcal{D}_F} P(v_i^\top)$ with $v_i^\top$ being the variable set to $\top$ for distribution D_i. Hence, selecting the most likely variable first aims to favor the most likely interpretations.

Finally, the outer loop calling the recursive function (lines 20-25) is specific to LDS. The bounds are initialized (line 19), and while the time limit is not reached, an iteration of LDS is done (line 21). The bounds are updated (line 22), output to the user (line 23), and the discrepancy is incremented (line 24). Notice that the bounds are updated if they are better than the previously found bounds. Indeed, if the time limit is reached, the last iteration of LDS may explore only a small part of the search space, leading to worse bounds than in the previous iteration. When the whole search space has been explored or the time limit is reached, the bounds are returned to the user (line 26).

4.3 From Bounds to an Epsilon Guarantee

We have shown that it is possible to modify a DPLL-style search algorithm to compute a lower and upper bound on the true probability. We now show a relationship between these bounds and an ε-guarantee. Remember that a model counter computes an approximate probability $\hat{p}$ of p with ε-guarantee on its error if $\frac{p}{1+\varepsilon} \leq \hat{p} \leq p(1+\varepsilon)$.

▶ **Theorem 4.** *Let F be a formula with true probability $P[F]$, $\varepsilon \geq 0$ an error factor. If $P_l[F]$ is a lower bound on $P[F]$, $P_u[F]$ an upper bound on F, and $P_u[F] \leq P_l[F] \times (1+\varepsilon)^2$ then we have that*

$$\frac{P[F]}{1+\varepsilon} \leq \sqrt{P_l[F] \times P_u[F]} \leq P[F](1+\varepsilon).$$

Proof. Let us first prove the left part of the inequality. By definition, we have that $P[F] \leq P_u[F]$ and, by assumption, $\frac{P_u[F]}{(1+\varepsilon)^2} \leq P_l[F]$, hence

$$\begin{aligned} P[F]^2 \leq P_u[F]^2 \Leftrightarrow \frac{P[F]^2}{(1+\varepsilon)^2} &\leq \frac{P_u[F]^2}{(1+\varepsilon)^2} = \frac{P_u[F]P_u[F]}{(1+\varepsilon)^2} \leq P_l[F] \times P_u[F] \\ \Leftrightarrow \sqrt{\frac{P[F]^2}{(1+\epsilon)^2}} &= \frac{P[F]}{(1+\varepsilon)} \leq \sqrt{P_l[F] \times P_u[F]} \end{aligned}$$

We prove the second inequality in a similar manner.

$$\begin{aligned} P[F] \geq P_l[F] \Leftrightarrow P[F]^2(1+\varepsilon)^2 &\geq P_l[F]^2(1+\varepsilon)^2 = P_l[F]P_l[F](1+\varepsilon)^2 \geq P_l[F]P_u[F] \\ \Leftrightarrow \sqrt{P[F]^2(1+\varepsilon)^2} &= P[F](1+\varepsilon) \geq \sqrt{P_l[F] \times P_u[F]} \end{aligned}$$ ◀

This theorem tells us two things. First, if the bounds on a probability $P[F]$ are close enough, with respect to an allowed ε error factor, the search can be stopped before a time-out is reached and we can return a probability that respects a required ε-guarantee. Second, however, if a time-out is reached, we can also still estimate the quality of the model count at that moment, with respect to an $\varepsilon-$guarantee: given a lower bound $P_l[F]$ and an upper bound $P_u[F]$, Theorem (4) allows us to compute the *minimum required* ε that would stop the search at that moment. Algorithm 4 shows an adaptation of the outer loop of Algorithm 3 to perform LDS with an optional error factor ε. The algorithm runs until the condition of Theorem 4 is met or until the time limit is reached. At each iteration, it outputs an approximate probability with the ε error factor that would stop the search at that moment.

We believe that this setting can be useful for instance in experiments where a large number of probabilities need to be calculated; using our solver this can be done with a time out limit on each query, while at the end of the experiment statistics can be calculated on the approximation quality of the probabilities calculated in the experiment.

Algorithm 4 Approximate PWMC with LDS.

```
$d \leftarrow 0$; $lb \leftarrow 0$; $ub \leftarrow 0$
while not $TIMEOUT$ and not $ub \leq lb(1+\varepsilon)^2$ do
    $(p_{in}, p_{out}) \leftarrow$ Schlandals-LDS($F$, $d$)
    $lb \leftarrow \max(p_{in}, lb)$, $ub \leftarrow \min(1-p_{out}, ub)$
    $\varepsilon_d \leftarrow \sqrt{\frac{ub}{lb}} - 1$
    output $(\sqrt{lb \times ub}, \varepsilon_d)$
    $d \mathrel{+}= 1$
end
return $(\sqrt{lb \times ub}, \sqrt{\frac{ub}{lb}} - 1)$
```

5 Experimental Results

In this section, we evaluate the effectiveness of our LDS-based[1] search method against state-of-the-art methods. All methods are evaluated on two axes: i) How many instances they can solve and ii) how fast they converge toward the true probability. Moreover, we evaluate the effectiveness of our value-selection heuristic.

We evaluate our methods against the following state-of-the-art methods. We compare against `PartialKC` [18], a recently proposed anytime *unweighted* model counter[2] and `ApproxMC` [3, 24, 23], which is a popular hashing-based model counter providing $(\varepsilon, \delta)-$guarantees. We omit `WeightMC` [1] in our experiments as the available code was not updated recently and did not scale on our instances. We also compare our LDS search against the TP-compilation algorithm of `Problog`[7, 28], which provides a lower and an upper bound on the probability in an anytime fashion. Unfortunately, the upper bound is not available to users, and there is no easy way to extract it[3]. The Toulbar2 [21] solver was also run with the algorithm presented in [26]. Finally, using Theorem 4, we evaluate an anytime version of Algorithm 2, which can be seen as the DFS alternative to our LDS-based search. To do so, we ran Algorithm 2 with increasingly high timeouts, storing for each timeout the returned bounds. In the rest of this section, we denote by `DFS` this approach, and by `LDS` the approach presented in Algorithm 4. To better evaluate these approaches, we also ran `D4` [15, 16], `GPMC` [25] and `ExactMC` [17] on our benchmarks. All methods ran with a timeout of 600 seconds and a memory limit of 15Gb when the option was available with the solver. Except for parameters related to approximations, all methods have been executed with their default parameters. We evaluated these methods on two problems: computing the marginals in Bayesian networks without evidence and computing reachability queries in probabilistic graphs.

The *Bayesian networks* have various sizes, ranging from a few parameters to tens of thousands, and originated from the bnlearn R package [22]. For each network, we use one query per value of the leaves: if a leaf node in the network has four values, we create four instances that compute the marginal probability of each value. Overall, there are 2749

[1] The source code is available at `https://github.com/aia-uclouvain/schlandals`.

[2] At the time of the writing of this work, there exists a parameter in `PartialKC` to perform weighted model counting. Unfortunately, the feature is not yet implemented

[3] We checked this with the maintainers of the Problog implementation.

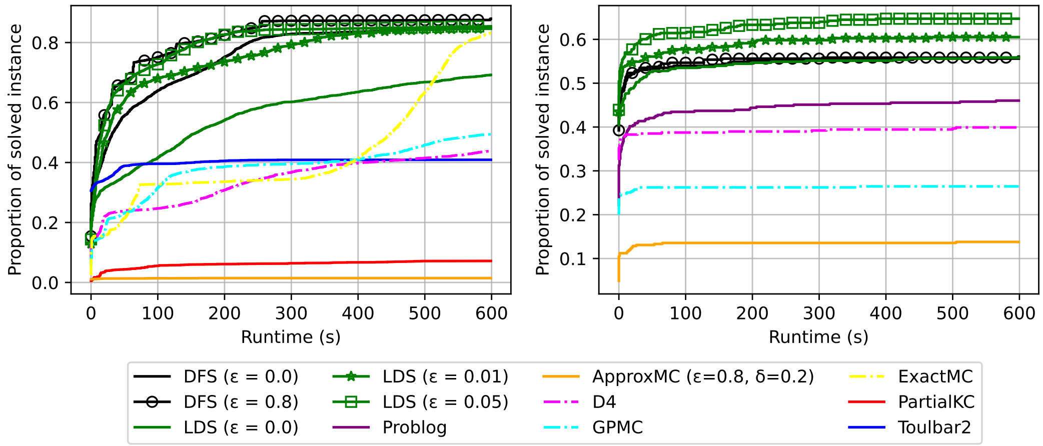


Figure 2 Proportion of solved instances over time, with a timeout of 600 seconds, for Bayesian networks (left) and reliability estimation (right) problems.

instances[4]. For Schlandals, we use the encoding presented in [9], and for PartialKC and ApproxMC, we use the encoding from [4]. Since these solvers work on unweighted formulas, we transformed the weighted instances into unweighted ones using the approach proposed in [2], which encodes the weight with additional variables and clauses. Then, the unweighted count can be divided by a normalization factor to obtain the weighted model count. A precision parameter can adjust the number of clauses and variables added at the cost of a less precise weighted model count. We chose a precision of 10 so that the result after normalization is similar to the true probability up to five decimals. However, using a lower precision did not change the results significantly. For Toulbar2, we used the UAI format with, as evidence, the pair node-value to query. Since Problog is not designed to work directly on CNF formulas, implementing the various optimizations for encoding Bayesian networks is impossible. Hence, we decided not to run it on these benchmarks as it would only scale to the easiest instances.

For the *reachability queries*, we took the graphs representing power grid networks in Europe and the USA extracted by the GridKit tool [19, 29]. For each of these graphs, sub-graphs were created by splitting the nodes and edges by country (for Europe) and state (for the USA). Following the description in [10], we assign each edge a probability of 0.125 of being down. Finally, five queries are created for each sub-graph by taking random pairs of nodes and computing their connection probability[5]. For Schlandals, we again used the encoding presented in [9], which is based on the ones presented in [10] and used for the other solvers. Unfortunately, PartialKC and Toulbar2 are not projected model counters and cannot be launched on these instances. For ApproxMC, the instances were made unweighted using the same approach as for the Bayesian networks but with a precision of 3.

Evaluation of Solving Instances

First, we analyze how many instances each method can solve. Here, we consider an instance solved if it respects the approximation contract, if any, or finds the true probability. For example, an instance with a probability of 0.5 would be solved by a solver returning 0.55

[4] The scripts to generate the instances can be found at https://github.com/AlexandreDubray/bn-benchmarks

[5] The scripts to generate the instances can be found at https://github.com/AlexandreDubray/probabilistic-graph-benchmarks

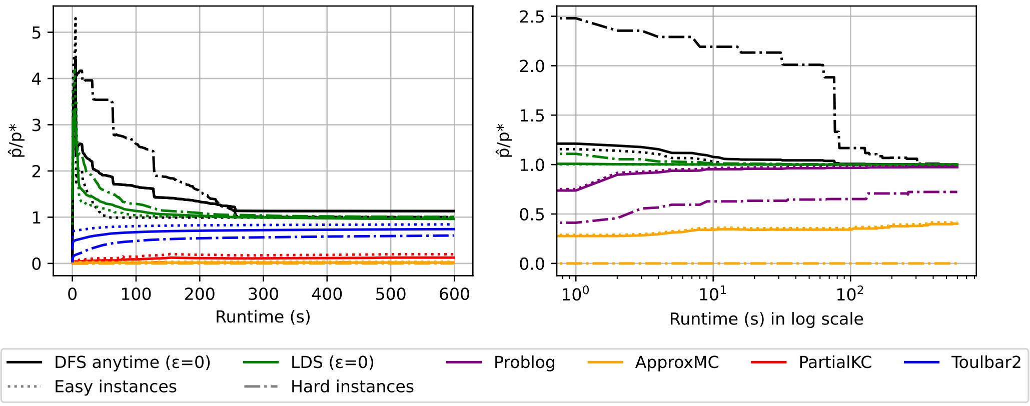


Figure 3 Average convergence of the estimated probability ($\hat{p}$) towards the true probability ($p^\star$) for Bayesian networks (left) and reliability estimation (right) problems. For `Schlandals`, the estimated probability is given by $\sqrt{lb \times ub}$; the other methods directly output an estimated probability. The continuous lines represent the average over all instances, the dotted lines are the average over the easy instances (can be solved by `Schlandals` in less than 60 seconds), and the dashed lines are the average over the hard instances.

when required to produce an ε-approximation with $\varepsilon = 0.2$. Figure 2 shows the proportion of solved instances within the time limit for Bayesian networks (left) and reachability queries (right). It can be seen that `Schlandals` performs the best regardless of the problem and the type of solver (exact, approximate, or anytime). The unweighted model counters are performing less well. Adding variables and clauses for the weights heavily impacts their performance, particularly for large probabilistic models with thousands of weights. In our experiments, on most instances, `Toulbar2` was not able to improve its performance when calculating an ε-approximation using the algorithm presented in [26]. Hence, this graph only includes the result for $\varepsilon = 0$.

It can be seen that when solving the problems exactly ($\varepsilon = 0$), the LDS version of `Schlandals` performs less well than its DFS version; this is to be expected as parts of the search space are traversed multiple times. However, when a small error (ε) is acceptable, then `LDS` performs roughly the same as `DFS` on Bayesian networks and surpasses it on reachability queries (with $\varepsilon = 0.01$, the DFS does not solve more instances within the time limit, so we did not include this curve in the graph). Even more striking, `LDS` can solve at least as many instances, in the same amount of time, as `DFS` and produce probabilities under much stronger guarantee requirements ($\varepsilon = 0.05$ instead of $\varepsilon = 0.8$). Although `LDS` has some overhead by exploring some parts of the search space, and it can be seen when the whole search space must be explored, when computing ε-approximation, it seems that it allows to converge much faster towards the true probability. We confirm this intuition in the next section by analyzing the convergence of the various methods.

Convergence of the Anytime Methods

Figure 3 shows the average convergence of the methods towards the true probability when computing it exactly (i.e., with $\varepsilon = 0$). For the two versions of `Schlandals`, we use Theorem 4 and the bounds to produce an approximation. We modified the outer loop of `ApproxMC` (see Algorithm 1 in [3]) to output the estimated probability at each iteration. While these intermediate solutions have weaker guarantees, considering `ApproxMC` as an anytime model

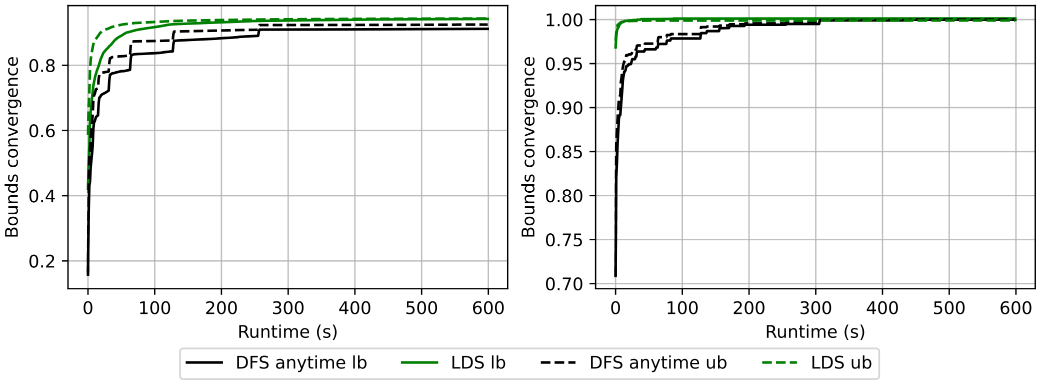


Figure 4 Lower and upper bound convergence towards the true probability $p^\star$ on Bayesian networks (left) and reachability queries (right) for the two versions of Schlandals.

counter is possible. All other methods provide a lower bound on the probability during their execution, which is used as a comparison in the figure. Figure 3 shows the average convergence on all instances (solid lines), but also on easy (dotted lines) and hard instances (dashed lines). We use a simple threshold to consider an instance difficult or not: if Schlandals can solve the instance in less than a minute, we consider the instance as easy. For the Bayesian networks, the true probability is known for each instance, but this is not true for reachability queries. Hence, for the later problem, we only consider, in this graph, the instances for which at least one solver can produce an exact solution. Such instances are mainly the easiest ones, explaining why the methods converge faster than for Bayesian networks.

As previously noted, Schlandals is the best-performing method. Hence, its estimated probability converges, on average, to the true probability. However, the gap between the convergence of the easy and hard instances is much larger for the DFS version of Schlandals than for any other method. Due to its depth-first nature, it stays in parts of the search space containing small-weight interpretations. On the other hand, LDS suffers much less from this problem, and the convergence gap between the easy and the hard instances is smaller. Due to its inability to solve the hardest instances, Toulbar2 does not converge, on average, toward the true probability. We hypothesize that it gets trapped in part of the search space where it accumulates little probability mass before being timed out. Finally, due to their poor scalability on weighted problems, PartialKC and ApproxMC converge poorly. Indeed, they can not produce a first solution for many hard instances. Hence, the default lower bound on these instances is 0, pulling their average convergence towards that value.

Finally, we evaluate if limited discrepancy search effectively allows finding likely models first. To do so, we analyze how the lower and upper bounds converge toward the true probability. If our search-based methods find the most likely model first, the bounds should converge quickly towards the probability, then plateau until the final convergence or timeout. To assess the convergence of a lower bound $P_l[F]$, we use the metric $P_l[F]/P[F] \in [0, 1]$; when no solution has been found, it is 0 and when all solutions have been found it is 1. For the upper bound $P_u[F]$, we use $(1 - P_u[F])/(1 - P[F])$, which exhibits the same characteristics. Figure 4 shows, for both problems, the average convergence, using the metrics defined above, for DFS and LDS. It can be seen that both methods quickly accumulate probability mass for models *and non-models* of F, validating that our value-selection heuristic favors the most likely models. Moreover, LDS performs significantly better than DFS, reaching an average of more than 0.9 for both metrics in less than 100 seconds.

6 Conclusion

In this work, we proposed a new method for performing (projected) weighted model counting that can be stopped at any time during its execution. Unlike other anytime methods, the proposed approach provides deterministic lower *and upper* bounds on the true count. We have shown that the propagators of DPLL-style weighted model counters can be leveraged to maintain such an upper bound during exploration. We proposed a simple heuristic that favors exploring the highly weighted interpretations first, fastening the convergence of the bounds. We enhanced this convergence further using *Limited Discrepancy Search* (LDS). Finally, we have shown that these lower and upper bounds can be linked to the well-known notion of ε-approximations; hence, the proposed method can be used seamlessly to perform an interruptable computation for either the true weighted model count or an ε-approximation. We implemented this method in the `Schlandals` solver, specialized for probabilistic inference, and evaluated it against state-of-the-art methods on probabilistic reasoning tasks. Our experiments show that `Schlandals` is the best-performing solver on the studied tasks and that the LDS-based search outperforms classical DFS when computing ε-approximations.

The method presented in this work uses a simple strategy for the discrepancy, and more work could be done to enhance the performance of the LDS-based search. Moreover, the method presented here could be applied to other weighted model counters, including those that support arbitrary CNF formulas, such as provided in model counting competitions; moreover, an interesting research direction is to apply LDS to unweighted model counting. In this case, our simple heuristic can not be applied since all interpretations have the same weight, and another heuristic would be necessary.

References

1 Supratik Chakraborty, Daniel Fremont, Kuldeep Meel, Sanjit Seshia, and Moshe Vardi. Distribution-aware sampling and weighted model counting for SAT. In *AAAI*, 2014.

2 Supratik Chakraborty, Dror Fried, Kuldeep S. Meel, and Moshe Y. Vardi. From Weighted to Unweighted Model Counting. In *IJCAI*, 2015.

3 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*, 2016.

4 Mark Chavira and Adnan Darwiche. Encoding CNFs to empower component analysis. In *Theory and Applications of Satisfiability Testing-SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings 9*. Springer, 2006.

5 Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7), 2008.

6 Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

7 Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7. Hyderabad, 2007.

8 Alexandre Dubray. Schlandals. Software, version 1.0.3., swhId: `swh:1:dir:3ffca0d07dbd88cffbbdfda6e3f0ae09a9e77ac0` (visited on 2024-08-19). URL: `https://github.com/aia-uclouvain/schlandals`.

9 Alexandre Dubray, Pierre Schaus, and Siegfried Nijssen. Probabilistic Inference by Projected Weighted Model Counting on Horn Clauses. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.CP.2023.15*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. `doi:10.4230/LIPIcs.CP.2023.15`.

10 Leonardo Duenas-Osorio, Kuldeep Meel, Roger Paredes, and Moshe Vardi. Counting-based reliability estimation for power-transmission grids. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

11 Vibhav Gogate and Rina Dechter. SampleSearch: Importance sampling in presence of determinism. *Artificial Intelligence*, 175(2), 2011.

12 Vibhav Gogate and Rina Dechter. Importance sampling-based estimation over and/or search spaces for graphical models. *Artificial Intelligence*, 184, 2012.

13 Carla P. Gomes, Jörg Hoffmann, Ashish Sabharwal, and Bart Selman. From Sampling to Model Counting. In *IJCAI*, volume 2007, 2007.

14 William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *IJCAI (1)*, 1995.

15 Jean-Marie Lagniez and Pierre Marquis. An Improved Decision-DNNF Compiler. In *IJCAI*, volume 17, 2017.

16 Jean-Marie Lagniez and Pierre Marquis. A recursive algorithm for projected model counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 2019.

17 Yong Lai, Kuldeep S. Meel, and Roland HC Yap. The power of literal equivalence in model counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, 2021.

18 Yong Lai, Kuldeep S. Meel, and Roland HC Yap. Fast Converging Anytime Model Counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, 2023.

19 Wided Medjroubi, Ulf Philipp Müller, Malte Scharf, Carsten Matke, and David Kleinhans. Open data in power grid modelling: New approaches towards transparent grid models. *Energy Reports*, 3, 2017.

20 Tian Sang, Paul Beame, and Henry Kautz. Solving Bayesian networks by weighted model counting. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, volume 1. AAAI Press, 2005.

21 T. Schiex, S. de Givry, and M. Sanchez. Toulbar2—an open source weighted constraint satisfaction solver. *URL http://mulcyber. toulouse. inra. fr/projects/toulbar2*, 2006.

22 Marco Scutari. Learning Bayesian networks with the bnlearn R package. *arXiv preprint arXiv:0908.3817*, 2009. arXiv:0908.3817.

23 Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, Detached, and Lazy CNF-XOR solving and its Applications to Counting and Sampling. In *CCAV*, 2020.

24 Mate Soos and Kuldeep S. Meel. BIRD: Engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 2019.

25 Ryosuke Suzuki, Kenji Hashimoto, and Masahiko Sakai. Improvement of projected model-counting solver with component decomposition using SAT solving in components. Technical report, JSAI Technical Report, SIG-FPAI-506-07, 2017.

26 Clément Viricel, David Simoncini, Sophie Barbe, and Thomas Schiex. Guaranteed Weighted Counting for Affinity Computation: Beyond Determinism and Structure. In Michel Rueher, editor, *CP*, 2016.

27 Jonas Vlasselaer, Angelika Kimmig, Anton Dries, Wannes Meert, and Luc De Raedt. Knowledge compilation and weighted model counting for inference in probabilistic logic programs. In *Proceedings of the First Workshop on Beyond NP*. AAAI Press, 2016.

28 Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. Anytime inference in probabilistic logic programs with Tp-compilation. In *IJCAI*, 2015.

29 Bart Wiegmans. Gridkit: European And North-American Extracts, March 2016. doi:10.5281/ZENODO.47317.

A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers

Maarten Flippo
Delft University of Technology, The Netherlands

Konstantin Sidorov
Delft University of Technology, The Netherlands

Imko Marijnissen
Delft University of Technology, The Netherlands

Jeff Smits
Delft University of Technology, The Netherlands

Emir Demirović
Delft University of Technology, The Netherlands

Abstract

Proof logging is used to increase trust in the optimality and unsatisfiability claims of solvers. However, to this date, no constraint programming solver can practically produce proofs without significantly impacting performance, which hinders mainstream adoption. We address this issue by introducing a novel proof generation *framework*, together with a CP proof format and proof checker. Our approach is to divide the proof generation into three steps. At runtime, we require the CP solver to only produce a proof sketch, which we call a scaffold. After the solving is done, our proof processor trims and expands the scaffold into a full CP proof, which is subsequently verified. Our framework is agnostic to the solver and the verification approach. Through MiniZinc benchmarks, we demonstrate that with our framework, the overhead of logging during solving is often less than 10%, significantly lower than other approaches, and that our proof processing step can reduce the overall size of the proof by orders of magnitude and by extension the proof checking time. Our results demonstrate that proof logging has the potential to become an integral part of the CP community.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Theory of computation → Logic and verification

Keywords and phrases proof logging, formal verification, constraint programming

Digital Object Identifier 10.4230/LIPIcs.CP.2024.11

Supplementary Material *Software (Source code and evaluation)*: `https://doi.org/10.5281/zenodo.12567314`

Funding *Maarten Flippo and Jeff Smits*: 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).
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 Introduction

Constraint Programming (CP) is a pivotal field recognized for its efficacy in addressing intricate combinatorial challenges across diverse sectors. Central to its importance is the *model + solve* paradigm, which allows practitioners to express problem constraints in a

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 11; pp. 11:1–11:20
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

declarative manner without having to reason about *how* the problem needs to be solved. By combining modeling flexibility with powerful solving techniques, CP enables intuitive problem formulation and effective optimisation strategies.

In a typical CP workflow, the solver is often treated as a black box. When a solver reports a solution, it is easy to verify whether all constraints are satisfied. However, for unsatisfiability and optimality claims, no such trivial verification can be done. How can we be sure that a model has no solutions when that is the claim of a solver?

Unfortunately, bugs are common due to the complex nature of solvers, regardless of the paradigm. For example, fuzz testing [41] revealed that virtually all solvers in the MaxSAT Evaluation 2022 have bugs. Similar issues have been reported for SAT and SMT solvers [13]. A variety of techniques have been used to discover bugs in well-maintained CP solvers [4, 29]. Long-standing MIP solvers may report incorrect results due to numerical instabilities [18]. For applications where correctness is crucial, e.g., chip design [1] and combinatorial actions [22], a bug may undermine the entire process and result in legal and financial consequences.

The general approach to verifying claims of unsatisfiability or optimality is *proof logging*. The proof can then be checked by a separate *proof checker*, possibly implemented in a formally verified toolchain. As a result, when a checker accepts a proof as correct, the claim of the solver may be trusted with high confidence.

Proof logging has proven immensely successful in the SAT community [47, 46, 19]. It has enabled the closing of several open math problems [36, 34] and has become standardised in the community, e.g., solvers in the SAT competition [6] must produce proofs in order to participate. Aside from SAT solvers, SMT solvers can also commonly produce proofs [12, 8], and in 2017 the MIP solver SCIP [2] gained the ability to produce machine-checkable proofs for a subset of techniques [16].

Unfortunately, proof logging is far from standard in CP due to several challenges. First, CP solvers consider heterogeneous constraints and reasoning capabilities, making it difficult to select a suitable proof system. Second, as with proof logging for MIP and SMT solvers, the overhead of proof logging may be significant. Third, when models are distributed through standardized file formats, the internal representation of the solver could differ from those models. As a result, none of the state-of-the-art CP solvers submitted to the MiniZinc Challenge [44] and XCSP Competition [5] in 2023 supported proof logging in any form.

The current state of affairs is that proof logging is useful for CP and shows great promise, however, its mainstream adoption is clearly out of reach for current methods. An unpublished approach [27] suggests logging every nogood and propagation step, incurring a three-fold slowdown with potentially prohibitive disk space requirements. Another recent approach proposes to trace the solver using pseudo-Boolean reasoning [31], which leads to high confidence in each individual step of the solver as well as the final claim. Unfortunately, similar downsides hold, with slowdowns of (several) orders of magnitude, and even longer proof checking. Furthermore, both approaches log with respect to the internal problem representation of the solver, rather than a standardised input format.

We address the prohibitive runtime and space requirements for CP proof logging by contributing a novel CP proof generation *framework*, together with a proof format and proof checking facilities. Our approach has three steps. In the first step, the CP solver produces a *proof scaffold* of unsatisfiability by only logging nogoods (learned clauses), which can be understood as a compressed version of the final proof. Once the solving and scaffold production is done, in the second step, our proof processor (a separate piece of software) identifies nogoods in the scaffold that can be removed without invalidating the proof, and expands the trimmed scaffold into a full CP proof by adding only the necessary inference steps based on the remaining nogoods. In the third step, the proof is verified using one of the available proof-checking approaches. Our framework has several major advantages:

- From the solver (developer) side, we only require generating a proof scaffold, which introduces a minor overhead (5-10% runtime), requires little memory (scaffolds are at most hundreds of megabytes, with most expanded proofs well below one GB – compared to tens or hundreds of GBs of current approaches), and places virtually no burden on the solver developer. All of these points are in stark contrast with previous approaches.
- Our approach reduces the proof size by removing redundant nogoods *before* expanding the proof, which in our experiments removes 20-90% of the nogoods. This is significant given that verification runtime is a bottleneck, which is correlated with proof size.
- The resulting proof is in our *CP proof format*, which, in the spirit of CP, embraces the diversity of constraints and records the proof steps in CP terminology, rather than encoding them into a different paradigm. This allows our approach to be agnostic to the verification technology. In our implementation, we verify the proof using VeriPB [30].
- The proof verification is done with respect to the *input file*, rather than the internal solver representation, decoupling proof generation from verification. In our implementation, we use FlatZinc as the input file, although XCSP and other formats may be used similarly.

We demonstrate the effectiveness of our framework by experimentally evaluating against over 4000 instances from the MiniZinc benchmark repository. These are realistic problems and many have been used in previous competitions. Furthermore, we discuss proof logging FlatZinc components with pseudo-Boolean reasoning, and introduce a novel pseudo-Boolean justification technique for the cumulative global constraint.

To summarise, we present a framework for certifying the output of CP solvers that is 1) efficient enough to be used on realistic problems with little overhead, 2) agnostic of the solver, as the proof steps are given with respect to the input file rather than the internal solver representation, and 3) generic in the sense of our CP proof format which enables using any existing verification approach and reduces the verification time by proving shorter proofs. We believe our results demonstrate that proof logging has the potential to become an integral part of the CP community with a clear value in increasing the trustworthiness of our solvers.

The paper is organized as follows: Section 2 covers some preliminaries and Section 3 covers related work. In Section 4 we present our proof logging approach, and Section 5 presents an empirical evaluation of the approach. Finally, Section 6 gives our conclusion and presents directions for future work.

2 Preliminaries

2.1 Constraint Satisfaction Problems

A *constraint satisfaction problem* (CSP) $\mathcal{P} = (\mathcal{X}, \mathcal{C}, \mathcal{D})$ is a triple where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of decision variables, $\mathcal{D}$ is the domain, which, for each variable $x_i \in \mathcal{X}$, denotes the set of possible values for x_i as $\mathcal{D}(x_i)$. We consider all variables to be integer variables. Booleans are modelled as integers with domain $\{0, 1\}$, where 1 is the true value, and 0 is the false value. A constraint $C(X) \in \mathcal{C}$ is an n-ary relation between the variables $X \subseteq \mathcal{X}$. *Reified* constraints are of the form $r \leftrightarrow C(X)$, where $r \in \mathcal{X}$ is a Boolean variable which is used to conditionally enforce the constraint $C(X)$. A specific constraint that is important to this work is the *clause*, which is a disjunction of Boolean variables. An *assignment* A is a mapping from each variable x_i to a single value $v_i \in \mathcal{D}(x_i)$. If A satisfies all the constraints, then A is called a *solution.* If there are no solutions to a CSP $\mathcal{P}$, then $\mathcal{P}$ is called *unsatisfiable.* Otherwise, if $\mathcal{P}$ has at least one solution, the problem is called *satisfiable.* A CSP can be extended to a *constraint optimisation problem* (COP) by adding an objective function $f : \mathbb{Z}^n \mapsto \mathbb{Z}$ which maps any solution A to a cost $f(A)$ such that a solution A_1 is better than solution A_2 iff $f(A_1) < f(A_2)$. A solution A is *optimal* if there does not exist a solution A' with a lower cost.

2.2 Constraint Programming

Constraint Programming (CP) is a paradigm used to solve CSPs and COPs. In a CP solver, constraints are represented through propagators. A *propagator* is a function $p : \mathcal{D} \mapsto \mathcal{D}'$ that removes values from the domain $\mathcal{D}$ which cannot be present in any solution. *Fixpoint propagation* (FP) is the process of applying all propagators until fixpoint, i.e. until no more values can be removed. Solvers recursively partition the domain in at least two subproblems over which it calls fixpoint propagation. This process continues until a subproblem is found where all variables have singleton domains, or the solver concludes no solution exists.

An *atomic constraint* is a predicate $\langle x \diamond v \rangle$, where x is an integer variable, $\diamond \in \{\leq, \geq, =, \neq\}$, and v is a constant. During the search, CP solvers may learn *nogoods*, i.e., a conjunction of atomic constraints that cannot be part of a feasible solution. In this work, we will use the term *learned clause* to mean the negation of a nogood, following conventions from lazy clause generation CP [25, 40] and CDCL-SAT solving [38]. The set of all clauses in the solver is called the *clause database.* The propagation of nogoods/clauses to fixpoint is referred to as *Boolean constraint propagation* (BCP). The notation $\mathcal{C} \vDash_{BCP} l$ means that the atomic constraint l is entailed by the set of clauses $\mathcal{C}$ and this implication is identified through BCP.

2.3 Proof Logging for Black-Box Solvers

In this work, we are primarily concerned with unsatisfiability claims, since a proof of optimality can be understood as an unsatisfiability proof claiming that there is no solution with a lower objective value than the optimal value. This applies directly to linear search (branch-and-bound) commonly used in CP solvers, and a similar approach could be used for other optimisation approaches such as core-guided search [26] and logic-based Benders decomposition [21, 37] (known as implicit hitting sets in MaxSAT [20]).

Conceptually, a proof of unsatisfiability $P = [S_1, \ldots, S_n]$ is a sequence of proof steps, such that each step $S_i \in P$ is entailed by the model and all preceding steps $S_j : 1 \leq j < i$. Every step should be checkable in polynomial time of the number of steps preceding it. The final step of the proof $S_n = \bot$ is the trivial conclusion that the model is unsatisfiable. For solvers, the proof is generally a log of the reasoning steps performed during the search. Note that if a solver finds a solution to the model, it will still have logged its proof steps.

Given a model F and a proof of unsatisfiability P, a *proof checker* certifies that every step in the log is a valid reasoning step. If all the proof steps are correct, then the proof is valid and the model is confirmed to be unsatisfiable. Note that the checker merely asserts the correctness of the proof, and does not provide any guarantees on how the proof was generated. In this sense, an incorrect solver may still produce a correct proof, and as long as the proof is correct, the checker will accept it and assert the claim.

2.4 Proof Logging for SAT

In propositional proofs of unsatisfiability produced by SAT solvers, every step in the proof is a clause which is entailed by the conjunction of the clauses in the problem and the previous clauses in the proof. The entailment has to be identified through *reverse unit propagation*

(RUP) [33, 28], i.e. $\neg c \wedge F \wedge C \vDash_{BCP} \bot$ for a proof clause c, its predecessors in the proof C and the original formula F. The last clause is the empty clause, which cannot be satisfied and is therefore equivalent to $\bot$. Proofs of this style are referred to as RUP proofs. They have the advantage that it is not necessary to indicate what premises lead to the derivation of the clause c. However, a disadvantage is that the runtime cost of checking every clause grows as the number of clauses in the clause database grows.

As the SAT solver runs, it learns many RUP clauses using the CDCL solving framework [38] and adds the clauses to its database. Occasionally, a portion of the learned clauses are deleted. This deletion is often also logged in the proof, extending the RUP format to DRUP (Deletion Reverse Unit Propagation) [35]. The checker can use the deletion information to remove the clause from consideration, to reduce its memory usage and runtime cost. Note, DRUP has been superseded by the modern DRAT [46] and LRAT [19] formats in SAT solvers.

Proof checkers for SAT proofs check the RUP property for every clause in the proof. To do so, there are two possible directions the proof can be traversed: *forward checking* or *backward checking* [33]. In forward checking, the proof is read in one clause at a time. When reading clause C_i, the RUP property is checked and if it holds, C_i is added to the clause database for the checker to use at clause C_{i+1}. Backward checking takes the opposite approach. The entire proof is read into the checker and it starts by checking RUP on the empty clause and moving toward the first clause in the proof. When checking clause C_i, it is first removed from the clause database and then the RUP property is checked. All the clauses from the proof that are used to conclude $\bot$, are marked. Then, as the checker comes across clauses that are not marked, they can be removed from the clause database *without* checking the RUP property, as that clause did not contribute to the derivation of any clause used to derive the empty clause. Essentially, a backward checking proof checker trims the proof it is checking, only verifying proof steps that are used to derive other proof steps.

3 Related Work

Proof logging has been successfully developed for SAT solvers over the last 20 years, and is now adopted by virtually all modern SAT solvers. The introduction of proof logging surfaced many bugs in solver implementations, and the SAT competition, a yearly competition between SAT solvers, requires solvers to be proof-producing. Early proofs were resolution-based proofs [47], but since then the common proof systems are so-called clausal proofs [35, 46, 19].

SMT solvers like Z3 [10] and cvc5 [7] can also produce proofs. These systems are similar to CP solvers in that they have an extremely large number of possible reasoning steps they perform. For any new theory, the reasoning must be individually justified in the proof. The proofs are prohibitively large since the solvers perform a vast amount of reasoning which turns out to be irrelevant to derive unsatisfiability.

A third solving paradigm which has seen work in proof logging is the MIP paradigm. VIPR [16] is a format for MIP proofs using cutting planes reasoning. It is implemented for the SCIP [2] solver. When logging, SCIP has to run with exact arithmetic, which impacts the performance already. On top of that, the large I/O overhead means that the impact of logging the proof limits its applicability.

The first implementation of a proof-producing CP solver was done in the PCS/HaifaCSP solver [45], which produced resolution proofs. For each propagation, the reasoning is explicitly stated in terms of the premises and the conclusion in clausal form to perform the resolution steps. It is unclear what the experimental impact is of this proof logging implementation.

Another promising approach implements proof logging into the Chuffed LCG solver [27]. The proof format can be seen as an extension of the DRUP format, with propagations introduced as temporary clauses. The work is not published, and their experimental data shows a threefold slowdown when proof logging is enabled with potentially large proof files.

A more recent approach to proof logging for CP is done in the Glasgow CP solver [31], which encodes propagations and conflicts into pseudo-Boolean form. This is an elegant approach that decouples the proof format from the capabilities of the CP solver, and new propagators can be supported by devising an encoding of the filtering algorithms into pseudo-Boolean form [24]. It does require the encoding into pseudo-Boolean form [14] to become part of the trusted base of the application, as that component is not formally checked. The key issue, however, is that it imposes a (several) orders-of-magnitude runtime overhead on the solver, as encodings are created during the solving process, and since every propagation is logged, the proof files easily exceed tens and hundreds of GB even for small problems which result in even longer verification times. Although engineering efforts may bring down the runtime, two main issues hinder scalability: 1) CP solvers spend most of their time in the propagation engine, so adding additional work during propagation is likely to negatively impact performance, and 2) only a portion of the work done by the solver is needed for the proof of unsatisfiability. Our framework addresses these issues by removing logging during propagation via scaffolds and expanding only the necessary part of the proof before checking.

Previous CP proof logging approaches show the value of proof logging CP solvers, however, all have similar drawbacks with varying degrees: considerable overhead for the solver, large file requirements to store the proofs, and the proof is done in terms of the internal solver representation rather than a standardised input format.

Finally, ensuring the correctness of solvers has been investigated by correct-by-construction methods [15] using Coq, an automated theorem prover. This removes the necessity of proof checkers, however, the solver is not competitive with implementations in imperative languages, as many of the implementation strategies are difficult to express in formalisms like Coq.

4 Our Contribution: A Multi-Stage Framework for CP Proof Logging

Our framework has three stages: proof scaffolding, proof processing, and proof verification. Before diving into the details, we first show two illustrative examples.

▶ **Example 1.** On the left, an unsatisfiable CSP is given. The table on the right shows one possible proof for why the CSP is unsatisfiable. Note that the there is another, shorter proof that could have been found.

$$
\begin{aligned}
x, z \in \{0,1\}, y &\in \{0,1,2\}\\
c_1 : 2x + y + 2z &\geq 2\\
c_2 : 2x + y - 2z &\geq 0\\
c_3 : 2x - y + 2z &\geq 0\\
c_4 : 2x - y - 2z &\geq -2\\
c_5 : -2x + y + 2z &\geq 2\\
c_6 : -2x + y - 2z &\geq 0
\end{aligned}
$$

#	Implied by	Proof step
1	c_3	$\langle x \leq 0\rangle \wedge \langle y \geq 2\rangle \rightarrow \langle z \geq 1\rangle$
2	c_4	$\langle x \leq 0\rangle \wedge \langle y \geq 2\rangle \rightarrow \langle z \leq 0\rangle$
3	1, 2	$\langle x \leq 0\rangle \wedge \langle y \geq 2\rangle \rightarrow \bot$
4	c_1	$\langle x \leq 0\rangle \wedge \langle y \leq 1\rangle \rightarrow \langle z \geq 1\rangle$
5	c_2	$\langle x \leq 0\rangle \wedge \langle y \leq 1\rangle \rightarrow \langle z \leq 0\rangle$
6	3, 4, 5	$\langle x \leq 0\rangle \rightarrow \bot$
7	c_5	$\langle x \geq 1\rangle \rightarrow \langle z \geq 1\rangle$
8	c_6	$\langle x \geq 1\rangle \rightarrow \langle z \leq 0\rangle$
9	6, 7, 8	UNSAT

▶ **Example 2.** Let us consider the three-queens problem. The problem states that we have to place three queens on a three-by-three chessboard, such that no two queens are in the same row, column, or on the same (anti)diagonal. On the left is a model for the three-queens problem, and on the right a proof for why it is unsatisfiable.

$$
\begin{aligned}
q_1, q_2, q_3 &\in \{1, 2, 3\} \\
c_1 &: \texttt{Distinct}(q_1, q_2, q_3) \\
c_2 &: \texttt{Distinct}(q_1 + 1, q_2 + 2, q_3 + 3) \\
c_3 &: \texttt{Distinct}(q_1 - 1, q_2 - 2, q_3 - 3)
\end{aligned}
$$

#	Implied by	Proof step
1	c_1	$\langle q_1 = 1\rangle \rightarrow \langle q_2 \neq 1\rangle$
2	c_1	$\langle q_1 = 1\rangle \rightarrow \langle q_3 \neq 1\rangle$
3	c_2	$\langle q_1 = 1\rangle \rightarrow \langle q_2 \neq 2\rangle$
4	c_1	$\langle q_2 = 3\rangle \rightarrow \langle q_3 \neq 3\rangle$
5	c_3	$\langle q_2 = 3\rangle \rightarrow \langle q_3 \neq 2\rangle$
6	1-5	$\langle q_1 = 1\rangle \rightarrow \bot$
7	c_1	$\langle q_1 = 2\rangle \rightarrow \langle q_2 \neq 2\rangle$
8	c_2	$\langle q_1 = 2\rangle \rightarrow \langle q_2 \neq 3\rangle$
9	c_3	$\langle q_1 = 2\rangle \rightarrow \langle q_2 \neq 1\rangle$
10	7-8	$\langle q_1 = 2\rangle \rightarrow \bot$
11	c_1	$\langle q_1 = 3\rangle \rightarrow \langle q_2 \neq 3\rangle$
12	c_1	$\langle q_1 = 3\rangle \rightarrow \langle q_3 \neq 3\rangle$
13	c_2	$\langle q_1 = 3\rangle \rightarrow \langle q_2 \neq 2\rangle$
14	c_1	$\langle q_2 = 1\rangle \rightarrow \langle q_3 \neq 1\rangle$
15	c_3	$\langle q_2 = 1\rangle \rightarrow \langle q_3 \neq 2\rangle$
16	6, 10-15	`UNSAT`

In Example 2 the solver can derive nogoods (steps 6 and 10) based on the propagations it performs. In the example, these nogoods are all singleton, i.e. they contain one atomic constraint. However, nogoods can contain an arbitrary number of atomic constraints, as shown in Example 1.

We proceed with describing each of our stages, referring to these examples as appropriate.

4.1 Verification Flow

One of the main barriers to wider adoption of CP proof logging is the significant potential overhead on the solver in terms of runtime and memory, and possibly the effort required from the solver developer to enable proof logging. Our aim is to address both of these concerns.

Since proof production may have a significant overhead on the solving process, we aim to log as few steps as possible during the search. As a CP solver can potentially log many proof steps, especially when logging individual propagations, proofs can grow prohibitively large. It may also be that the overhead of logging turns out to be wasted, for example when a satisfaction problem turns out to be satisfiable or when the solver never concludes optimality in an optimisation problem. We therefore aim to log as few steps as possible during the solving phase to keep the runtime cost introduced by proof logging as low as possible.

Given this motivation, we split the proof production into multiple phases. Note that while each phase may introduce bugs, a sound checker will only ever accept valid proofs, and bugs due to unsound reasoning will be detected. A schematic overview of the following steps in proof production is given in Figure 1.

1. **Scaffolding** The first step is to run the solver. During the solving process, the solver produces a *proof scaffold*, which is a list of nogoods identified by the solver.

   The proof scaffold of Example 2 would consist of two single-predicate nogoods, $\langle q_1 \neq 1\rangle$ (step 6) and $\langle q_1 \neq 2\rangle$ (step 10), and the unsatisfiability claim (step 16). For Example 1, the proof scaffold would consist of proof steps 3, 6, and 9.

   The requirement is that the i-th proof step can be recovered by contradiction in polynomial time by the next stage considering the original problems and the first $(i-1)$ proof steps (similar to RUP), i.e., negating the nogood leads to a conflict by propagation.

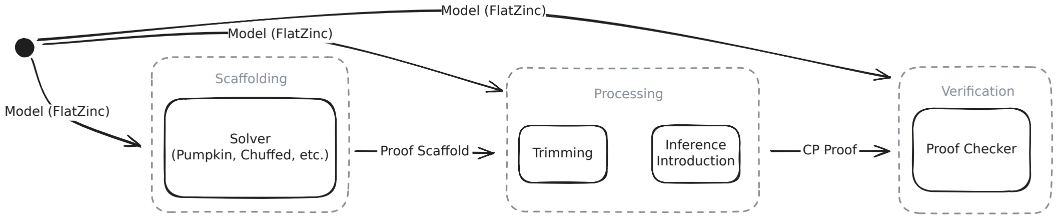


Figure 1 Schematic overview of our multi-stage framework for CP proof logging.

2. **Processing** The proof scaffold is completed into a full proof by the proof processor. Here we describe what the processing step provides to the framework, and Section 4.3 presents the algorithm to achieve it. Conceptually, there are two transformations the processor runs on the scaffold:

   Trimming A proof scaffold will likely contain many nogoods that are unnecessary to derive unsatisfiability. For SAT solvers, proof trimming is a vital part of proof checking [35], and our experiments in Section 5 show the same holds for CP proofs. Given that the next step can potentially introduce many steps to the proof, the key idea is that we trim before inference introduction.

   In the presented three-queens example, trimming would not remove any of the two nogoods in the proof. However, the solver could also have derived the nogood $\langle q_2 \neq 1\rangle$, meaning the proof scaffold would have been $\langle q_1 \neq 1\rangle, \langle q_2 \neq 1\rangle, \langle q_1 \neq 2\rangle$. Had this been the case, then trimming would have removed the middle nogood since it is redundant in proving unsatisfiability.

   Inference Introduction For every nogood C in the trimmed proof scaffold, the necessary inferences (i.e. propagations) relevant to concluding C are generated. The output of the inference introduction stage is considered to be a full CP proof. It explains, in CP terms, the reasoning steps required to prove unsatisfiability.

   Note that this implies two points. First, the processor needs to be able to reason at least as strongly as the solver to accept the proof scaffold as a valid proof. Second, the processor is free to provide reasoning that is different from the reasoning used by the solver used during runtime, as long as it reaches the same conclusion.

   In Example 2 and Example 1, the introduced inferences are the proof steps which are implied by constraints from the model, rather than any previous proof steps.

3. **Verification** After proof processing we get what we refer to as the full *CP proof*. It is ready to be checked by a proof checker. Ideally the checker is formally verified, to leave as little doubt in the conclusion as possible.

4.2 Proof Format

The proof format we propose reasons about the CP model in terms of atomic constraints and nogoods over those atomic constraints. The format is valid for both the proof scaffold and the full CP proof. A step in a valid CP proof in this format is one of the following:

Nogood ⟨nogood id⟩ ⟨clause⟩ A clausal representation of a nogood, which satisfies the RUP property. In a solver which derives nogoods using the CDCL [43] algorithm, all nogoods satisfy the RUP property. A trivial nogood which can be logged by non-learning CP solvers consists of current decision variable assignments when a conflict is detected. In Example 2, steps 3 and 6 are nogoods.

Deletion ⟨nogood id⟩ The nogood with the given id is no longer relevant for BCP, and can be removed from consideration in the checker. This step is required to reduce the memory usage and BCP runtime. In an LCG solver, as in SAT solving, it is common to occasionally delete nogoods that are deemed unworthy by the solver to save space and propagation time. Therefore, such nogoods cannot contribute to later propagations (unless they are learned again), and therefore we can help a proof checker by indicating the deletion.

Inference ⟨name⟩ ⟨premises⟩ ⟨propagated atomic constraint⟩ A lemma which is enforced by a particular propagator in the solver. It states the premise that leads to the given propagated atomic constraint. The inference may be used in checking the RUP property of a nogood. For instance, in Example 2, step 1 is an inference given by constraint c_3, and it is used when checking the nogood in step 3.

The inference is required to satisfy an extension of the RUP property, i.e., asserting the premises and the negation of the propagated atomic constraint leads to a conflict by *fixpoint propagation*. Given this requirement, the "name" is not strictly necessary – as in the scaffolding phase, this could be left freely to the processor to determine the reason. Nevertheless, the name serves as a hint to simplify the verification step and in our evaluation, the name was necessary to select the proper encoding for the inference.

Typically a CP solver performs many propagations during search. Keeping all inferences in the proof in memory would slow down BCP when checking a nogood. Hence, we specify that all inferences only apply to the first nogood following it. After checking that nogood, the inferences are deleted from memory. This means the same inference may be present multiple times in the proof file, but it avoids having to delete inferences explicitly.

Conclusion ⟨conclusion⟩ The final step of the proof. The value of ⟨conclusion⟩ can be one of the following:

- `UNSAT` when the conclusion is that the problem is unsatisfiable,
- a literal corresponding to the atomic constraint $\langle objective \geq v\rangle$, where v is the value of the objective variable in the optimal solution.

Conceptually, this format extends the DRUP [33] format. The nogoods in the proof are the RUP clauses which are checked identically to the RUP clauses in a DRUP SAT proof, and the inferences are temporary problem clauses which are created by the propagators in the CP solver. An interesting property of our approach is that for pure SAT problems, i.e., problems containing only clauses and Boolean variables, our approach resembles standard SAT proof logging and introduces no overhead compared to SAT proof logging, contrary to other CP proof logging approaches.

4.3 Proof Processor

Given a CP proof scaffold, the processing stage can now apply trimming and inference introduction. The proof processor uses a backward checking approach [33] to implement trimming and inference introduction in a single pass.

The pseudo-code for the proof processor is given in Algorithm 1. The proof is loaded, the empty clause is marked as used, and the output proof P' is initialized to an empty sequence. The proof processor then goes over the scaffold back to front. The last clause is popped from the input proof sequence, and if it is marked, the RUP property is checked. Unmarked clauses are skipped immediately. The Propagation(φ) procedure propagates all constraints until the fixpoint. In case of no conflict, the implication graph is empty and the checker cannot complete the proof scaffold, as the producer of the scaffold performed reasoning the

Algorithm 1 Proof Processing. F is a set of constraints, and P is the proof scaffold.

```
Require: The top-most element of $P$ is $\emptyset$.
Ensure: $P'$ is the full CP proof.
$\varphi \leftarrow F \cup P$
$marked \leftarrow \{peek(P)\}$          ▷ $peek$: Get top-most element in $P$ without popping
$P' \leftarrow \emptyset$
while $|P| > 0$ do
    $c \leftarrow pop(P)$
    $\varphi \leftarrow \varphi \setminus \{c\}$
    if $c \in marked$ then
        $P' \leftarrow push(P', c)$
        $confl \leftarrow$ PROPAGATION($\varphi \wedge \bigwedge_{l \in c} \neg l$)          ▷ Perform the RUP check.
        if $confl$ is empty then
            return Fail
        end if
        $nogoods, inferences \leftarrow$ ANALYZE($confl$)
        $marked \leftarrow marked \cup nogoods$
        $P' \leftarrow P'$ concatenated with $inferences$
    end if
end while
$P' \leftarrow$ REVERSE($P'$)
return Success
```

processor cannot imitate. In case of a conflict, the RUP property for the nogood holds and the Propagation(φ) procedure returns an implication graph describing the conflict. The Analyze(G) procedure walks the implication graph backwards. Through this traversal, the nogoods used to derive the conflict are marked, and the propagations performed by propagators are introduced as inferences. Finally, after processing all the nogoods in the scaffold, the output proof P' contains the full CP proof in reverse, so it is reversed.

4.4 Checking Proofs

Once we obtain the full CP proof, the final step is to check the proof with respect to the model. A possible approach is implementing a formally verified proof checker with a theorem prover. This approach was suggested in previous work [27] and would fit the framework well. Another approach, which we adopted in our evaluation, is to encode the CP proofs to a pseudo-Boolean format and check the proofs with VeriPB, following the Glasgow solver [31].

For this paper, the description of a CP model is given in the FlatZinc format. Therefore, the models need to be encoded to a pseudo-Boolean formula. Then, every proof step in the CP proof is encoded to an equivalent pseudo-Boolean justification.

4.4.1 FlatZinc Encoding

Encoding the FlatZinc model into a pseudo-Boolean formula entails encoding the variables, the constraints, and, if present, the objective variable. Following the narrative from [31], we encode the FlatZinc variables with a binary encoding. For example, if variable x has a domain $D(x) = [0..5]$, the binary encoding will have three Boolean variables $a_3a_2a_1$, and the binary expansion of x is $1 \times a_1 + 2 \times a_2 + 4 \times a_3$. Next, we focus on three types of constraints in this work:

Linear Inequalities A linear inequality is a constraint in the form $\sum w_i x_i \geq c$, where $w_i, c \in \mathbb{Z}$ are integer constants and $x_i \in \mathcal{X}$ are integer variables. These constraints are encoded as described in [31].

Reified Linear Inequalities A reified version of the linear inequality constraint. We rewrite constraints of the form $r \leftrightarrow \sum w_i x_i \geq c$ as a big-M constraint in the pseudo-Boolean model; more formally, this constraint is introduced as $M\neg r + \sum w_i x_i \geq c$ and $Mr - \sum w_i x_i \geq 1 - c$, where M is a suitably large integer constant. Encoding this constraint is now the same as encoding a regular linear inequality. Additionally, any justification for an inference implied by a reified linear inequality is rewritten to a linear bounds justification over the big-M constraint.

Cumulative To encode the cumulative constraint [3] we used Equations 2–3 from Bofill et al. [11].

Finally, if the FlatZinc model defines an objective variable $\mathcal{O}$, the FlatZinc objective function is given as the binary expansion of $\mathcal{O}$.

4.4.2 Atomic Constraint Encoding

The pseudo-Boolean encoding of the proof starts by introducing the atomic constraints as pseudo-Boolean variables. We achieve this by reifying the corresponding condition with redundance-based strengthening [32]. For example, an atomic constraint $\langle x \geq v \rangle$ is encoded by introducing a new pseudo-Boolean variable (which we introduce with the same notation) and deriving two big-M constraints $M\neg \langle x \geq v \rangle + x \geq v$ and $M \langle x \geq v \rangle - x \geq -v + 1$ for large enough values of M.

Given the semantics of integer variables and values, every proof *implicitly* includes the constraints enforcing the consistency of the atomic constraints, including $\forall x \in \mathcal{X}, v \in \mathbb{Z} : \langle x \geq v \rangle \rightarrow \langle x \geq v + 1 \rangle$ or $\forall x \in \mathcal{X}, v \in \mathbb{Z} : \langle x \leq v \rangle \rightarrow \langle x \leq v - 1 \rangle$. Therefore, after declaring the atomic constraints, we introduce the constraints above with VeriPB inferences. For example, a condition $\langle x \geq u \rangle \rightarrow \langle x \geq v \rangle$ for constants u, v such that $u \geq v$ is introduced by taking the half-reification definitions of form $M\neg \langle x \geq u \rangle + x \geq u$ and $M \langle x \geq v \rangle - x \geq -v + 1$, adding them and dividing with rounding up by the big-M constant. The resulting condition reads as $\neg \langle x \geq u \rangle + \langle x \geq v \rangle \geq 1$, which is precisely the desired implication.

4.4.3 Proof Encoding

After introducing the atomic literals, the encoder unpacks each CP proof step into valid VeriPB statements. The nogood and deletion steps correspond to VeriPB RUP and deletion steps respectively. Hence, we only discuss the encoding of the inference steps.

To support the constraints listed in Section 4.4.1 we have to justify two kinds of inferences: linear bound reasoning for linear inequalities, as used in Example 1, and time-table [9] reasoning for the cumulative constraint. As stated previously, reified linear inequalities are rewritten to regular inequalities and the justification for reified linear inequality inferences is regular linear bound reasoning. A pseudo-Boolean justification of inferences implied by linear inequalities is described by [31]. Hence, we only describe the time-table justification here.

Let $\langle s_i \geq v \rangle \land \langle s_i \leq v' \rangle \rightarrow \langle s_j \geq w \rangle$ be a time-table inference, where variables s_i and s_j model the start time of activities i and j, respectively. This inference states that the bounds on variable s_i are such that activity j cannot be scheduled before time point w, because, task i must consume a certain amount of resource in an interval ending at time point $w - 1$ and scheduling variable s_j before w would guarantee to overload the resource.

The justification is done by contradiction, i.e., we assume activity j is scheduled before time point w. From the encoding of the constraint, we have Boolean literals corresponding to $\langle s_i \leq t < s_i + p_i \rangle$ for every time point t, where $p_i \in \mathbb{Z}$ is the constant duration of activity i. These literals indicate that task i is active at time point t. The premises of the inference correspond to setting these literals to true for the time points within the bounds described by the inference. The negation of the propagated atomic constraint also leads to these literals becoming true in the time interval ending at time point w. Pseudo-Boolean propagation then identifies the conflict in resource usage, so the inference is valid.

Finally, when the proof conclusion is the optimal objective value, the encoder will first identify the lower bound $\langle x \geq v \rangle$ on the objective variable x from the conclusion of the proof. The constraint $\neg \langle x \geq v \rangle$ is then appended to the encoding of the FlatZinc model. The rest of the proof is treated as proof of unsatisfiability for the amended model.

5 Evaluation

To show our method works on a standardised input format, we use MiniZinc [39] models. We designed experiments to evaluate our framework in three aspects:

- The overhead caused by logging the scaffold in terms of runtime. We aim to keep this low, to increase the practical use of the framework. We show that logging takes most of the time less than 10%, which is orders of magnitude lower compared to existing approaches.
- The value of scaffolding, trimming, and expansion, compared to logging all proofs steps as in other approaches. Processing the proof should not be prohibitively expensive in runtime and disk space. We demonstrate that our approach is indeed computationally inexpensive with notable reductions in proof size.
- The time spent to verify the proofs. For proof logging to be adopted, verifying the proofs should not take an unreasonable amount of time. Our current runtimes are promising, in particular for optimisation problems.

We implemented our approach in our LCG CP solver Pumpkin[1]. We support most FlatZinc primitives and the cumulative global constraint. The solver is competitive to established solvers Gecode [42] and Chuffed [17] (see Figure 2) when using the globals constraints that we support and decomposing others. This gives credibility to our results.

We use our solver within the proof processor. Note that while proofs may be potentially produced with incorrect solvers, the final proof is verified with an external tool which gives us high confidence in the results, e.g., the solver is not checking itself.

Our main focus is on proof production, nevertheless we also implemented our own checker as a proof-of-concept. The checker is based on VeriPB, and certifies that the proof is correct with respect to the FlatZinc file. Our checker implements a subset of FlatZinc primitives and the cumulative with time-table filtering, as described in Section 4.4.

We considered all MiniZinc benchmarks comprising over 4000 instances, however depending on the experiment, we used a different subset (around 3000 instances). These are detailed in the appropriate sections.

All the experiments were run on the DelftBlue [23] supercomputer. Each benchmark used a single core of an Intel Xeon E5-6248R 24C 3.0GHz processor and 12GB of RAM. The solver was given 10 minutes, and subsequent steps in the framework were only performed on instances that did not time out. Any time measurements given are of CPU time.

[1] Available at `https://github.com/consol-lab/pumpkin`. The version used for experiments is stored on Zenodo (see title page).

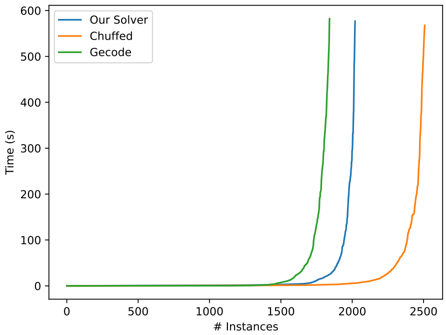


Figure 2 A cactus plot of Gecode, Chuffed and our solver on instances of the MiniZinc benchmarks. Only completely solved instances are shown.

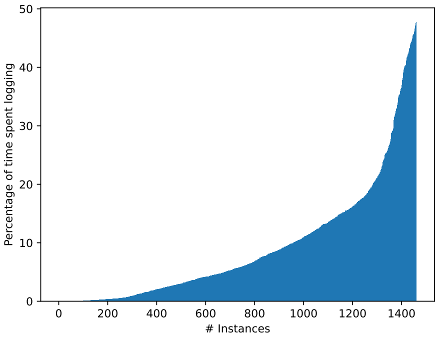


Figure 3 Distribution of logging as a percentage of the solving time for all the instances taking longer than 10 seconds.

5.1 Proof Logging Overhead

The solver was instrumented to measure the CPU time spent within the proof logging. Given the capabilities of our solver, we considered 3923 MiniZinc benchmarks. Note that we reduced the number of instances for the `mrcpsp` and `rcpsp` families by 80% by removing instances randomly, since these instances had an overwhelming presence compared to other benchmark families. The comparison with other solvers was done on this set of benchmarks as well.

Table 1 The time spent logging as a percentage of the total runtime. After discarding instances solved under 10 seconds, 1462 instances remain.

Minimum	Median	Geometric Mean	Mean	Maximum
0.01%	5.70%	3.29%	9.33%	47.78%

We discarded easy problems from the evaluation which were solved within ten seconds, leaving us with 1462 instances. Figure 3 shows the distribution of the overhead, and this information is aggregated in Table 1. Logging takes, on average, less than 10% of the runtime. This is a significant improvement compared to the orders of magnitude slowdown reported by previous methods. Even the worst case only presents a 47.78% overhead, but that is on an instance which finishes so quickly the I/O is the main bottleneck. We note that the overhead can be notably reduced by using a better implementation when writing to the file, e.g., we produce plain text files, whereas outputting in binary form would be more efficient. Nevertheless the logging time even in the current form remains low.

Our experiments highlight the advantage of this approach, and the reason for logging only the derived nogoods becomes clear. Every derived nogood typically requires many propagations, and a slowdown by a multiple of what is presented here is inevitable when the propagations are logged as well during solving.

5.2 Proof Processing

Aside from proof production, important metrics in this evaluation are the cost of proof processing and the benefit in terms of proof size reduction. We focus on instances that we support with our checker that we could completely solve with our solver, i.e., where the solver concluded unsatisfiability or optimality. In total, we considered 3382 benchmarks and generated a total of 1825 proofs from three models from the MiniZinc benchmark repository[2]:

RCPSP: an optimisation model using the cumulative global constraint and linear inequalities. This model has 1784 instances which are solvable to optimality within 10 minutes.

2DPacking: an optimisation model with linear (in)equality constraints, with 10 instances solvable to optimality within 10 minutes.

Market Split: a satisfaction model with only linear equality constraints, and has 20 instances that are solved within 10 minutes and are unsatisfiable.

Figure 4a shows how many nogoods remain in the proof after trimming. The plot shows only 574 proof scaffolds since the remaining 1251 instances are solved with an empty scaffold, i.e., instances were determined unsatisfiable at the root level. It is notable that for 373 instances, from both RCPSP and 2DPacking, not a single nogood remains after trimming. The final proof only consists of inferences, and the trimmer used the bound on the objective to identify the required inferences and discarded all nogoods. This means that these instances with an infeasible lower bound on the objective variable can be identified through propagation alone, without any search. We note that these 373 instances had scaffolds ranging from a single nogood to 987,272 nogoods.

On the market split instances, we see different behaviour. For these instances, trimming did almost nothing. There is one instance for which 1% of the nogoods were removed, but on all the other instances not a single nogood could be trimmed away. Based on the impact of trimming on DRUP proofs for SAT solvers [35], we expected to see at least some impact. We hypothesise this is not the case because the instances are relatively small, all solved within 3 seconds. It is therefore easy to find the unsatisfiability, and the solver never ends up exploring redundant parts of the search space.

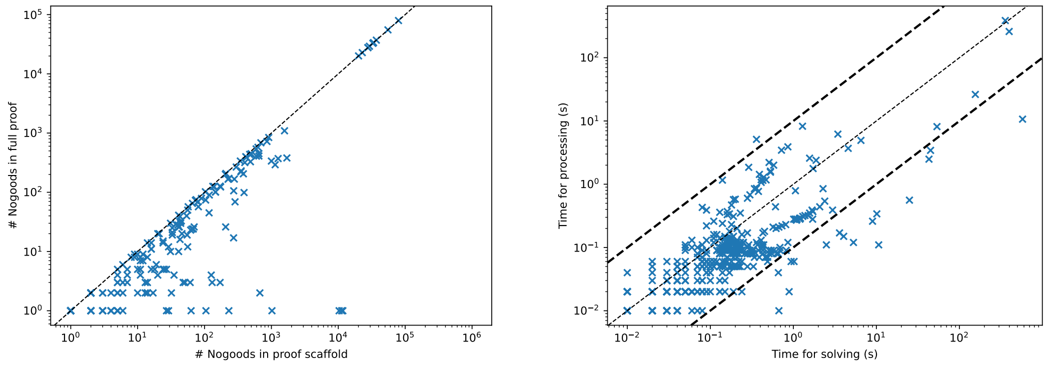
Given the benefits of the explicit processing step in terms of proof size, we now discuss the cost in terms of extra runtime. Figure 4b presents the time taken to process every proof compared to the time it took to solve that instance. From the figure we can see that processing time is between an order of magnitude quicker and an order of magnitude slower than solving the model, however, most of these orders of magnitude differences are due to very small runtimes. Out of the instances which take the longest to solve, all take less than 100% of the solving time to be processed. Since we are dealing with many instances which are solved quickly, and given that noise in the timing data is more pronounced on shorter time spans, we conclude that instances with long solving times give a good upper bound to the processing time. This aligns with our expectations: we expect processing time to take less time than solving for reasonably sized problems.

5.3 Proof Checking

As described in Section 4.4, we implemented an encoder for the proof so they can be checked by VeriPB. This means we spend some time encoding the proofs. Depending on their size, this encoding time is not negligible.

Encoding a proof is linear in the number of proof steps. Every step in the CP proof can be encoded independently from the others, and encoding a single step is cheap. On instances with only a few 100 propagations and few nogoods, the I/O required in the encoder dominates the time. However, on instances which take longer than 2 seconds to solve, the median encoding time is 77% of the solving time.

[2] Available at `https://github.com/MiniZinc/minizinc-benchmarks/tree/26bcd0a`

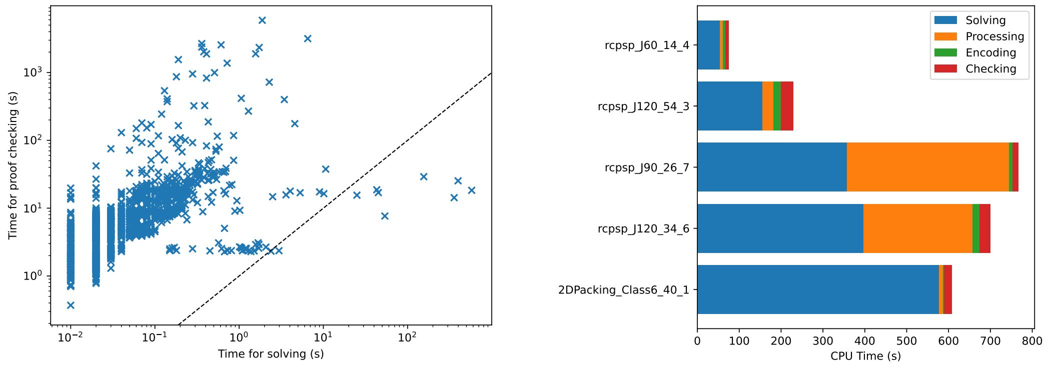


(a) Effect of trimming on the number of nogoods. This does not include the 373 instances which had no remaining nogoods after trimming.

(b) The time taken to process the proof compared to the time it took to produce the scaffold.

Figure 4 Experimental results on the processing of proofs.

The final step to fully certifying the solutions is to look at what VeriPB does with the encoded proofs. Figure 5a plots, for every proof, how long it took to verify relative to the time it took to produce the scaffold. For some instances, the checking time is orders of magnitude slower than producing the proof. In particular, this is the case for the market split instances. This is likely because for optimisation problems the solver performs upper-bounded linear search, which means it starts out deriving nogoods as it finds improving solutions. Only after the last solution does the solver start to conclude optimality, which means many of the first nogoods can be removed. For satisfaction problems, this does not happen, and therefore more nogoods from the scaffold are relevant, which means checking the proof takes longer relative to the solving time compared to optimisation instances.



(a) Time taken to check an instance compared to solving the instance with proof generation.

(b) Solving to checking for the five instances that took the longest to solve. This excludes market split instances because they are solved too quickly.

Figure 5 Experimental results on the checking of proofs.

To give an idea of the breakdown of the total time it takes for the proof to be accepted, starting with the solving time, Figure 5b shows the time taken at each stage for the five instances that took the longest to solve. We focus on the long instance since these instances are likely to run into scalability issues with the other methods. We note that this excludes any market split instances because they are all solved in less than 3 seconds, and at that scale, the timing data is extremely noisy. However, precisely these instances highlight why our method is practically usable. Four out of the five instances are solved in more than 150 seconds, and the `rcpsp_J60_14_4` instance is solved in approximately 53 seconds. All of the instances are optimisation problems, and again we can see that checking whether a claim is really true *can* be extremely cheap compared to solving the problem.

6 Conclusion and Future Work

We have presented a framework to support proof logging in CP solvers, given a FlatZinc model as an input. It addresses problems with large proof sizes and significant runtime overhead by delaying as much of the proof construction for as long as possible: only until the necessity of the proof becomes clear does one have to pay the price to obtain the proof. The proof format itself is designed to be almost arbitrarily extensible, in acknowledgement of the versatility of CP solvers and their propagation algorithms.

Through our framework, we can feasibly certify unsatisfiability and optimality claims in practical settings, with minimal overhead compared to when the solver is not logging a proof. This minimal overhead is not trivial to achieve. On optimisation models, solvers may derive many facts which do not directly relate to proving optimality, which adversely affects proof checking times. By removing the reasoning steps which are redundant for the optimality claim, at least RCPSP and 2DPacking optimality proofs, can be checked quickly.

Future work includes expanding the number of supported constraints in the MiniZinc library. These include robust implementations of the rest of the FlatZinc builtins and would be followed by implementing MiniZinc globals other than cumulative. Supporting stronger reasoning would likely drastically shorten the proofs which would benefit not just solving and processing times, but also checking times. Furthermore, we would like to expand support to other formalisms such as XCSP, engage with solver developers to further reduce practical barriers, and include other techniques such as preprocessing in the proof format.

Another avenue to pursue is whether the design and implementation of a dedicated, formally verified checker for the CP proof format can reduce the end-to-end overhead even more, which was shown to be promising [27]. If the proof no longer needs to be encoded, and the checker can implement CP-specific reasoning natively, we postulate the checking will become cheaper and more accessible.

References

1 Tobias Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin, October 2007.

2 Tobias Achterberg. SCIP: solving constraint integer programs. *Math. Program. Comput.*, 1(1):1–41, July 2009. doi:10.1007/s12532-008-0001-1.

3 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, $1^{ères}$ Journées Francophones de Programmation Logique, 25-27 Mai 1992, Lille, France*, volume 17, page 51, 1992. doi:10.1016/0895-7177(93)90068-A.

4 Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736, Cham, 2018. Springer. doi:10.1007/978-3-319-98334-9_46.

5 Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2023 XCSP3 competition. *CoRR*, abs/2312.05877, 2023. doi:10.48550/arXiv.2312.05877.

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

7 Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442, Cham, 2022. Springer. doi:10.1007/978-3-030-99524-9_24.

8 Clark Barrett, Leonardo De Moura, and Pascal Fontaine. Proofs in satisfiability modulo theories. *All about proofs, Proofs for all*, 55(1):23–44, 2015.

9 Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, pages 63–79, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-46135-3_5.

10 Nikolaj S. Bjørner, Clemens Eisenhofer, and Laura Kovács. Satisfiability modulo custom theories in Z3. In Cezara Dragoi, Michael Emmi, and Jingbo Wang, editors, *Verification, Model Checking, and Abstract Interpretation - 24th International Conference, VMCAI 2023, Boston, MA, USA, January 16-17, 2023, Proceedings*, volume 13881 of *Lecture Notes in Computer Science*, pages 91–105, Cham, 2023. Springer. doi:10.1007/978-3-031-24950-1_5.

11 Miquel Bofill, Jordi Coll, Josep Suy, and Mateu Villaret. Compact mdds for pseudo-boolean constraints with at-most-one relations in resource-constrained scheduling problems. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 555–562, 2017. doi:10.24963/ijcai.2017/78.

12 Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. verit: An open, trustable and efficient smt-solver. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-02959-2_12.

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

14 Sam Buss and Jakob Nordström. Proof complexity and sat solving. *Handbook of Satisfiability*, 336:233–350, 2021. doi:10.3233/FAIA200990.

15 Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. A certified constraint solver over finite domains. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 116–131, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-32759-9_12.

16 Kevin K. H. Cheung, Ambros M. Gleixner, and Daniel E. Steffy. Verifying integer programming results. In Friedrich Eisenbrand and Jochen Könemann, editors, *Integer Programming and Combinatorial Optimization - 19th International Conference, IPCO 2017, Waterloo, ON, Canada, June 26-28, 2017, Proceedings*, volume 10328 of *Lecture Notes in Computer Science*, pages 148–160, Cham, 2017. Springer. doi:10.1007/978-3-319-59250-3_13.

17 Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, a lazy clause generation solver. URL: `https://github.com/chuffed/chuffed/`.

18 William Cook, Thorsten Koch, Daniel E Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, 2013. `doi:10.1007/s12532-013-0055-6`.

19 Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236, Cham, 2017. Springer. `doi:10.1007/978-3-319-63046-5_14`.

20 Jessica Davies and Fahiem Bacchus. Exploiting the power of mip solvers in maxsat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 166–181. Springer, 2013. `doi:10.1007/978-3-642-39071-5_13`.

21 Toby O. Davies, Graeme Gange, and Peter J. Stuckey. Automatic logic-based benders decomposition with minizinc. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 787–793, 2017. `doi:10.1609/aaai.v31i1.10654`.

22 Sven De Vries and Rakesh V Vohra. Combinatorial auctions: A survey. *INFORMS Journal on computing*, 15(3):284–309, 2003. `doi:10.1287/ijoc.15.3.284.16077`.

23 Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 1). `https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1`, 2022.

24 Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-boolean reasoning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, volume 34, pages 1486–1494. AAAI Press, April 2020. `doi:10.1609/aaai.v34i02.5507`.

25 Thibaut Feydy and Peter J Stuckey. Lazy clause generation reengineered. In *International Conference on Principles and Practice of Constraint Programming*, pages 352–366. Springer, 2009. `doi:10.1007/978-3-642-04244-7_29`.

26 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`.

27 Graeme Gange, Geoffrey Chu, and Peter Stuckey. Certifying optimality in constraint programming. unpublished, 2017. URL: `https://github.com/gkgange/cert-cp`.

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

29 Xavier Gillard, Pierre Schaus, and Yves Deville. Solvercheck: Declarative testing of constraints. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 of *Lecture Notes in Computer Science*, pages 565–582, Cham, 2019. Springer. `doi:10.1007/978-3-030-30048-7_33`.

30 Stephan Gocht, Ciaran McCreesh, and Jakob Nordstrom. Veripb: The easy way to make your combinatorial search algorithm trustworthy. *From Constraint Programming to Trustworthy AI*, 2020.

31 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *DROPS-IDN/v2/document/10.4230/LIPIcs.CP.2022.25*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.CP.2022.25`.

32 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, May 2021. doi:10.1609/aaai.v35i5.16494.

33 Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10886–10891. IEEE Computer Society, March 2003. doi:10.1109/DATE.2003.10008.

34 Marijn Heule. Schur number five. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(11), April 2018. doi:10.1609/aaai.v32i1.12209.

35 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, October 2013. doi:10.1109/FMCAD.2013.6679408.

36 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, Cham, 2016. Springer. doi:10.1007/978-3-319-40970-2_15.

37 John N Hooker and Greger Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96(1):33–60, 2003. doi:10.1007/s10107-003-0375-9.

38 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.

39 Nicholas Nethercote, Ralph Stuckey, Peter J. Becket, Sebastian Brand, Gregory J. Duck, and Guido Tac. Minizinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

40 Olga Ohrimenko, Peter J Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14:357–391, 2009. doi:10.1007/s10601-008-9064-x.

41 Tobias Paxian and Armin Biere. Uncovering and classifying bugs in maxsat solvers through fuzzing and delta debugging. In Matti Järvisalo and Daniel Le Berre, editors, *Proceedings of the 14th International Workshop on Pragmatics of SAT co-located with the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), Alghero, Italy, July 4, 2023*, volume 3545 of *CEUR Workshop Proceedings*, pages 59–71. CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3545/paper5.pdf.

42 Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode - generic constraint development environment. URL: https://www.gecode.org/.

43 João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In Rob A. Rutenbar and Ralph H. J. M. Otten, editors, *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, pages 220–227. IEEE Computer Society / ACM, November 1996. doi:10.1109/ICCAD.1996.569607.

44 Guido Tack and Peter J. Stuckey. Minizinc challenge 2023. URL: https://www.minizinc.org/challenge2023/results2023.html.

45 Michael Veksler and Ofer Strichman. A proof-producing CSP solver. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, volume 24, pages 204–209. AAAI Press, July 2010. doi:10.1609/aaai.v24i1.7543.

46 Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429, Cham, 2014. Springer. doi:10.1007/978-3-319-09284-3_31.

47 Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10880–10885. IEEE Computer Society, March 2003. `doi:10.1109/DATE.2003.10014`.

Using Constraint Programming for Disjunctive Scheduling in Temporal AI Planning

Adam Francis Green
Department of Informatics, King's College London, UK
Tango Hospitality Inc., Toronto, Canada

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

Amanda Coles
Department of Informatics, King's College London, UK

Abstract

We present a novel scheduling model that leverages Constraint Programming (CP) to enhance problem solving performance in Temporal Planning. Building on the established strategy of decomposing causal and temporal reasoning, our approach abstracts two common fact structures present in many Temporal Planning problems – Semaphores and Envelopes – and performs temporal reasoning in a CP-based scheduler. At each search node in a heuristic search for a temporal plan, we construct and solve a Constraint Satisfaction Problem (CSP) and integrate feedback from the CP-based scheduler to guide the causal planning search towards a solution. Through experimental analysis, we validate the impact of these advances, demonstrating a significant reduction in both the number of states searched and in search time alongside an increase in problem-solving coverage.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling; Theory of computation → Constraint and logic programming; Computing methodologies → Search methodologies

Keywords and phrases AI Planning, Temporal-Numeric Planning, Constraint Programming, Scheduling

Digital Object Identifier 10.4230/LIPIcs.CP.2024.12

1 Introduction

Temporal AI Planning is an extension of classical AI planning that includes a representation of the duration of actions and reasoning about numeric variables that change over time [9]. For such a problem, a solution is a sequence of scheduled actions, called a *plan*, which transforms the world to a desired goal state from an initial description of the world. Unlike scheduling problems such as Job-Shop Scheduling [15], but similar to most planning problems, solutions to temporal AI planning problems do not have a pre-defined set of actions to execute, but rather actions must be both selected and scheduled by a planner.

Planners designed to handle temporal problems typically take one of two approaches. *Decision-epoch* planners build a plan by adapting a classical planning approach where actions are selected and applied to a state to generate a new state, when a new action is started its effects are realised and the time of its end is added to a queue, stored in the state. The planner can then decide to apply another action at the current time or advances time until after the next action in the queue, in order to realise that action's effects. Further actions can then be applied at this time point, with the process repeated until all goals are achieved and all executing actions have completed [7, 8]. In the decision-epoch approach, the planner reasons about both the causal and the temporal aspects of the problem.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 12; pp. 12:1–12:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Decomposition planners separate causal and temporal reasoning [11]. The planner applies actions to states to generate an atemporal successor state, then checks the temporal consistency of the partial plan for that state using a scheduler. If the resulting state is numerically and temporally consistent, additional actions can be applied to extend the plan. If the resulting state is not consistent, it is discarded and the planner explores the expansion of other states. This alternation between planning and scheduling continues until a goal state is achieved. Thus, the causal reasoning takes place in task planning, while the temporal reasoning is done using a sub-solver model such as a Simple Temporal Network [11] or a Linear Program [3].

Decomposition Temporal Planning lends itself to transferring other temporal and/or combinatorial structures to the sub-solver, provided that it has the ability to model and solve such structures. This paper presents two contributions to decomposition approaches in temporal AI planning.

1. We show that, through the use of a constraint-based scheduler, additional temporal and combinatorial reasoning can be transferred to the sub-solver by abstracting semaphore and envelope fact structures, reducing the number of states that the planner needs to explore.
2. We show how this transfer also enables an increase in the feedback from the scheduler to the planner enabling the planning heuristic to identify dead-ends earlier and, thus, substantially reducing the search effort.

Our experiments show that our approach leads to a significant reduction in states searched, increased coverage, and improved solve times in a range of International Planning Competition (IPC) benchmark domains. Moreover, because we use a preprocessing detection to identify these semaphore and envelope fact structures, there is no negative impact of our approach on domains where these structures do not exist. This paper demonstrates that CP can be a powerful tool for a kind of combinatorial and temporal reasoning that is traditionally solved inefficiently in the search, which is designed for causal reasoning in AI planners.

These contributions are different from previous applications of Constraint Programming in Temporal Planning – such as CPT [18, 10] or EUROPA [1] – because previous approaches have directly solved the temporal planning problem. Moreover, our approach can model numeric resources and supports a temporal-numeric fragment of PDDL2.1 even though the performance improvements we demonstrate are derived from the temporal aspect of the problem.

2 Background

2.1 Temporal Planning

We consider the fragment of PDDL 2.1 [9] planning problems supported by COLIN and POPF [3] and limit our attention to problems without continuous numeric effects, although our work could be extended to support them.

We define a planning problem as a tuple $\langle F, V, A, I, G\rangle$. F is a finite set of facts. A fact $f \in F$ is a proposition which, if present in a given state s, indicates that the fact is true. V is a finite set of numeric variables. I denotes the initial state. A state is a tuple $\langle F_s, n_s\rangle$, where $F_s \subseteq F$ and n_s is a set of assignments to the variables in V. $n_s[v]$ denotes the value of variable $v \in V$ in state s. A is a set of durative actions, representing operators that can be applied to states. G is a set of conditions that must hold true following execution of any valid solution plan.

We begin by defining a condition ψ. Conditions can either be propositional ($\psi \in F$) or numeric. Numeric conditions are defined as a tuple $\psi = \langle v \ op \ c \rangle$ such that $op \in \{\geq, \leq, <, >, =\}$ and $c \in \mathbb{R}$. For a given state $\langle F_s, n_s \rangle$, ψ is satisfied if it is a proposition where $\psi \in F_s$ or it is a numeric condition where $\langle n_s[v] \ op \ c \rangle$ is true.

A durative action $a \in A$ is defined as a tuple $\langle pre(a)_{\vdash}, \mathit{eff}(a)_{\vdash}, pre(a)_{\leftrightarrow}, pre(a)_{\dashv}, \mathit{eff}(a)_{\dashv}, d_a \rangle$. $pre(a)_{\vdash}$ ($pre(a)_{\dashv}$) is a set of propositional and numeric conditions (preconditions) on the start (end) of the action a. $pre(a)_{\leftrightarrow}$ are invariant conditions over the duration of action a that must be satisfied for all the time the action is executing. Individually we denote these preconditions as $pre(a)^{p}_{\{\vdash,\dashv,\leftrightarrow\}}$ for propositional conditions and $pre(a)^{n}_{\{\vdash,\dashv,\leftrightarrow\}}$ for numeric conditions. d_a is a pair $\langle d_{min}, d_{max} \rangle$ representing the minimum and maximum duration of the action a as positive real values.

$\mathit{eff}(a)_{\vdash}$ ($\mathit{eff}(a)_{\dashv}$) are effects that occur at the start (end) of the action a and consist of a tuple $\langle \mathit{eff}^{+}, \mathit{eff}^{-}, \mathit{eff}^{n} \rangle$. eff^{+} (eff^{-}) is a set of propositions in F that are added to (deleted from) a state s to create the subsequent state s' (e.g. $F_{s'} = (F_s \setminus \mathit{eff}^{-}) \cup \mathit{eff}^{+}$).

eff^{n} is a set of numeric effects of the form $\langle v \ op \ c \rangle$, where $op \in \{+=, -=, =\}$, $c \in \mathbb{R}$ and $v \in V$. Numeric effects using the operators $+=$ and $-=$ use the value of v in s, to calculate the value of v in the subsequent state s'. This is a commonly used restricted version of PDDL 2.1 numeric planning.

The goal G is a set of propositional and numeric conditions. A state $s \models G$ if s satisfies the propositional and numeric conditions in G.

A plan π is a list of tuples of the form $\langle a, t, d \rangle$, with t representing the timestamp at which action a is applied and d representing the duration of a where $d_{min} \leq d \leq d_{max}$. A plan π is valid iff for all tuples $\langle a, t, d \rangle$ in π, $pre(a)_{\vdash}$ is satisfied at time t, $pre(a)_{\dashv}$ is satisfied at time $t + d$, $pre(a)_{\leftrightarrow}$ is satisfied at all points in the interval $(t, t + d)$. π is a solution if it is valid and the resulting state S_G, when all actions have finished executing, satisfies G.

2.2 Constraint Satisfaction Problems

Formally, a Constraint Satisfaction Problem (CSP) is a tuple $\langle X, D, C \rangle$, where X is a set of n variables, D is a set of n domains corresponding to the variables in X and $\forall d \in D, d \subset \mathbb{Z}$.

C is a set of *constraints*. A constraint is an m-ary ($m \leq n$) function $c(v_0, ..., v_m) \rightarrow \{\mathit{true}, \mathit{false}\}$ where $v_i \in d_i$ and $d_i \in D$ represents the domain of variable x_i. A constraint can be any mapping of an assignment for the variables in X to a truth value which indicates if the constraint is satisfied or not.

A solution for a CSP $\langle X, D, C \rangle$ is a n-tuple $V = \langle v_0, ..., v_n \rangle$ representing an assignment of the value v_i to the variable $x_i \in X$ where $v_i \in d_i$ and $\forall c \in C, c(V) = \mathit{true}$.

In CP-based scheduling, *interval variables* represent an optional time window. The domain of an interval variable is a set of the form $\{\perp\} \cup \{[s, e) | s, e \in \mathbb{Z}, s \leq e\}$ where s (e) represents the start (end) time of the interval. For an interval variable x, if $x = \perp$ it is not present in the solution to the problem.

A global constraint is a relation on an arbitrary number of variables, typically representing frequently observed combinatorial structure such as a set of variables all requiring pairwise different values. Explicit representation of such relations improves problem solving performance through the use of inference (or propagation) algorithms designed for the corresponding structure [16].

We use three common global constraints: *noOverlap* (or *disjunctive*), *alternative*, and *span*.

▶ **Definition 1.** *$noOverlap(S, \upsilon)$ is a global constraint over a set of interval variables S, and numeric constant υ where for any two present intervals in S, x_i and x_j, $start(x_i) \geq end(x_j)+\upsilon$ or $start(x_j) \geq end(x_i) + \upsilon$ holds.*

▶ **Definition 2.** *$alternative(x_a, S)$ is a global constraint over an interval variable x_a, and a set of interval variables S where $x_a \notin S$. x_a is present in the solution, iff exactly one interval $x \in S$ is also present. When x_a is present, $start(x_a) = start(x) \land end(x_a) = end(x)$.*

▶ **Definition 3.** *$span(x_c, S)$ is a global constraint over an interval variable x_c and a set of interval variables S, $x_c \notin S$, that ensures that all present intervals in S are scheduled between the start and end of x_c and that interval x_c starts with the start of the first present interval in S and ends with the end of the last present interval in S. x_c is absent iff all intervals in S are absent.*

2.3 Planning through Decomposition

Decomposition-based planners separate temporal reasoning from causal reasoning by relaxing the problem to a causal representation and finding a solution to this representation using search. In OPTIC, an A^* search with the Metric Temporal Relaxed Planning Graph (TRPG) [13, 6] heuristic is used to explore the state space. To reason about temporal planning, OPTIC splits durative actions into snap actions.

▶ **Definition 4.** *An action a of the form $\langle pre(a)_\vdash, eff(a)_\vdash, pre(a)_\leftrightarrow, pre(a)_\dashv, eff(a)_\dashv, d_a\rangle$, can be abstracted as a pair of* ***snap actions*** *$\langle a_\vdash, a_\dashv\rangle$: a tuple comprising $\langle pre(a)_\vdash, eff(a)_\vdash\rangle$ ($\langle pre(a)_\dashv, eff(a)_\dashv\rangle$). Snap actions represent the instantaneous start and end preconditions and effects of a durative action; the duration and invariant conditions must be tracked separately.*

The search is modified to ensure that each start snap action has a corresponding end action and that invariant conditions between start and end snap actions are satisfied [3].

The successor of a state s reached by applying the sequence of snap actions (partial plan) π_s are each generated by applying a new snap action a, resulting in a new state with partial plan π' (a appended to π_s). From π', OPTIC creates a scheduling problem τ as follows [4]:

- A set of temporal constraints of the form $min \leq t(a') - t(a) \leq max$ where $a, a' \in \pi'$ and min and max are constants.
  - If the temporal constraint represents an ordering between two snap actions $min = \epsilon$ and $max = \infty$. Ordering constraints ensure that (1) if an action has a precondition on a fact p then it is ordered after the last adder of p (i.e., the last action that adds p); (2) if an action adds p it is ordered after the last deletor of p and (3) if an action deletes p it is ordered after the last adder and all conditioners on p since it was last added (4) if an action contditions on or effects a variable v it is ordered after the last action to condition on or effect v.
  - If the temporal constraint represents the duration between the start and corresponding end snap action for duration action a, $min = d_{min}$ and $max = d_{max}$ where $d_a = \langle d_{min}, d_{max}\rangle$. Duration constraints ensure that the time between the start and end snap actions in any valid solution for τ is consistent with duration bounds of the durative action they represent.
- In the presence of continuous or duration-dependent effects these are encoded in the scheduling problem over the values of numeric variables (v in V) before or after each snap action.

In every state generated, a scheduling problem τ is constructed from the partial plan π'. τ is solved using a sub-solver such as an STN-based solver, Linear Program (LP) or Mixed Integer Program (MIP) which attempts to find a set of action timestamps that satisfy the constraints.

If there is no feasible timestamp assignment, the state is temporally inconsistent and is pruned. If an assignment is found and the state satisfies the goal, then the plan π constructed from the timestamps is a solution. If the state is consistent but not a goal, then control returns to the planner to continue its search.

3 Building a CP model from π'

We now construct a drop-in CP replacement for the STN/MIP scheduler currently used in OPTIC, based on the temporal and numeric constraints described in Section 2.3. Consider a partial plan π' and a scheduling problem τ; we construct a CSP $\langle X, D, C\rangle$ as follows.

Variables

For each start snap action $a_\vdash$ in π' we add an interval variable $x_a = [s, e)$ to X regardless of whether $a_\dashv$ exists in π'. s represents the timestamp of snap action $a_\vdash$, and e of $a_\dashv$, if $a_\dashv$ is present. If $a_\dashv$ is not part of the plan π', then we still represent the unclosed duration action that corresponds to $a_\vdash$ with an interval and constrain its duration according to the duration of the durative action it represents, however there will be no ordering constraints on $a_\dashv$ (and consequently on e) because the planner has yet to reason about the addition of $a_\dashv$ to the plan. This allows the scheduler to reason with x_a as if it were a closed action, preserving the duration. In any circumstance, even if the CP solver concludes that a schedule exists for a plan π' with an unclosed action, the plan is not valid and a final temporal consistency check will occur once a complete plan is produced.

If $V \neq \emptyset$ we add two integer variables $x_{a,v}$ and $x'_{a,v}$ for each $v \in V$ and each snap action $a \in \pi'$. These variables represent the value of variable v before and after the application of the snap action.

Temporal Constraints

For each temporal constraint in τ of the form $\mathit{min} \leq t(a') - t(a) \leq \mathit{max}$, a constraint is introduced according to whether the snap action a (a') correspond to a start $a_\vdash$ ($a'_\vdash$) or end $a_\dashv$ ($a'_\dashv$) snap action. Constraints are mapped accordingly as follows. If a and a' are start snap actions then $\mathit{min} \leq \mathit{start}(x'_a) - \mathit{start}(x_a) \leq \mathit{max}$ is introduced. If a and a' are end snap actions then $\mathit{min} \leq \mathit{end}(x'_a) - \mathit{end}(x_a) \leq \mathit{max}$ is introduced. In the case that a is a start snap action and a' is an end snap action (or vice versa) a constraint $\mathit{min} \leq \mathit{end}(x'_a) - \mathit{start}(x_a) \leq \mathit{max}$ ($\mathit{min} \leq \mathit{end}(x'_a) - \mathit{end}(x_a) \leq \mathit{max}$) is introduced.

Numeric Constraints

There are two types of numeric constraints which we model: conditions and effects. A numeric condition as defined in the planning problem with the form $\langle v\{\geq, \leq, <, >, =\}c\rangle$ occurring in $\mathit{pre}(a)$, is imposed on $x_{a,v}$ as it represents the value of the variable v before the application of snap action a e.g. $x_{a,v}\{\geq, \leq, <, >, =\}c$.

If a numeric effect exists in $\mathit{eff}^n(a)$ of the form $\langle v\{+\!=, -\!=, =\}c\rangle$ then the resulting constraint is $x'_{a,v} = x_{a,v}\{+\!=, -\!=, =\}c$. If no numeric effect exists then $x'_{a,v} = x_{a,v}$.

These constraints model discrete numeric effects within the scheduling problem. In problems where there are no continuous or duration-dependent effects, these need not be reasoned about in the scheduler (since, given the ordering constraints OPTIC generates, the timestamps assigned to actions cannot affect the value of numeric variables), so an STN, without any numeric constraints can be used to solve the scheduling problem. However, in the presence of either of these, the numeric preconditions and effects must be modelled to ensure temporal-numeric constraints are satisfied. The approach we take to modelling these in our CSP model, mirrors that used in OPTIC. We limit ourselves to discrete effects, here we will use these later in our feedback mechanism from the scheduler to the planner; but in general the same constraints over these variables that are used in OPTIC's MIP can be used to represent continuous/duration dependent effects in a CSP.

In OPTIC, when constructing a scheduling problem τ, a total ordering is imposed between snap actions which condition on or effect the same variable v (regardless of whether this is necessary). We could in principle relax these constraints, but since our focus here is not specifically on numeric planning we maintain them as is. The initial value of $x_{a',v}$, therefore, is constrained to be equal to the final value of v in a ($x_{a',v} = x'_{a,v}$) as we know this would have been the last time the value of the variable v changed. If a is the first snap action π' to condition or effect on v, then $x_{a,v}$ takes the value of v in the initial state ($x_{a,v} = n_I[v]$).

Domain

The domain d_a is defined for an interval variable x_a as the interval $[0, h]$ where h represents the sum of the maximum duration of all actions in the plan π', plus ϵ multiplied by the number of actions in the plan. The horizon h is an upper bound as in the worst case each action will be executed without overlap with any other.

The domain d_v – which applies to all integer variables representing v – is an interval $[min_v, max_v]$ defined by the sum of all positive numeric effects ($+=$) effects applied to the initial value max_v, and the sum of all negative effects ($-=$) min_v. In the event assignment ($= c$ where c is some constant) occurs, the interval is $[c - min_v, c + max_v]$.

Temporal planning and CP scheduling problems have adopted different conventions for representing the timing of events. As shown in the definition of an interval variable, in CP, time intervals are considered to be open on the right, thus allowing one interval to end at time t and another to begin at the same time even if they are constrained to not overlap. In contrast, temporal planning uses ϵ as the smallest representable unit of time. Actions that are constrained to not overlap must be separated by at least ϵ. To handle this mismatch, we represent ϵ in the CP model as one unit of time and thus force an extra gap between actions, consistent with the planning definition. Our CP scheduler does not affect OPTIC's support for self-overlapping actions, is equivalent to the current STN-based approach in OPTIC, and as such does not impact the soundness or completeness to OPTIC.

4 Abstracting Semaphores and Envelopes

So far our temporal reasoning problem is identical to that solved by the STN solver in OPTIC. In this section, we extend our temporal representation to take advantage of CP's greater expressivity and solving power.

Semaphore and *envelope* facts [11] are causal modelling patterns that, respectively, prevent and require the concurrent execution of a set of actions. Because these are facts, and therefore a causal consideration, they have been considered during causal reasoning, despite being a temporal structure. We formally define *semaphore* and *envelope* facts as follows.

▶ **Definition 5.** *A **semaphore fact** f is a fact such that $f \in I$ and $\forall a \in A$ exactly one of the following holds*

- *a is a **mutual exclusive action**, that is, f is in and only in $pre(a)^{p}_{\vdash}$, $eff(a)^{-}_{\vdash}$ and $eff(a)^{+}_{\dashv}$, or*
- *a is an unrelated action, that is f is not in any precondition or effect of a.*

Definition 5 ensures that all actions that condition on a *semaphore fact* delete it at the start and add it at the end, ensuring actions that condition on a semaphore are mutually exclusive (cannot execute in parallel). Requiring all other actions to be unrelated ensures the semaphore fact serves only to enforce mutual exclusion and has no other function.

▶ **Definition 6.** *An **envelope fact** f is fact such that $f \notin I$ and $\forall a \in A$ exactly one of the following holds*

- *a is an **envelope achiever**, that is, f is only in $eff(a)^{+}_{\vdash}$ and $eff(a)^{-}_{\dashv}$,*
- *a is an **envelope conditioner**, that is, f is in $pre(a)^{p}_{\leftrightarrow}$ and optionally in $pre(a)^{p}_{\vdash}$ and/or $pre(a)^{p}_{\dashv}$ and f is not in $eff(a)_{\vdash}$ and $eff(a)_{\dashv}$, or*
- *a is an unrelated action, that is, f is not in any of the sets: $pre(a)^{p}_{\vdash}$, $eff(a)_{\vdash}$, $pre(a)^{p}_{\leftrightarrow}$, $pre(a)^{p}_{\dashv}$, or $eff(a)_{\dashv}$.*

An *envelope fact* ensures that every envelope conditioner executes concurrently with some envelope achiever. Definition 6 ensures that an *envelope achiever* adds the envelope fact at its start and deletes the same fact at its end, thus creating a window where this fact is available. The definition also states that an *envelope conditioner* has an invariant condition and thus must execute concurrently with some envelope achiever. An unrelated action as defined in Definition 6 means an action can only act as an achiever or conditioner on an envelope fact and not use an envelope fact for any other purpose.

Definition 6's requirement for all other actions to be unrelated ensures that (i) no other actions adds the envelope fact, so all conditioners *must* occur within an achiever, (ii) no other actions delete the envelope fact, thus it remains throughout the entire execution of an envelope achiever, and (iii) no other actions condition on the envelope facts at only the start or end and thus only need to be executed partially concurrently with an achiever.

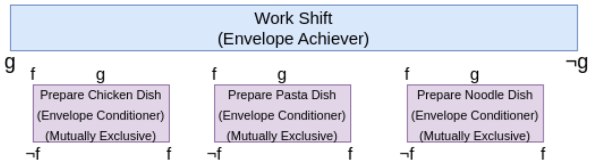


Figure 1 Envelope fact g enforces concurrency with a *work shift* whilst semaphore fact f prevents more than one *work activity* from happening at a time. A fact appearing above (below) an action indicates it is a condition (effect) respectively. Position indicates whether the fact is a start, invariant, or end condition or effect. $\neg f$ denotes that f is being deleted (made false).

Figure 1 shows how a semaphore f and envelope g interact. The blue "work shift" action is an envelope achiever of g, into which a number of work activities have to be scheduled. These activities – preparing Chicken, Pasta and Noodle dishes – are mutually exclusive due to semaphore fact f and are also envelope conditioners on g.

4.1 Abstracting Semaphore Facts

Current state-of-the-art decomposition planners search over all total ordering constraints between durative actions a_i and a_j that condition on semaphore fact f.

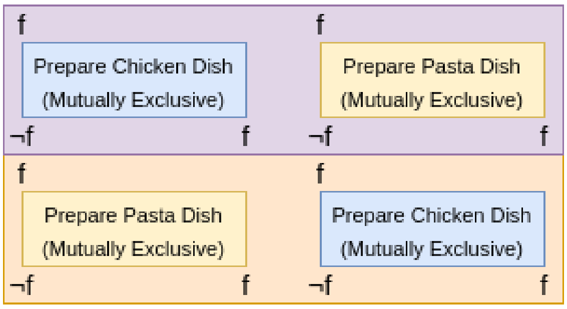


Figure 2 For two mutually exclusive actions, two orderings exist that achieve the same state.

Figure 2 shows how the mutual exclusion of semaphore f creates two alternate plans to explore to achieve the same state. If there are n actions required to achieve goal G that condition on the semaphore fact f, there are $n!$ orderings of those actions to be considered. However in many cases, side effect constraints and other optimisations [5] can reduce the search space.

To abstract a semaphore fact f, from a planning problem $\langle F, V, A, I, G\rangle$, we create a set of actions $m_f = \{a : a \in A, f \in pre(a)^p_\vdash\}$. We perform the following set operations $\forall a \in m_f$: $pre(a)^p_\vdash \setminus \{f\}$, $\mathit{eff}(a)^-_\vdash \setminus \{f\}$, and $\mathit{eff}(a)^+_\dashv \setminus \{f\}$. Finally, we remove f from the initial state I.

For the scheduling problem corresponding to partial plan π', a new *mutual exclusion constraint* is added for the actions in π' that appear in the set of actions m_f.

▶ **Definition 7.** *A mutual exclusion constraint is a noOverlap constraint of the form* $\boldsymbol{noOverlap}(\{x_a \in X : a \in \pi' \wedge a \in m_f\}, \epsilon)$.

Following Definition 5, we demonstrated that the only purpose of a semaphore fact was to ensure mutual exclusion between actions conditioning on it. A semaphore fact has no implied ordering between actions that condition on it. Search in OPTIC imposes a total ordering among such actions (due to ordering each conditioner after the most recent adder), with different orderings considered by exploring different plans. We replace the total orderings imposed at the planning level, with a mutual exclusion constraint imposed during scheduling. The scheduler will consider any ordering of actions in the mutual exclusion constraint that respects ordering constraints imposed by other facts (since these still remain as temporal constraints). As a result, the substitution preserves soundness. By considering all sound partial orderings, the substitution maintains completeness.

4.2 Abstracting Envelope Facts

An envelope achiever of fact g creates a time window that an envelope conditioner on g must execute within. With multiple achiever and conditioner actions, a set of possible time windows is defined and the planner must decide which envelopes each conditioner must execute within.

In Figure 3 two achievers of g – “Work Shift 1” and “Work Shift 2” – exist and “Prepare Chicken Dish” could be scheduled within either; resulting in two different assignments to consider. The number of assignments grows exponentially with both the number of envelope achievers n and the number of envelope conditioners m provided $n > 1$.

Assignment decisions are only important if envelope conditioners cannot be executed concurrently. If all envelope conditioners can execute concurrently, an envelope achiever large enough to execute concurrently with the longest conditioner can satisfy all conditioners.

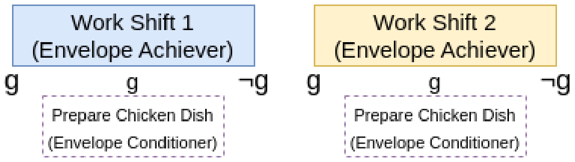


Figure 3 For two envelope achievers (work shift 1 and 2); prepare chicken could be assigned to either.

To abstract an envelope fact g from a planning problem $\langle F, V, A, I, G\rangle$, we create two sets of actions $\mathit{achievers}_g = \{a : a \in A, g \in \mathit{eff}(a)^{+}_{\vdash}\}$ and $\mathit{conditioners}_g = \{a : a \in A, g \in \mathit{pre}(a)^{p}_{\leftrightarrow}\}$. $\forall a \in \mathit{achievers}_g$, we perform $\mathit{eff}(a)^{-}_{\dashv} \setminus \{g\}$. $\forall a \in \mathit{conditioners}_g$ we perform the following set operations: $\mathit{pre}(a)^{p}_{\leftrightarrow} \setminus \{g\}$, $\mathit{pre}(a)^{p}_{\dashv} \setminus \{g\}$, and $\mathit{pre}(a)^{p}_{\vdash} \cup \{g\}$.

To ensure that envelope conditioners are scheduled concurrently with an envelope achiever, a new *envelope constraint* comprised of new variables and a set of constraints is added to the scheduling problem for the partial plan π'.

▶ **Definition 8.** *An envelope constraint for an envelope fact g, with sets of actions $\mathit{achievers}_g$ and $\mathit{conditioners}_g$ and partial plan π', comprises new interval variables $x_{a,c}, \forall a \in \mathit{achievers}_g, \forall c \in \mathit{conditioners}_g$, and a dummy optional interval $x_{a,dur}$ for all $a \in \mathit{achievers}_g$ and the following constraints:*

- *$\forall c \in \pi$ that also appears in $\mathit{conditioners}_g$, $\mathit{alternative}(x_c, \{x_{a,c} : a \in \mathit{achievers}_g \cap \pi'\})$. x_c is the interval variable in X representing the conditioner action c. Exactly one achiever, a, for each conditioner is assigned by enforcing the presence of one optional interval variable $x_{a,c}$.*
- *$\forall a \in \pi'$ that also appears in $\mathit{achievers}_g$, $\mathit{span}(x_a, \{x_{a,c} : c \in \mathit{conditioners}_g\} \cup \{x_{a,dur}\})$. x_a is the interval variable in X representing the achiever action a. This ensures that each envelope conditioner executes concurrently with the envelope achiever assigned in the alternative constraint.*

The dummy optional interval $x_{a,dur}$ ensures that the conditioners on an envelope do not have to be scheduled such that one starts exactly at the start of x_a and one finishes exactly at the end. The dummy action can be used to satisfy this condition imposed by the span constraint, and thus we do not compromise completeness (as the planning model does not necessarily imply this constraint).

Following Definition 6, we demonstrated that the only purpose of an envelope fact in the domain was to ensure that all envelope conditioners execute entirely within envelope achievers. An envelope fact does not imply an assignment of a specific conditioner to a specific achiever.

In OPTIC, an assignment of a conditioner to an achiever is done during search. Search first adds the start of an achiever ($a_{\vdash}$), then the start of the conditioner ($c_{\vdash}$), imposing an ordering constraint in τ: $t(a_{\vdash}) < t(c_{\vdash})$. Search then adds the end of the conditioner ($c_{\dashv}$). Finally, search adds the end of the achiever ($a_{\dashv}$), ordering the end of the achiever after the end of the conditioner ($t(c_{\dashv}) < t(a_{\dashv})$). The result of these constraints is a total ordering ($t(a_{\vdash}) < t(c_{\vdash}) < t(c_{\dashv}) < t(a_{\dashv})$).[1]

[1] OPTIC may choose to add some other action or actions between the addition of these individual snap actions. This is a simple example of how envelope concurrency is achieved.

In imposing these ordering constraints in the scheduling problem τ, search assigns a conditioner to an achiever and enforces their concurrent execution using the same constraints. Search in OPTIC considers a different assignment of an achiever to a conditioner, by performing the same process described above, but using different achievers.

By abstracting an envelope fact, leaving only a start add effect for achievers and a start precondition for conditioners, the only ordering constraint added by OPTIC to τ orders all conditioners after the first achiever; an ordering implied in any valid envelope assignment and execution. By using an *alternative* constraint, the scheduler can consider the assignment of a conditioner to any achiever. Meanwhile, the *span* constraint ensures the concurrent execution of a conditioner with the achiever it is assigned to.

Because we only abstract envelope facts, other constraints in τ are preserved (still generated by OPTIC's machinery). The assignment of a conditioner to an achiever will only be considered by the CP Scheduler if that assignment respects all other constraints in τ. As a result, this abstraction does not compromise soundness.

By allowing the CP Scheduler to consider different assignments of conditioners to achievers, we ensure that, in one state, the scheduler considers all orderings that OPTIC considers over a number of states. We therefore do not compromise completeness.

5 Improving Feedback to the Planner

The abstractions described in Section 4 and allow us to transfer some reasoning about orderings of actions in the planner's search to reasoning with powerful global constraints in a CP sub-solver. These abstractions also allow us to enhance the communication between the causal reasoning in task planning and the temporal reasoning in scheduling.

In delete-free relaxation heuristics [12], envelopes are a particular challenge [11] because once an envelope fact g is achieved, there is no further value to the heuristic to achieve g again. Without reasoning about how much time is required for envelope conditioners, search has to blindly add achievers until sufficient time is available for the scheduler. Yet if envelopes have limited time, it is critical that further achievers are added. The need for multiple achievers is greatest when conditioners are mutually exclusive

In Section 4.2, when an envelope fact g was abstracted, the add effects in all achievers were preserved and a start precondition was added to all conditioners on g. This transformation forces search to add at least one achiever prior to adding any conditioners for envelope fact g.

To guide search, we now add a new subset of numeric variables to the set V for a planning problem $\langle F, V, A, I, G\rangle$. These new numeric variables – *Envelope Time Tracking variables* – create a producer-consumer relationship between envelope achievers and mutually exclusive conditioners. Envelope Time Tracking variables are added for each semaphore fact f and envelope fact g.

▶ **Definition 9.** *A time tracking variable is a numeric variable $v_{f,g} \in V$. The initial state value of variable $v_{f,g}$ is* 0 *($n_I[v_{f,g}] = 0$).*

Each achiever $a \in achievers_g$ has a new start effect $v_{f,g}$ += d_{max}. Meanwhile each conditioner $c \in conditioners_g \cap m_f$ has a new start condition $v_{f,g} \geq d_{min}$ and a new start effect $v_{f,g}$ −= d_{min}.

The variable $v_{f,g}$ increases by the maximum duration of each achiever of g added to π'. The variable $v_{f,g}$ decreases by the minimum duration of each conditioner of g in m_f added to the partial plan π'. $v_{f,g}$ represents a trivial upper bound on the amount of free time remaining in envelope achievers for envelope conditioners in mutual exclusion m_f.

The introduction of the precondition $v_{f,g} \geq d_{min}$ means that, across all achievers, there must exist enough free time to fully contain a conditioner before the search can add it. Pruning based on the value of $v_{f,g}$ does not compromise completeness because it is an overestimate and so we necessarily need to add another achiever before we can fit any further conditioners.

Prioritising More Easily Schedulable States

If π' is shown to be temporally consistent, the scheduler produces a plan π which includes a valid schedule for all actions in the plan. We can use this schedule to identify when adding a new action to the plan will likely lead to a trivially schedulable partial plan; and when we might need to add another envelope achiever in order to find a solution.

For a given envelope achiever a in π we can compute the maximum free time, $v^a_{f,g}$, by summing the available free time in a in the computed schedule.

▶ **Definition 10.** *For an envelope achiever a, $v^a_{f,g} = d_a - \sum d_c$, $\forall c \in conditioners_g \cap m_f$ where conditioner c is in the plan π and scheduled concurrently with a (i.e. in π, t_c is in the interval $[t_a, t_a + d_a]$).*

Variable $v^a_{f,g}$ allows us to determine the maximum free space within a single envelope achiever a in a plan π. Figure 4 shows an example of the calculation in Definition 10, where the maximum possible free space within the envelope a is 10 time units.

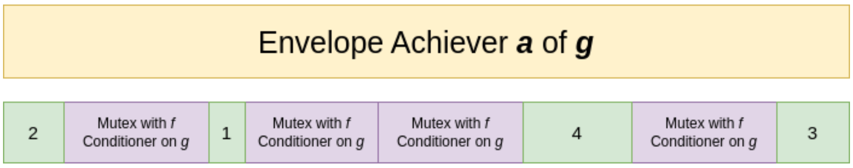

Envelope Achiever a of g
2
Mutex with f Conditioner on g
1
Mutex with f Conditioner on g
Mutex with f Conditioner on g
4
Mutex with f Conditioner on g
3

Figure 4 illustrates an envelope achiever a of envelope fact g to which several conditioners have been assigned in a plan.

$v^\pi_{f,g}$ is calculated for a plan π using Definition 10, a semaphore fact f and envelope fact g as follows:

▶ **Definition 11.** *For a plan π, a semaphore fact f, and an envelope fact g is $v^\pi_{f,g} = max(v^a_{f,g})$ (Definition 10), $\forall a \in achievers_g$, where a is in the plan π.*

$v^\pi_{f,g}$ gives us an estimate across the plan π of the longest conditioner we could schedule within any achiever. Whilst this bound is a good estimate of the upper bound, it is not a guaranteed maximum because it is possible that reassigning conditioners to different achievers could increase the value of $v^a_{f,g}$ for some achiever a. Thus, we cannot use $v^\pi_{f,g}$ for pruning as we do with $v_{f,g}$. Instead we favour expanding states that are more likely to be schedulable: any state generated by applying an action for which $v^\pi_{f,g} \leq d_{min}$ is added to a second open list that is only expanded if the first open list becomes empty.

This manipulation guides search to favour adding conditioners whose duration fits within an existing achiever or adding another achiever first; before attempting to add further conditioners. Such a preference is useful in counteracting the heuristic blind spot discussed earlier, in which there is no heuristic guidance to open new envelopes. Prioritisation of states in this way does not compromise completeness because, whilst it would not be sound to prune states based on $v^\pi_{f,g}$, the use of a second open list ensures that these states will be explored eventually if required.

Table 1 Benchmark domains used.

IPC Domain	No. of Problems	Type
cafe	29	both
crew planning	29	both
driverlog (shift)	20	envelopes only
match	20	both
pipes (no tankage)	30	envelopes only
turn and open	20	envelopes only
satellites	30	both
TMS	20	both

This type of communication between the planner and scheduler is novel. Where previous communication has been limited to constraints and inconsistencies, here the scheduler is communicating temporal information that is incorporated into the planner's search, opening the possibility of the communication of other temporal search guidance.

6 Evaluation

To evaluate our transformations, we identify a wide variety of International Planning Competition (IPC) domains from across a number of years that contain envelopes and (optionally) semaphores. Table 1 shows the domains used. The use of our transformations and the CP scheduler is automatically decided based on the presence of envelope and/or semaphore facts, in a preprocessing step that takes $< 0.001s$, if none are detected the planner runs exactly as before. Therefore performance on domains which do not contain envelopes or semaphore facts is unaffected; thus we do not consider these in our evaluation.

We compare the performance of the CP Scheduling approach to the baseline standard approach in OPTIC. The two configurations we tested are:

- The *base configuration* is a version of OPTIC [2], a leading general-purpose temporal-numeric planner. OPTIC uses an STN-based scheduler with a P-time complexity.[2]
- The *experimental configuration* is the base configuration plus a CP scheduler that includes our abstraction of semaphores and envelopes and the time tracking variables. The CP scheduler is configured to return the first feasible solution found. The search and memoization machinery of OPTIC is otherwise unchanged.

We experimented with a configuration that caches the scheduler solution to allow warm starting in subsequent states. This was not successful because the memory overheads of storing a CP solution in every state are too high, and the planner began to hit the memory limit.

All problems were executed on an Intel i7-8650U 1.9GHz machine with 3GB of memory and a search time limit of 30 minutes. We use IBM's CP Optimizer 22.1.0 as our CP sub-solver. Each plan is validated using VAL [14]. We note that all unsolved problems were the result of search timeout, rather than memory limits.

[2] OPTIC has both an STN and a MIP scheduler which it chooses based on the nature of the problem. None of these problems have continuous numerics, so OPTIC defaults to the STN scheduler.

The objective of this evaluation is to investigate whether the transformations reduce the number of states explored, by replacing multiple planning states with a single state with a less constrained partial order plan and increased search guidance through temporal feedback from the scheduler. We go on to investigate whether planner performance is improved as a result. Thus our evaluation focuses on comparing the two scheduling approaches within the same planner.

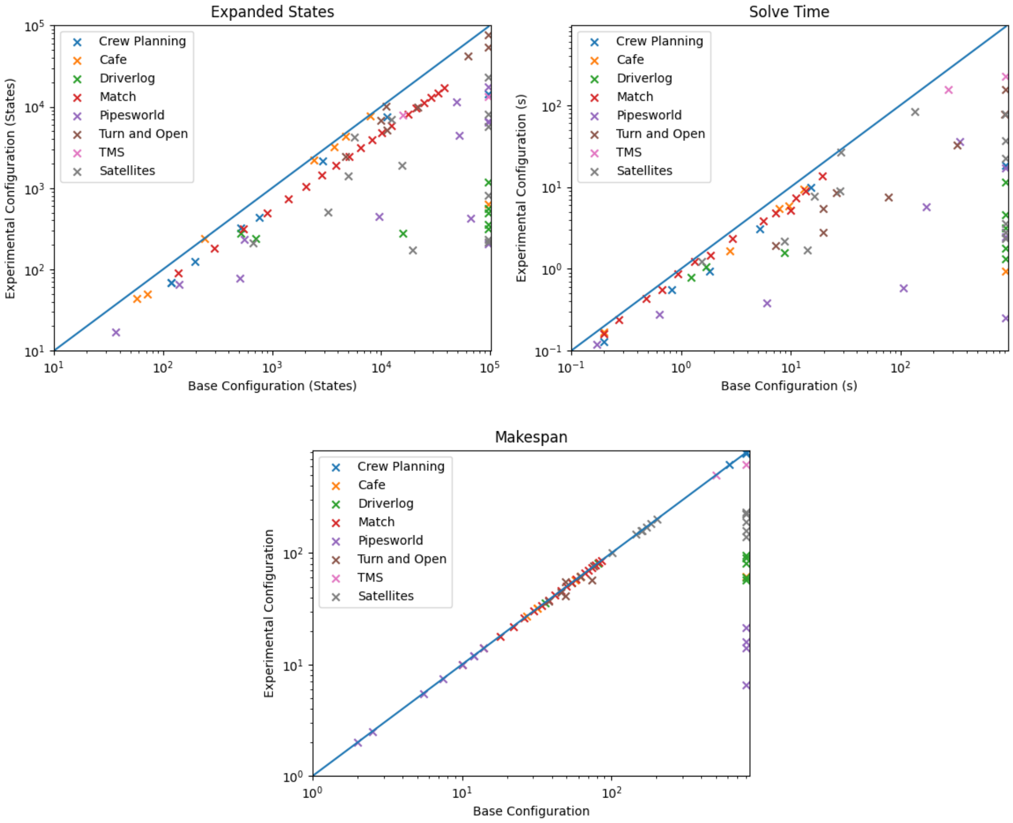


Figure 5 Comparison of base and experimental configuration, by states, time and makespan. Points on the far right/top axis indicate problems not solved by the base/experimental configuration respectively.

6.1 Reduction in States Generated

The graph of *states generated* in Figure 5, shows that the experimental configuration reduces the number of states generated across all domains. The shift of the disjunctive reasoning, created by semaphore and envelope facts, from the planner to the scheduler means fewer disjunctive decisions are made in planning search and consequently fewer states are generated.

Domains with a modest reductions in the number of states generated, such as the Match domain and Cafe domain, are a result of the underlying envelope and semaphore structure. In the Match domain, the goal is to mend a set of fuses in a power outage where matches must be lit to provide light. Each mend fuse action is mutually exclusive, and any fuse can be fixed whilst any match is lit. Each match can be used to repair at most two fuses, and the goal is for all fuses to be fixed with the matches provided. Matches are symmetrical; therefore,

Table 2 No. of Problems solved by base and experimental configurations.

IPC Domain	Problems	Base	Experimental
cafe	29	7	8
crew planning	29	7	8
driverlog	20	3	8
match	20	19	19
pipes	30	8	12
turn and open	20	8	8
satellites	30	7	13
TMS	20	1	2

reordering how matches are lit does not make a temporally inconsistent partial plan consistent, so abstracting mutual exclusion is redundant. For the envelope assignment problem that exists between matches and fuses, each match is equally capable of mending each fuse. Thus the choice of assignment is also redundant. The only benefit of the transformations described is in guiding search to add sufficient matches to mend all fuses.

Cafe has a similar structure to Match, with three distinctions. Firstly, Match consists of one "temporal" resource being consumed: matches. In Cafe, there are two symmetrical resources, Ovens and Cooks. Ovens and Cooks are not a consumable resource, i.e. they can be reused to complete other actions. The final distinction is that the meals being prepared have different durations, whereas each fuse takes the same amount of time to fix in Matches. Despite these differences, they are otherwise very similar domains and therefore experience a similar issue where searching symmetrical space yields similar generated state space to the base configuration.

6.2 Coverage

Table 2 shows the coverage (number of problems solved) for the base and experimental configurations. The experimental configuration performs as well or better than the baseline for all domains, solving a superset of the instances that the baseline solves. In the Driverlog, Satellites and Pipes domains the experimental configuration increased coverage dramatically. Across the 198 problems we evaluated, coverage rose from 30.3% in the base configuration to 39.3% in the experimental configuration. Because these transformations are only applied in domains where semaphores or envelopes exist, coverage in other domains is unaffected.

The improvements in coverage are a reflection of the reduction in states generated. The Driverlog and Pipes domain, which experienced larger reductions on smaller problems, increased coverage as their respective problem sets scaled.

6.3 Search Time

The graphs of *states generated* and *search time* in Figure 5 have a similar spread, with a downward shift for points in the search time graph. To understand this shift, Table 3 presents the states generated per second. The experimental condition has an average 1.69 fold increase in time taken per state compared to the baseline.

The increased time per state is a result of the two types of scheduler used. The base configuration solves a P-time scheduling problem each time a state is generated whereas the experimental configuration solves an NP-complete scheduling problem. As a result of the difference in complexity, it is to be expected that the CP solver takes longer per state.

Table 3 States per second, and ratio (base/experimental). TMS excluded due to insufficient data.

IPC Domain	Base	Experimental	Ratio
Cafe	882	992	0.89
Crew Planning	747	759	0.98
Driverlog	891	204	4.37
Match	8696	5703	1.52
Pipes	1339	1224	1.09
Turn and Open	354	1184	0.3
Satellites	475	178	2.67
Average	**1912**	**1464**	**1.69**

A sufficiently large reduction in the number of states generated results in a reduction in the search time, in spite of the increased overhead per state, as seen by the solutions that timed-out for the *base configuration*.

In domains where the experimental configuration outperforms the base configuration in states per second (Cafe, Crew Planning and Turn and Open), we attribute this to the heavily constrained nature of the scheduling problems within these problems. This constraining makes the search space for the CP solver significant smaller when compared to other problems.

6.4 IPC Benchmark Score

In the temporal track of the IPC, a benchmark score is used to compare planners [17]. For a given problem, let T^* be the minimum search time in seconds required by any planner to solve the problem. A planner that solves the problem in search time T (in seconds) gets a score of $\frac{1}{1+log_{10}(T/T^*)}$. If a configuration does not solve a problem, it receives a score of 0. Search times of less than one second are rounded up.

Across the domains presented in this evaluation, the base configuration achieved a score of 41.6, whilst the experimental configuration achieved one of 78.

To restate, these transformations and the CP scheduler are only applied in problems where a semaphore or envelope fact is detected. Thus, these transformations represent a net improvement on the IPC score of the base planner. The performance of OPTIC in all other domains remains unaffected.

6.5 Makespan

Makespan is broadly equivalent across all mutually solved problems. This is primarily because the envelope actions somewhat artificially determine the makespan of the plan, even if the conditioner actions are scheduled more efficiently, so the scope for improving makespan is limited in most of domains. The one exception is a slight reduction in makespan for the experimental configuration in the Turn and Open domain. The difference can be attributed to the reallocation of disjunctive reasoning to the CP scheduler.

States generated during search by the base configuration are more constrained and have a stricter ordering than propositionally and numerically equivalent states generated by the experimental configuration. A single state represents more partial-orderings of the relaxed plan that achieves it in the experimental configuration. This can result in a reduction or increase of the makespan depending on the solution that the CP scheduler finds to the envelope allocation and mutual exclusion problems.

6.6 Evaluating Individual Abstractions

As part of our evaluation, we also evaluated abstractions individually. We saw no significant performance improvements from abstracting envelopes or semaphores alone: our improvements lean heavily on the interactions between envelopes and semaphores, in particular, the constraining effect of envelopes relative to the consumptive effect of semaphores.

We also evaluated using the envelope and semaphore abstractions without the time tracking variables defined in Section 5. Experimentally, we saw that the number of expanded states is 93% of that of the base configuration compared to the experimental configuration's 48%. The overall run-time was similarly impacted. When excluding Time Tracking Variables, there was no increase in coverage or improvement in makespan compared to the base configuration. These results suggest that the enhanced search guidance provided by the novel communication from the sub-solver embodied in these variables is a key to the improved performance that we observed.

7 Conclusion

By abstracting semaphores and envelopes, we remove two forms of disjunctive temporal reasoning from the planning level that cause exponential state space growth in temporal-numeric planning problems. Introducing new numeric variables to represent the available time within envelope achievers and using these variables as a means to communicate remaining envelope time to the planner further guides search and further reduces the state space.

The reductions in search space show that there are significant improvements to be made in more complex decomposition approaches. This new abstraction and decomposition demonstrates that a more expressive scheduler can improve coverage, reduce search space and reduce search times. This work opens the door to further exploration of how planning decisions are divided between task planning and CP-based scheduling and how an appropriate division, further exploiting the strengths of the CP sub-solver, can be used to yield improvements in coverage and speed.

References

1 Javier Barreiro, Matthew Boyce, Minh Binh Do, Jeremy D. Frank, Michael Iatauro, Tatiana Kichkaylo, Paul Henry Morris, James C. Ong, Emilio Remolina, Tristan B. Smith, and David E. Smith. Europa : A platform for ai planning, scheduling, constraint programming, and optimization. In *ICAPS 2012*, 2012. URL: https://icaps12.icaps-conference.org/demo/Barreiro_et_al_abs.pdf.

2 J. Benton, Amanda Jane Coles, and Andrew Coles. Temporal planning with preferences and time-dependent continuous costs. In Lee McCluskey, Brian Charles Williams, José Reinaldo Silva, and Blai Bonet, editors, *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*, ICAPS'12, pages 2–10. AAAI, 2012. URL: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4699.

3 Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. Temporal planning in domains with linear processes. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1671–1676, January 2009. URL: http://ijcai.org/Proceedings/09/Papers/279.pdf.

4 Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *Proceedings of the 20th International Conference on Automated Planning and Scheduling,*

ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010, pages 42–49. AAAI, January 2010. URL: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS10/paper/view/1421.

5 Amanda Jane Coles and Andrew Ian Coles. Have I been here before? state memoization in temporal planning. In Amanda Jane Coles, Andrew Coles, Stefan Edelkamp, Daniele Magazzeni, and Scott Sanner, editors, *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016*, pages 97–105. AAAI Press, 2016. URL: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13187.

6 Andrew Coles, Maria Fox, Derek Long, and Amanda Smith. Planning with problems requiring temporal coordination. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 892–897. AAAI Press, January 2008. URL: http://www.aaai.org/Library/AAAI/2008/aaai08-142.php.

7 Minh Binh Do and Subbarao Kambhampati. Sapa: A multi-objective metric temporal planner. *J. Artif. Intell. Res.*, 20:155–194, 2003. doi:10.1613/jair.1156.

8 Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. AAAI, January 2009. doi:10.1007/978-3-642-25116-0_6.

9 Maria Fox and Derek Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.*, 20:61–124, 2003. doi:10.1613/jair.1129.

10 Antonio Garrido, Marlene Arangú, and Eva Onaindia. A constraint programming formulation for planning: from plan scheduling to plan generation. *J. Sched.*, 12(3):227–256, June 2009. doi:10.1007/s10951-008-0083-7.

11 Keith Halsey, Derek Long, and Maria Fox. CRIKEY - a temporal planner looking at the integration of scheduling and planning. In *Proceedings of the Workshop on Integration Scheduling Into Planning at Thirteenth International Conference on Automated Planning and Scheduling*, January 2003.

12 Jörg Hoffmann. FF: the fast-forward planning system. *AI Mag.*, 22(3):57–62, September 2001. doi:10.1609/aimag.v22i3.1572.

13 Jörg Hoffmann. The metric-ff planning system: Translating ”ignoring delete lists” to numeric state variables. *J. Artif. Intell. Res.*, 20:291–341, 2003. doi:10.1613/jair.1144.

14 Richard Howey and Derek Long. Val's progress: The automatic validation tool for PDDL2.1 used in the International Planning Competition. In *Proceedings of the ICAPS 2003 workshop on "The Competition: Impact, Organization, Evaluation, Benchmarks"*, November 2003.

15 Wen-Yang Ku and J Christopher Beck. Revisiting off-the-shelf mixed integer programming and constraint programming models for job shop scheduling. *Computers & Operations Research*, 73:165–173, 2016.

16 Jean-Charles Régin. Global constraints: A survey. In *Hybrid Optimization: The Ten Years of CPAIOR*, pages 63–134, New York, 2011. Springer.

17 Mauro Vallati, Lukás Chrpa, Marek Grzes, Thomas Leo McCluskey, Mark Roberts, and Scott Sanner. The 2014 international planning competition: Progress and trends. *AI Mag.*, 36(3):90–98, September 2015. doi:10.1609/aimag.v36i3.2571.

18 Vincent Vidal and Hector Geffner. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artif. Intell.*, 170(3):298–335, March 2006. doi:10.1016/j.artint.2005.08.004.

Improved Bounds of Integer Solution Counts via Volume and Extending to Mixed-Integer Linear Constraints

Cunjing Ge
National Key Laboratory for Novel Software Technology, Nanjing University, China
School of Artificial Intelligence, Nanjing University, China

Armin Biere
University of Freiburg, Germany

Abstract

Solution counting and solution space integration over linear constraints are important problems with many applications. Previous works addressed either only counting integer points in polytopes (integer counting) with integer variables or alternatively only computing the volume of polytopes (solution space integration) on variables over the reals, including approximating the integer count via a polytope's volume. We are not aware of a non-trivial algorithm which addresses the mixed case, where linear constraints are over mixed integer and real variables. In this paper, we propose a new randomized algorithm to approximate guarantees (bounds) of integer solution counts. Then we present upper and lower bounds for solution space integration over mixed-integer linear constraints. Thus, proposed algorithms can be extended to mixed-integer cases as well. The experiments show that approximations are often very close to exact results in practice, and bounds approximated by our algorithm are often tight and useful.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Integer Solution Counting, Mixed-Integer Linear Constraints, #SMT(LA) Problems, Volume of Polytopes

Digital Object Identifier 10.4230/LIPIcs.CP.2024.13

Supplementary Material *Software*: `https://github.com/bearben/mixintcount/`
archived at `swh:1:dir:9101f2a1faa2381bc4779dff649915a21ff3cbae`

Funding *Cunjing Ge*: Cunjing Ge is supported by the National Natural Science Foundation of China (62202218), and is sponsored by CCF-Huawei Populus Grove Fund (CCF-HuaweiFM202309).

1 Introduction

As one of the most fundamental types of constraints, linear constraints (LCs) have been studied thoroughly in many areas. Counting solutions over LCs has also many applications, such as counting-based search [27, 32], simple temporal planning [16] and probabilistic program analysis [14, 23]. Moreover, it can be incorporated into DPLL (T)-based #SMT (LA) counters [12] as a core subroutine.

Since a set of LCs corresponds to a convex polytope, counting integer solutions over LCs is equivalent to counting integer points inside the polytope. For real solutions, the counting problem is turned into computing the polytope's volume, which is defined by the Lebesgue measure. Naturally, we may be interested in the solution counting problem over mixed-integer variables. In this paper, we will show that it is a problem of computing the integration of solution space, and then study the methods for approximating such integrations. We will call such a problem *solution space integration* for short.

Solution counting problems over LCs were proved to be #P-hard [29]. Barvinok [1, 2] introduced an algorithm for integer counting. Based on it, tools LattE [18] and barvinok [30] were implemented, which are still state-of-the-art. For volume of polytopes, tool Vinci [5] is an implementation and combination of several volume computation algorithms. The Multiphase Monte-Carlo algorithm [9, 21] is a polynomial time volume approximation algorithm for convex bodies. The tool Polyvest [10] is a scalable implementation of the Multiphase Monte-Carlo algorithm over polytopes. A more recent work [11] studied the relation between the count of inner integer points and the volume of a polytope. They proposed an algorithm called Vol2Lat to approximate integer counts via volume. This work inspired us to investigate the relationship among solution space integration, a polytope's volume and inner integer points count. For example, let us consider the following simplified formula F extracted from a scheduling problem:

$$(2a + 0.3b \leq t) \wedge (1 \leq a \leq 32) \wedge (1 \leq b \leq 32) \wedge (0 < t < 50),$$

where a and b are the numbers of scheduled tasks A and B, the coefficient 2 and 0.3 are the time cost of two tasks, t is the time limit, which is a real variable. The solution space integration of F is 14204.5. Therefore, when we uniformly pick an assignment (a_0, b_0, t_0) satisfying the range constraints, the probability that (a_0, b_0, t_0) satisfies F is $14204.5/(32 * 32 * 49) \approx 0.283$. Assuming $a, b, t \in \mathbb{R}$, the volume of its solution space is 14418.9. Assuming $a, b, t \in \mathbb{Z}$, the integer solution count is 15261. From experiments (see Section 5), we observe that the values of volume and integer count are usually close to the solution space integration. Naturally, given a set of mixed-integer LCs, we would like to investigate whether solution space integrations can be approximated via the volume or the integer lattice count of the corresponding polytope. The contributions of this paper are the following:

- We propose a new randomized algorithm to approximate bounds of integer solution counts of a set of linear constraints via a polytope's volume. It returns upper and lower bounds of integer solution counts with respect to a given confidence.
- We introduce and prove bounds for the solution space integration, volume and integer solution count on a set of LCs with mixed-integer, pure real and pure integer variables respectively. The bounds show when the three values are close to each other in theory. Thus, our new algorithm can be directly extended to approximate bounds of solution space integration on mixed-integer cases.
- Experiments show that our approach is promising over mixed-integer cases and also outperforms the existing bound approximation algorithm Vol2Lat, including instances generated from program analysis.

2 Background

2.1 Notations and Preliminaries

A Linear Constraint (LC) can be written in the form $\sum_{i=1}^{n} a_i x_i \ \texttt{op}\ b$, where x_is are numeric variables, a_is and b are real coefficients, and $\texttt{op} \in \{<, \leq, >, \geq, =\}$. From a geometric point of view, a LC is an n-dimensional halfspace, and a finite set of LCs is a polytope. Thus a set of LCs F corresponds to a polytope P which is in the form of

$$P = \{\vec{x} \in \mathbb{R}^n : A\vec{x} \leq \vec{b}\}.$$

Naturally, $\mathbb{Z}^n$ represents the set of all integer lattices (points with all integer coordinates). Thus integer models of the linear constraints can be represented by $\{\vec{x} \in \mathbb{Z}^n : A\vec{x} \leq \vec{b}\}$. It is the same as the integer points inside the corresponding polytope, i.e.,

$$\{\vec{x} \in \mathbb{Z}^n : A\vec{x} \leq \vec{b}\} = P \cap \mathbb{Z}^n.$$

Now we consider Mixed-Integer Linear Constraints (MILC) whose variables include not only reals but also integers. We know that changing the sequence of variables of $\vec{x}$ will not affect the size of the solution space of a set of LCs, such as, exchanging two variables x_i, x_j and their coefficients a_i, a_j. So without loss of generality, a set of MILCs F can be written in the form $A\vec{x} = A_1\vec{x_I} + A_2\vec{x_R} \leq \vec{b}$, where $A = [A_1 A_2]$, $\vec{x} = [\vec{x_I}\vec{x_R}]$ and $\vec{x_I}, \vec{x_R}$ are subsets of integer and real variables of $\vec{x}$ respectively.

▶ **Definition 1.** *Given a set of MILCs F, which corresponds to a polytope P.*

- *Let* $\texttt{vol}(P)$ *denote the volume of P, i.e.,* $\texttt{vol}(P) = \int_{\vec{x}\in P} 1 \, \mathrm{d}\vec{x}$.
- *Let* $\texttt{lat}(P)$ *denote the count of integer lattices in P, i.e.,* $\texttt{lat}(P) = |P \cap \mathbb{Z}^n|$.
- *Let $n_I = |\vec{x_I}|$ and $n_R = |\vec{x_R}|$. Obviously, $n = n_I + n_R$.*
- *Let $\mathcal{M}(F) = \{\vec{x} = [\vec{x_I}\vec{x_R}] \in \mathbb{Z}^{n_I} \times \mathbb{R}^{n_R} : A_1\vec{x_I} + A_2\vec{x_R} \leq \vec{b}\} = P \cap \mathbb{Z}^{n_I} \times \mathbb{R}^{n_R}$ denote the solution space of F.*
- *Let $\mathcal{M}_{\mathcal{I}}(F) \subset \mathbb{Z}^{n_I}$ and $\mathcal{M}_{\mathbb{R}}(F) \subset \mathbb{R}^{n_R}$ denote the projection from $\mathcal{M}(F)$ to variables over $\vec{x_I}$ and $\vec{x_R}$ respectively.*
- *Given an integer assignment $\vec{\alpha_I}$ over $\vec{x_I}$. Let $F\{\vec{x_I} = \vec{\alpha_I}\}$ denote the remaining constraints of F by assigning $\vec{\alpha_I}$ to $\vec{x_I}$, i.e., $F\{\vec{x_I} = \vec{\alpha_I}\} = A_1\vec{\alpha_I} + A_2\vec{x_R} \leq \vec{b}$.*
- *Let* $\texttt{integral}(F)$ *denote the integral on $\mathcal{M}(F)$. In detail,*

$$\begin{aligned}\texttt{integral}(F) &= \sum_{\vec{\alpha_I}\in\mathcal{M}_{\mathcal{I}}(F)} \int_{\vec{x}\in\mathcal{M}(F\{\vec{x_I}=\vec{\alpha_I}\})} 1 \, \mathrm{d}\vec{x} &(1)\\ &= \sum_{\vec{\alpha_I}\in\mathcal{M}_{\mathcal{I}}(F)} \texttt{integral}(F\{\vec{x_I} = \vec{\alpha_I}\}). &(2)\end{aligned}$$

Note that $\mathcal{M}(F\{\vec{x_I} = \vec{\alpha_I}\})$ is essentially a polytope in n_R-dimensional space. Let $P_{\vec{\alpha_I}}$ represent the corresponding polytope of $F\{\vec{x_I} = \vec{\alpha_I}\}$. We have $\texttt{integral}(F\{\vec{x_I} = \vec{\alpha_I}\}) = \texttt{vol}(P_{\vec{\alpha_I}})$. Consider an assignment $\vec{\beta_I}$ on $\vec{x_I}$ s.t. $\vec{\beta_I} \notin \mathcal{M}_{\mathcal{I}}(F)$, then $\texttt{vol}(P_{\vec{\beta_I}})$ must be zero, otherwise, $\exists\vec{\beta_R} \in \mathcal{M}(F\{\vec{x_I} = \vec{\beta_I}\})$ and $[\vec{\beta_I}\vec{\beta_R}]$ would be a solution of F which contradicts with $\vec{\beta_I} \notin \mathcal{M}_{\mathcal{I}}(F)$. Therefore, Equation (2) is equivalent with

$$\texttt{integral}(F) = \sum_{\vec{\alpha_I}\in S} \texttt{vol}(P_{\vec{\alpha_I}}), \forall\mathcal{M}_{\mathcal{I}}(F) \subseteq S \subset \mathbb{Z}^{n_I}. \tag{3}$$

It also indicates that $\vec{\alpha_I}$s can be enumerated in a looser space S than $\mathcal{M}_{\mathcal{I}}(F)$.

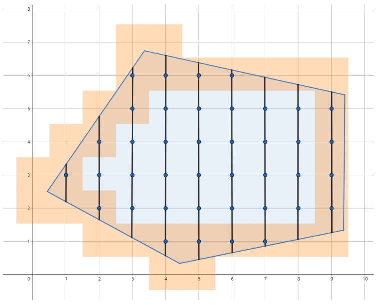


Figure 1 An example over two variables x and y, where x is an integer variable and y is a real variable. The integration $\texttt{integral}(F)$ is the sum of lengths of black lines parallel to y-axis. The count $\texttt{lat}(P) = 40$ is the number of dots on those black lines. $\mathrm{C}(P)$ is the set of orange and gray squares, $\mathrm{C}(\mathrm{B}(P))$ is the set of orange squares, and thus the set of gray squares can be represented by $\mathrm{C}(P) \setminus \mathrm{C}(\mathrm{B}(P))$. $\texttt{union}(\mathrm{C}(P))$ is the union space of orange and grey squares.

▶ **Definition 2.** *An* ***integer-cube*** *is a unit-cube whose center is an integer point. Given an integer point $\vec{\alpha}$ and a polytope P.*

- *Let* $\texttt{cube}(\vec{\alpha})$ *denote the integer-cube centered at $\vec{\alpha}$, i.e.,*

$$\texttt{cube}(\vec{\alpha}) = \{\vec{x} \in \mathbb{R}^n : \alpha_i - \frac{1}{2} \leq x_i \leq \alpha_i + \frac{1}{2}, i = 1, \dots, n\}.$$

- *Let* $\mathrm{C}(P)$ *represent the set of all integer-cubes which intersect with P.*
- *Let* $\mathrm{C}(\mathrm{B}(P))$ *represent the set of all integer-cubes which intersects with* $\mathrm{B}(P)$*, where* $\mathrm{B}(P)$ *is the boundary (facets) of P.*
- *Let* $\texttt{union}(C)$ *denote the union $\bigcup_{\kappa \in C} \kappa$, where C is a set of integer-cubes.*

Note that each integer-cube corresponds to a unique integer point and its volume is 1. Therefore, the integer-cube is introduced for bridging the gap between the volume and the integer count. Figure 1 is an example of integer-cubes, $\mathrm{C}(P)$, $\mathrm{C}(\mathrm{B}(P))$, etc.

2.2 Approximating Lattice Counts via Polytope's Volume

Ge et al. [11] observed that the lattice count and the volume of a given polytope are often close. They also pointed out that there exist cases in which lattice counts and volume are greatly different. For example, a very 'thin' rectangle whose sides are parallel to the coordinates and the short side lies in interval $(0, 1)$. Then there is no integer point in it, but its volume can be arbitrarily large as the long side stretches. Therefore, they focused on the distance between the count and the volume, and further proposed a method to approximate the count by the volume. The following theorems are their main results.

▶ **Lemma 3.** *Both* $\texttt{vol}(P), \texttt{lat}(P)$ *are in the interval* $[|\mathrm{C}(P)| - |\mathrm{C}(\mathrm{B}(P))|, |\mathrm{C}(P)|]$.

▶ **Theorem 4.** $|\texttt{vol}(P) - \texttt{lat}(P)| \leq |\mathrm{C}(B(P))|$.

▶ **Theorem 5.** $|\mathrm{C}(\mathrm{B}(P))| \leq 2\sum_{i=1}^{n}\prod_{i\neq j}(M_j(P) - m_j(P))$*, where* $M_i(P) = \lfloor \max\{x_i| \vec{x} \in P\} + 1 \rfloor$ *and* $m_i(P) = \lceil \min\{x_i|\vec{x} \in P\} - 1 \rceil$.

Take Figure 1 as the example. We observe that each gray square contains exactly one integer point in the polytope while orange squares maybe not, so it suggests that the differences between $\texttt{vol}(P)$, $\texttt{lat}(P)$ and $|\mathrm{C}(P)|$ are related to those orange squares, i.e., $|\mathrm{C}(\mathrm{B}(P))|$. It is the intuition behind the proof of Lemma 3. Based on Lemma 3, it is easy to obtain Theorem 4 that the difference between $\texttt{vol}(P)$ and $\texttt{lat}(P)$ is bounded by $|\mathrm{C}(\mathrm{B}(P))|$. Then by Theorem 5, a looser bound $2\sum_{i=1}^{n}\prod_{i\neq j}(M_j(P) - m_j(P))$ is proved which is easier to be computed in practice. As a result, the count of lattices in a given polytope P can be approximated via its volume $\texttt{vol}(P)$, i.e.,

$$|\texttt{vol}(P) - \texttt{lat}(P)| \leq 2\sum_{i=1}^{n}\prod_{i\neq j}(M_j(P) - m_j(P)).$$

However, according to experimental results in [11], the bound $2\sum_{i=1}^{n}\prod_{i\neq j}(M_j(P) - m_j(P))$ may be very loose, which prevents some applications. Naturally, we are interested in improving the above bounds. In this paper, we first propose a new method to approximate $|\mathrm{C}(\mathrm{B}(P))|$ which is usually much tighter in experiments. Then we extend Theorem 4 to mixed-integer cases to approximate $\texttt{integral}(F)$ by $\texttt{vol}(P)$, where P is the corresponding polytope of F.

2.3 Sampling in Polytopes

The classical algorithm [4, 22, 20, 19, 8, 12] for sampling real points in a polytope P is presented in Algorithm 1. It first calls a rounding method, such as the Shallow-β-Cut Ellipsoid method [15], to find an affine transformation T, s.t., $\texttt{ball}(0,1) \subset T(P) \subset \texttt{ball}(0,2n)$, where $\texttt{ball}(0,r)$ is a radius r ball centered at origin. Then it employs a hit-and-run random walk method to generate real points in $T(P)$. It finally returns sample points in P by applying the inverse transformation T^{-1}. Intuitively, T transforms a very "thin" polytope into a well-bounded one. Thus, random walks will mix (converge to limiting distribution) faster on the new polytope $T(P)$. In addition, it guarantees that $T(P)$ contains the origin, which will be used as the start point for the random walks.

Algorithm 1 Real Points Sampling Algorithm.

```
Function Sampling_Real($P$, $\vec{x_0}$, $w$)
    $T \leftarrow$ Ellipsoid($P$);
    $\vec{x} \leftarrow T(\vec{x_0})$;
    while $w > 0$ do
        $\vec{x} \leftarrow$ Hit-and-run($T(P)$, $\vec{x}$);
        $w \leftarrow w - 1$;
    return $T^{-1}(\vec{x})$;
```

The Hit-and-run random walk method was first introduced in [4], where its limiting distribution was proved to be uniform. It was employed and improved for approximating a polytope's volume by [22, 20]. A variation called Coordinate Directions Hit-and-run is found more efficient by experiments [8, 12]. Thus, we also adopt this variation in our paper, which consists of the following steps:

Step 1. Given a point $\vec{x_0} \in P$, it first selects a line L uniformly over n coordinate directions (parallel to the axes) which passes through the point $\vec{x_0}$.

Step 2. It then chooses the next point $\vec{x_1}$ uniformly on the segment of L in P.

Step 3. Repeat above steps w times, $\vec{x_w}$ is finally obtained and adopted.

Earlier works [20] proved that Hit-and-run method mixes in $w = O(n^2)$ steps for a random initial point and $O(n^3)$ steps for a fixed initial point. However, further numerical studies [19, 12] reported that $w = n$ is sufficient for nearly uniformly sampling in polytopes with dozens of dimensions.

3 Our Approach

In this section, we first introduce our new algorithms for approximating $\texttt{lat}(P)$ via $\texttt{vol}(P)$. Then we extend Theorem 4 and algorithms to mixed-integer cases, i.e., approximating $\texttt{integral}(F)$ via $\texttt{vol}(P)$.

3.1 The Framework of Bounds Approximation Algorithm

To compute bounds of $\texttt{lat}(P)$, i.e., $|\mathrm{C}(P)| - |\mathrm{C}(\mathrm{B}(P))|$ and $|\mathrm{C}(P)|$, we introduce a Monte-Carlo algorithm which samples points in $\texttt{union}(\mathrm{C}(P))$ and then counts the number of points that lie in P and $\texttt{union}(\mathrm{C}(\mathrm{B}(P)))$.

▶ **Theorem 6.** *Suppose X is a set of sample points uniformly generated from* $\texttt{union}(\mathrm{C}(P))$*. Let $\hat{r}_1 = \frac{|X \cap P|}{|X|}$ and $\hat{r}_2 = 1 - \frac{|X \cap \texttt{union}(\mathrm{C}(\mathrm{B}(P)))|}{|X|}$. Then $|\mathrm{C}(P)| - |\mathrm{C}(\mathrm{B}(P))| = \texttt{vol}(P) \cdot \lim_{|X|\to\infty} \frac{\hat{r}_2}{\hat{r}_1}$ and $|\mathrm{C}(P)| = \texttt{vol}(P) \cdot \lim_{|X|\to\infty} \frac{1}{\hat{r}_1}$.*

Proof. Let $r_1 = \frac{\texttt{vol}(P)}{|\mathrm{C}(P)|}$ and $r_2 = 1 - \frac{|\mathrm{C}(\mathrm{B}(P))|}{|\mathrm{C}(P)|}$. Since

$$\texttt{vol}(\texttt{union}(\mathrm{C}(P))) = \sum_{\kappa \in \mathrm{C}(P)} \texttt{vol}(\kappa) = |\mathrm{C}(P)|,$$

then $r_1 = \frac{\texttt{vol}(P)}{\texttt{vol}(\texttt{union}(\mathrm{C}(P)))}$. Note that sampling in $\texttt{union}(\mathrm{C}(P))$ in uniform and then counting the number of points that lie in P is a Bernoulli process. Thus $\hat{r}_1$ is the estimated proportion of successes for r_1. Therefore, $\lim_{|X|\to\infty} \hat{r}_1 = r_1$. Similarly, we could find that $\lim_{|X|\to\infty} \hat{r}_2 = r_2$, and $\lim_{|X|\to\infty} \frac{\hat{r}_2}{\hat{r}_1} = \frac{r_2}{r_1} = \frac{|\mathrm{C}(P)| - |\mathrm{C}(\mathrm{B}(P))|}{\texttt{vol}(P)}$. ◀

From Lemma 3 and Theorem 6, we know that $\frac{\hat{r}_2}{\hat{r}_1} \cdot \texttt{vol}(P)$ and $\frac{1}{\hat{r}_1} \cdot \texttt{vol}(P)$ are the approximations of the lower bound and the upper bound of $\texttt{lat}(P)$ respectively. Therefore, we aim to approximate $\frac{1}{\hat{r}_1}$ and $\frac{\hat{r}_2}{\hat{r}_1}$. Since sampling points is a Bernoulli trial, then $\hat{r}_1 = \frac{|X \cap P|}{|X|} \in [0, 1]$ is an approximation of proportion of a binomial distribution, and $\hat{r}_2$ as well. The confidence interval (CI) of r_1 is thus a binomial CI. The well-known $1 - \delta$ normal approximation CI on r_1 is

$$\hat{r}_1 - z_{1-\delta/2} \sqrt{\frac{\hat{r}_1(1 - \hat{r}_1)}{|X|}} \le r_1 \le \hat{r}_1 + z_{1-\delta/2} \sqrt{\frac{\hat{r}_1(1 - \hat{r}_1)}{|X|}}, \tag{4}$$

where $z_{1-\delta/2}$ is the $1 - \delta/2$ quantile of a standard normal distribution. Intuitively, CIs of r_1 and r_2 can be used as the algorithm's stopping criterion. Let $\hat{e}_1 \equiv z_{1-\delta/4} \sqrt{\frac{\hat{r}_1(1-\hat{r}_1)}{|X|}}$ and $\hat{e}_2 \equiv z_{1-\delta/4} \sqrt{\frac{\hat{r}_2(1-\hat{r}_2)}{|X|}}$ which are the margin errors. We obtain $1 - \delta/2$ CIs of r_1 and r_2: $\hat{r}_1 - \hat{e}_1 \le r_1 \le \hat{r}_1 + \hat{e}_1$ and $\hat{r}_2 - \hat{e}_2 \le r_2 \le \hat{r}_2 + \hat{e}_2$. Then $r_1 \ge \hat{r}_1 - \hat{e}_1$ and $r_2 \le \hat{r}_2 + \hat{e}_2$ with probability at least $1 - \delta/4$. Thus we have $\frac{\hat{r}_2 - \hat{e}_2}{\hat{r}_1 + \hat{e}_1} \le \frac{r_2}{r_1}$ with probability at least $1 - \delta/2$. By the same way, we obtain the intervals

$$\frac{\hat{r}_2 - \hat{e}_2}{\hat{r}_1 + \hat{e}_1} \le \frac{r_2}{r_1} \le \frac{\hat{r}_2 + \hat{e}_2}{\hat{r}_1 - \hat{e}_1} \text{ and } \frac{1}{\hat{r}_1 + \hat{e}_1} \le \frac{1}{r_1} \le \frac{1}{\hat{r}_1 - \hat{e}_1}, \tag{5}$$

so that $\frac{\hat{r}_2}{\hat{r}_1}$ and $\frac{1}{\hat{r}_1}$ lie in them with probability at least $1 - \delta$.

▶ **Theorem 7.** *In Theorem 6, if $|X| \ge z^2_{1-\delta/4} \cdot (\frac{1}{\epsilon} \cdot \sqrt{\frac{1-r_2}{r_2}} + \frac{1-\epsilon}{\epsilon} \cdot \sqrt{\frac{1-r_1}{r_1}})^2$ and $|X| \ge z^2_{1-\delta/4} \cdot (\frac{1+\epsilon}{\epsilon})^2 \cdot \frac{1-r_1}{r_1}$, then $\mathrm{Prob}(|\frac{\hat{r}_2}{\hat{r}_1} - \frac{r_2}{r_1}| \le \epsilon \cdot \frac{r_2}{r_1}) \ge 1 - \delta$ and $\mathrm{Prob}(|\frac{1}{\hat{r}_1} - \frac{1}{r_1}| \le \epsilon \cdot \frac{1}{r_1}) \ge 1 - \delta$.*

Proof. Note that $\frac{1}{\hat{r}_1 - \hat{e}_1} - \frac{1}{\hat{r}_1} \le \epsilon \cdot \frac{1}{\hat{r}_1} \iff \epsilon \hat{r}_1 \ge (1+\epsilon)\hat{e}_1 \iff |X| \ge z^2_{1-\delta/4} \cdot (\frac{1+\epsilon}{\epsilon})^2 \cdot \frac{1-\hat{r}_1}{\hat{r}_1} \approx z^2_{1-\delta/4} \cdot (\frac{1+\epsilon}{\epsilon})^2 \cdot \frac{1-r_1}{r_1}$. From Equation 5, we have $\mathrm{Prob}(|\frac{1}{\hat{r}_1} - \frac{1}{r_1}| \le \epsilon \cdot \frac{1}{r_1}) \ge \mathrm{Prob}(\frac{1}{\hat{r}_1 - \hat{e}_1} - \frac{1}{\hat{r}_1} \le \epsilon \cdot \frac{1}{\hat{r}_1}) \ge 1 - \delta$. The proof of $|X| \ge z^2_{1-\delta/4} \cdot (\frac{1}{\epsilon} \cdot \sqrt{\frac{1-r_2}{r_2}} + \frac{1-\epsilon}{\epsilon} \cdot \sqrt{\frac{1-r_1}{r_1}})^2$ is similar. ◀

Theorem 7 discusses the relations among the confidence δ, the error ϵ and the number of samples $|X|$. Since $\frac{r_2}{r_1} \in (0, 1]$ and $\frac{1}{r_1} \in [1, \infty)$, the scale of samples $|X|$ may vary a lot with respect to the values of $\frac{r_2}{r_1}$ and $\frac{1}{r_1}$. Therefore, based on the proof of Theorem 7, we introduce a dynamical stopping criterion for our algorithm which checks the quality of the approximation whenever a sample is obtained. The algorithm stops when it found $\frac{\hat{r}_2}{\hat{r}_1} - \frac{\hat{r}_2 - \hat{e}_2}{\hat{r}_1 + \hat{e}_1} \le \epsilon \cdot \frac{\hat{r}_2}{\hat{r}_1}$ and $\frac{1}{\hat{r}_1 - \hat{e}_1} - \frac{1}{\hat{r}_1} \le \frac{\epsilon}{\hat{r}_1}$, which guarantees $\mathrm{Prob}(|\frac{\hat{r}_2}{\hat{r}_1} - \frac{r_2}{r_1}| \le \epsilon \cdot \frac{r_2}{r_1}) \ge 1 - \delta$ and $\mathrm{Prob}(|\frac{1}{\hat{r}_1} - \frac{1}{r_1}| \le \epsilon \cdot \frac{1}{r_1}) \ge 1 - \delta$.

Algorithm 2 MixIntCount.

```
Input: $P$
Parameter: $\epsilon$, $\delta$, $w$, $N$
Output: $\texttt{lb}(P)$, $\texttt{ub}(P)$
$X \leftarrow \emptyset$;
Initialize $\vec{x}$ with an arbitrary point in $P$;
while $|X| \leq N$ do
    $\vec{x} \leftarrow \text{Sampling}(P, \vec{x}, w)$;
    $X \leftarrow X \cup \{\vec{x}\}$;
    $\hat{r}_1 \leftarrow \frac{|X \cap P|}{|X|}$;
    $\hat{r}_2 \leftarrow 1 - \frac{|X \cap \texttt{union}(\mathrm{C}(\mathrm{B}(P)))|}{|X|}$;
    if $\frac{\hat{r}_2}{\hat{r}_1} - \frac{\hat{r}_2 - \hat{e}_2}{\hat{r}_1 + \hat{e}_1} \leq \epsilon \cdot \frac{\hat{r}_2}{\hat{r}_1}$ and $\frac{1}{\hat{r}_1 - \hat{e}_1} - \frac{1}{\hat{r}_1} \leq \frac{\epsilon}{\hat{r}_1}$ then break;
return $\texttt{vol}(P) \cdot \frac{\hat{r}_2 - \hat{e}_2}{\hat{r}_1 + \hat{e}_1}$, $\texttt{vol}(P) \cdot \frac{1}{\hat{r}_1 - \hat{e}_1}$;
```

The pseudocode of the our main framework is presented in Algorithm 2. The parameters ϵ, δ and N determine the accuracy and the confidence of approximations and the maximum number of samples respectively. Note that according to Theorem 7, $|X|$ could be a quite large number when $\frac{r_2}{r_1}$ is close to 0 or $\frac{1}{r_1} \gg 1$. In such cases, the approximations of bounds are meaningless, for example, $\frac{\texttt{lat}(P)}{\texttt{vol}(P)} \geq \frac{r_2}{r_1} \approx 0$. So we introduce a sampling limit N. The bounds would be rather meaningless when the algorithm reachs the limit N. The setting of parameters will be further discussed in Section 4. In general, Algorithm 2 returns the bounds of $\texttt{lat}(P)$, i.e., $\texttt{lb}(P) \leq \texttt{lat}(P) \leq \texttt{ub}(P)$ and $|\texttt{vol}(P) - \texttt{lat}(P)| \leq \texttt{ub}(P) - \texttt{lb}(P)$ with probability at least $1 - \delta$.

3.2 Sampling in Unions of Integer-cubes

To generate sample points in $\texttt{union}(\mathrm{C}(P))$ nearly uniformly, we combine Algorithm 1 with rejection sampling. Algorithm 3 presents the new sampling algorithm. It first enlarges P to obtain a new polytope P', such that P' contains all integer-cubes in $\mathrm{C}(P)$. Next it samples real points in P' by Algorithm 1. Then it accepts those in $\texttt{union}(\mathrm{C}(P))$. Obviously, the larger P', the lower probability of acceptance. Now a question arises:

- How to obtain such a P' that is as small as possible?

Algorithm 3 Sampling in $\texttt{union}(\mathrm{C}(P))$.

```
Function Sampling($P$, $\vec{x_0}$, $w$)
    $P' \leftarrow \text{Enlarging}(P)$;
    $\vec{x} \leftarrow \vec{x_0}$;
    while $true$ do
        $\vec{x} \leftarrow \text{Sampling\_Real}(P', \vec{x}, w)$;
        if $\vec{x} \in \texttt{union}(\mathrm{C}(P))$ then
            return $\vec{x}$;
```

Intuitively, we can obtain P' by shifting every facet H of P to H', s.t., the distance between H and H' is sufficient to contain an integer-cube. Minimizing this shifting distance is formulated into a linear programming (LP) problem, with constraints $\{-1 \leq x_i \leq 1\}$ and an objective, maximize $\vec{A_k}\vec{x}$, where $\vec{A_k}$ is the kth row of the matrix A. Let v_k be such maximum value for the kth LC. Then we shift it by adding v_k. The pseudocode of constructing P' is shown in Algorithm 4. The following theorem guarantees that P' obtained by Algorithm 4 contains union(C(P)).

▶ **Theorem 8.** *Given the kth LC $H_k \equiv \vec{A_k}\vec{x} \leq b_k$. Let $v_k = \max\{\vec{A_k}\vec{x} | -1 \leq x_i \leq 1, i = 1, \dots, n\}$ and $H'_k \equiv \vec{A_k}\vec{x} \leq b_k + v_k$. Then we have $\kappa \subset H'_k, \forall \kappa \in \mathrm{C}(H_k)$.*

Proof. Let $G_k = \vec{A_k}\vec{x} \leq 0$ and $G'_k = \vec{A_k}\vec{x} \leq v_k$. Then it is equivalent to prove $\kappa \subset G'_k, \forall \kappa \in \mathrm{C}(G_k)$. Assume $\exists$cube$(\vec{\alpha}) \in C(G_k)$ and $\exists \vec{p_{out}} \in$ cube$(\vec{\alpha})$ s.t., $\vec{p_{out}} \notin G'_k$. Let $c_{max} = \max\{\vec{A_k}\vec{x} | \vec{x} \in \texttt{cube}(\vec{\alpha})\}$ and $c_{min} = \min\{\vec{A_k}\vec{x} | \vec{x} \in \texttt{cube}(\vec{\alpha})\}$. Recall that cube$(\vec{\alpha}) = \{\alpha_i - \frac{1}{2} \leq x_i \leq \alpha_i + \frac{1}{2}\}$, we can find that $v_k = c_{max} - c_{min}$. Since cube$(\vec{\alpha}) \in C(G_k)$, then $\exists \vec{p_{in}} \in \texttt{cube}(\vec{\alpha}) \cap G_k$. Thus the hyperplane $\vec{A_k}\vec{x} = \vec{A_k}\vec{p_{out}}$ is outside of G'_k, and $\vec{A_k}\vec{p_{out}} > v_k$. Similarly, we could find that $\vec{A_k}\vec{p_{in}} \leq 0$, yielding the contradiction $v_k = c_{max} - c_{min} \geq \vec{A_k}\vec{p_{out}} - \vec{A_k}\vec{p_{in}} > v_k$. ◀

Algorithm 4 Enlarging P by shifting hyperplanes.

```
Function Enlarging($P$)
    for each $\vec{A_k}\vec{x} \leq b_k$ from $P$ do
        constraints ← $\{-1 \leq x_i \leq 1\}$;
        object ← $\vec{A_k}\vec{x}$;
        $v_i$ ← Simplex(object, constraints);
    return $\{A\vec{x} \leq \vec{b} + \vec{v}\}$;
```

3.3 Efficient Cube Checking

In Algorithm 2 and 3, we have to frequently check whether a point is in any integer-cube in C(P) or C(B(P)). So in this section, we focus on the following question.

- How to efficiently check whether a point $\vec{p}$ is in union(C(P)) or union(C(B(P)))?

We observe that if p is not on the boundary of an integer-cube, then $\vec{p} \in$ union(C(P)) iff cube($[\vec{p}]$) $\in$ C(P), where $[\vec{p}] = ([p_1], \dots, [p_n])$ is obtained by rounding numbers to integers. Since in practice, it is nearly impossible to generate a sample point right on the boundaries of cubes. We assume p does not sit on the boundary of an integer-cube in this section. Similarly, we have if p is not on the boundary of an integer-cube, $\vec{p} \in$ union(C(B(P))) iff cube($[\vec{p}]$) $\in$ C(B(P)). Furthermore, cube($[\vec{p}]$) $\in$ C(P) iff $\vec{p} \in P$ or cube($[\vec{p}]$) $\in$ C(B(P)). Since checking whether $\vec{p} \in P$ is trivial, we only have to find an efficient method to check whether cube($\vec{\alpha}$) $\in$ C(B(P)) with a given integer point $\vec{\alpha}$.

Algorithm 5 presents the method for fast cube checking. Note that v_k is the same as in Algorithm 4, that is, $v_k = \max\{\vec{A_k}\vec{x} | -1 \leq x_i \leq 1, i = 1, \dots, n\}$, where $\vec{A_k}$ is the kth row of A. Obviously, $v_k/2 = \max\{\vec{A_k}\vec{x} | -0.5 \leq x_i \leq 0.5, i = 1, \dots, n\}$. Since $\vec{\alpha}$ is the center of cube($\vec{\alpha}$), we have $\vec{A_k}\vec{\alpha} - v_k/2 = \min\{\vec{A_k}\vec{x} | \vec{x} \in \texttt{cube}(\vec{\alpha})\}$ and $\vec{A_k}\vec{\alpha} + v_k/2 = \max\{\vec{A_k}\vec{x} | \vec{x} \in \texttt{cube}(\vec{\alpha})\}$. If $\exists j$, s.t., $\vec{A_j}\vec{\alpha} - v_j/2 > b_j$, it indicates that cube($\vec{\alpha}$) is completely outside P. Otherwise, if $\exists j$, s.t., $\vec{A_j}\vec{\alpha} + v_j/2 \geq b_j$, it indicates that cube($\vec{\alpha}$) intersects with B(P).

Algorithm 5 Check whether $\texttt{cube}(\vec{\alpha}) \in \mathrm{C}(\mathrm{B}(P))$.

```
Function CubeOnBound($\vec{\alpha}, P$)
    flag ← $false$;
    for each $\vec{A_k}\vec{x} \le b_k$ from $P$ do
        if $\vec{A_k}\vec{\alpha} - v_k/2 > b_k$ then
            return $false$;
        if $\vec{A_k}\vec{\alpha} + v_k/2 \ge b_k$ then
            flag ← $true$;
    return flag;
```

3.4 Extending to Mixed-Integer Cases

In this section, we will introduce and prove the theoretical result on mixed-integer cases. Combined with Lemma 3, it not only provides bounds for $\texttt{integral}(F)$, but also shows when $\texttt{integral}(F)$ can be approximated by $\texttt{lat}(P)$ and $\texttt{vol}(P)$.

▶ **Theorem 9.** $|\mathrm{C}(P)| - |\mathrm{C}(\mathrm{B}(P))| \le \texttt{integral}(F) \le |\mathrm{C}(P)|$.

Proof. According to Equation (3), the theorem is equivalent with

$$|\mathrm{C}(P)| - |\mathrm{C}(\mathrm{B}(P))| \le \sum_{\vec{\alpha_I} \in \mathcal{M}_{\mathcal{I}}(F)} \texttt{vol}(P_{\vec{\alpha_I}}) \le |\mathrm{C}(P)|. \tag{6}$$

From Lemma 3, we have $|\mathrm{C}(P_{\vec{\alpha_I}})| - |\mathrm{C}(\mathrm{B}(P_{\vec{\alpha_I}}))| \le \texttt{vol}(P_{\vec{\alpha_I}}) \le |\mathrm{C}(P_{\vec{\alpha_I}})|$. Then

$$\sum_{\vec{\alpha_I}} (|\mathrm{C}(P_{\vec{\alpha_I}})| - |\mathrm{C}(\mathrm{B}(P_{\vec{\alpha_I}}))|) \le \sum_{\vec{\alpha_I}} \texttt{vol}(P_{\vec{\alpha_I}}) \le \sum_{\vec{\alpha_I}} |\mathrm{C}(P_{\vec{\alpha_I}})|. \tag{7}$$

Given an arbitrary $\texttt{cube}(\vec{\alpha_R}) \in \mathrm{C}(P_{\vec{\alpha_I}})$. Obviously, $\exists \vec{\alpha_R}' \in \texttt{cube}(\vec{\alpha_R})$ such that $\vec{\alpha_R}' \in P_{\vec{\alpha_I}}$. Let $\vec{\alpha} = [\vec{\alpha_I}\vec{\alpha_R}]$ and $\vec{\alpha}' = [\vec{\alpha_I}\vec{\alpha_R}']$ which are concatenations of integer and real variables. Then $\vec{\alpha}'$ is an interior point in P and $\vec{\alpha}' \in \texttt{cube}(\vec{\alpha})$. It means $\texttt{cube}(\vec{\alpha}) \in \mathrm{C}(P)$. By this way, we could map $\mathrm{C}(P_{\vec{\alpha_I}})$ to $\mathrm{C}(P \cap \vec{\alpha_I} \times \mathbb{R}^{n_R}) \subset \mathrm{C}(P)$. Note that $\mathrm{C}(P \cap \vec{\alpha_I} \times \mathbb{R}^{n_R}) \cap \mathrm{C}(P \cap \vec{\alpha_I'} \times \mathbb{R}^{n_R}) = \emptyset$, and $\vec{\alpha_I} \neq \vec{\alpha_I'}$, then we have

$$\sum_{\vec{\alpha_I}} |\mathrm{C}(P_{\vec{\alpha_I}})| \le |\mathrm{C}(P)|. \tag{8}$$

Given an arbitrary $\texttt{cube}(\vec{\beta}) \in \mathrm{C}(P) \setminus \mathrm{C}(\mathrm{B}(P))$. Let $\vec{\beta} = [\vec{\beta_I}\vec{\beta_R}]$. Note that $\texttt{cube}(\vec{\beta}) \subset P$, it means $\texttt{cube}(\vec{\beta}) \cap \mathcal{M}(F) \neq \emptyset$. Thus $\vec{\beta_I} \in \mathcal{M}_{\mathcal{I}}(F)$. Then we obtain an unique integer-cube $\texttt{cube}(\vec{\beta_R}) \in \mathrm{C}(P_{\vec{\beta_I}}) \setminus \mathrm{C}(\mathrm{B}(P_{\vec{\beta_I}}))$. Similarly, we have

$$|\mathrm{C}(P)| - |\mathrm{C}(\mathrm{B}(P))| = |\mathrm{C}(P) \setminus \mathrm{C}(\mathrm{B}(P))| \le \sum_{\vec{\alpha_I}} (|\mathrm{C}(P_{\vec{\alpha_I}})| - |\mathrm{C}(\mathrm{B}(P_{\vec{\alpha_I}}))|). \tag{9}$$

Combine Equation (7) (8) and (9), then Equation (6) is obtained. ◀

Theorem 9 indicates that Algorithm 2 can be directly applied for solving the solution space integration problem on mixed-integer constraints, i.e., $\texttt{lb}(P) \le \texttt{integral}(F) \le \texttt{ub}(P)$.

4 Implementation

We implemented a prototype tool called MixIntCount in C++. MixIntCount employs GLPK for linear programming. It calls Vinci [5] and Polyvest [10] for polytopes' volume computation and approximation.

In our implementation, the Ellipsoid method is employed only once, and the affine transformation T will be reused by the real point sampling in Algorithm 3. Moreover, whenever a sample is obtained, it will immediately update the counts of samples that lie in P or $\texttt{union}(\mathrm{C}(\mathrm{B}(P)))$ and check whether the stopping criterion is satisfied.

Wilson score interval

For convenience, we introduced the normal approximation CI implied by the Central Limit Theorem for describing Algorithm 2 in Section 3. However, the normal CI suffers from problems of overshoot and zero-width intervals, e.g., r_1 and r_2 may sometimes be close to 0 or 1. In such circumstances, the Wilson score interval (or Wilson CI) [31] performs much better than the normal CI. Therefore, we replace Equation 4 by Wilson CI, and let MixIntCount compute Wilson CIs as the stopping criterion.

The setting of parameters

We choose parameters $\epsilon = 0.1$, $\delta = 0.05$, $w = n$, and $N = 100000$ for MixIntCount, and $\epsilon = 0.2, \delta = 0.05$ for Polyvest. Note that ϵ and δ are different in two tools. In PolyVest, ϵ and δ control the errors of volume approximations. In MixIntCount, ϵ and δ control the errors of lower and upper bounds. Since previous numerical studies [19, 12] reported that $w = n$ is sufficient for nearly uniformly sampling points in polytopes with dozens of dimensions, we also chose $w = n$ in our implementation. Experiments in Section 5 show that $N = 100000$ is sufficiently large, as the lower bounds $\texttt{lb}(P) \leq 0.001 \cdot \texttt{vol}(P)$ are useless when $|X|$ reaches N.

A straightforward method for mixed-integer cases

Based on the definition, i.e., Equation 2, it is easy to propose an algorithm for computing $\texttt{integral}(F)$. First, it enumerates all assignments $\vec{\alpha_I} \in \mathcal{M}_{\mathcal{I}}(F)$. Then for each $\vec{\alpha_I}$, it computes $\texttt{vol}(P_{\vec{\alpha_I}})$ by volume computation algorithms. Finally, $\texttt{integral}(F)$ is obtained by summing up $\texttt{vol}(P_{\vec{\alpha_I}})$. We implemented this method and called it ExactMI, which could provide exact integration. We adopted it as the baseline for performance comparison in our evaluation (Section 5). Note that in practice, we could try MixIntCount first, and if approximations of bounds are not tight enough, we would then employ ExactMI.

Approximating bounds on SMT(LA) formulas

We incorporated MixIntCount into the DPLL(T)-based #SMT(LA) counter [12] directly to approximate bounds of solution space integration of SMT(LA) formulas. Without loss of generality, an SMT(LA) formula ϕ with l Boolean variables, n numeric variables and m LCs can be formally represented as a Boolean formula $PS_\phi(b_1, \ldots, b_{m+l})$ together with definitions in the form: $b_i \equiv H_i, i = 1, \ldots, m$, where H_is are LCs. Then $b_{m+1}, \ldots, b_{m+l}$ are the pure Boolean variables of ϕ. An assignment $\vec{\gamma}$ of PS_ϕ is a vector $(\gamma_1, \ldots, \gamma_{m+l}) \in \mathbb{B}^{m+l}$, where γ_i is either 1 or 0. A partial assignment $\vec{\gamma}$ means there are some γ_is not assigned. Let $\texttt{bool}(\vec{\gamma})$ represent the vector $(\gamma_{m+1}, \ldots, \gamma_{m+l}) \in \mathbb{B}^l$ which corresponds to those pure Boolean variables. Let $H_{\vec{\gamma}} = \bigcup_{1 \le i \le m} H_{\vec{\gamma},i}$, where $H_{\vec{\gamma},i}$ is $\{H_i\}$ or $\{\neg H_i\}$ or $\emptyset$ if γ_i is 1 or 0 or not assigned respectively. Note that H_i and $\neg H_i$ are LCs, $H_{\vec{\gamma}}$ is the set of LCs that corresponds to $\vec{\gamma}$. Thus, an assignment $\vec{\mu}$ of ϕ consists of $(\vec{x}, \texttt{bool}(\vec{\gamma}))$, where $\vec{\gamma}$ is an assignment of PS_ϕ and $\vec{x}$ is a point. Let $\mathcal{M}_{\phi,\vec{\gamma}}$ represent $\mathcal{M}(H_{\vec{\gamma}}) \times \texttt{bool}(\vec{\gamma})$. The solution space of ϕ is then the union of sets, formally:

$$\mathcal{M}(\phi) = \bigcup_{\vec{\gamma} \in \mathcal{M}(PS_\phi)} \mathcal{M}(H_{\vec{\gamma}}) \times \texttt{bool}(\vec{\gamma}) = \bigcup_{\vec{\gamma} \in \mathcal{M}(PS_\phi)} \mathcal{M}_{\phi,\vec{\gamma}}. \tag{10}$$

To enumerate $\vec{\gamma} \in \mathcal{M}(PS_\phi)$, a DPLL(T)-based scheme was introduced:

- Step 1. Find a model $\vec{\mu}$ of ϕ by DPLL(T) algorithm. From Equation 10, there exists a partial assignment $\vec{\gamma} \in \mathcal{M}(PS_\phi)$, s.t., $\vec{\mu} \in \mathcal{M}_{\phi,\vec{\gamma}}$.
- Step 2. Conjunct ϕ with the negation formula G of partial assignment $\vec{\gamma}$, which would prevent the DPLL(T) algorithm finding models in $\mathcal{M}_{\phi,\vec{\gamma}}$ again. In detail, $G = \bigvee G_i$, where $G_i \equiv b_i$ if $\gamma_i = 0$, $G_i \equiv \neg b_i$ if $\gamma_i = 1$.
- Step 3. Find the next model $\vec{\mu}' \in \mathcal{M}(\phi')$ and a partial assignment $\vec{\gamma}'$, s.t., $\vec{\mu}' \in \mathcal{M}_{\phi',\vec{\gamma}'}$ like Step 1. Repeat above steps until $\mathcal{M}(\phi') = \emptyset$, i.e., unsatisfiable.

In this way, we could find a set $\Gamma = \{\vec{\gamma}, \vec{\gamma}', \dots\} \subset \mathcal{M}(PS_\phi)$. The above scheme guarantees

$$\mathcal{M}(\phi) = \bigcup_{\vec{\gamma} \in \Gamma} \mathcal{M}_{\phi,\vec{\gamma}} \text{ and } \mathcal{M}_{\phi,\vec{\gamma}_1} \cap \mathcal{M}_{\phi,\vec{\gamma}_2} = \emptyset, \forall \vec{\gamma}_1, \vec{\gamma}_2 \in \Gamma, \vec{\gamma}_1 \ne \vec{\gamma}_2. \tag{11}$$

From Equation 11, we know that $\mathcal{M}_{\phi,\vec{\gamma}}$s are non-overlapping, then

$$\texttt{integral}(\phi) = \sum_{\vec{\gamma} \in \Gamma} \texttt{integral}(\mathcal{M}_{\phi,\vec{\gamma}}) = \sum_{\vec{\gamma} \in \Gamma} \texttt{integral}(H_{\vec{\gamma}}) \cdot 2^{d_{\vec{\gamma}}}, \tag{12}$$

where $d_{\vec{\gamma}}$ is the number of γ_is which are not assigned, $m + 1 \le i \le l$. From Algorithm 2 and Theorem 9, we could approximate bounds for each $H_{\vec{\gamma}}$, i.e., $\texttt{lb}(H_{\vec{\gamma}}) \le \texttt{integral}(H_{\vec{\gamma}}) \le \texttt{ub}(H_{\vec{\gamma}})$. By summing up, we obtain the total bounds for ϕ:

$$\texttt{lb}(\phi) \le \texttt{integral}(\phi) = \sum_{\vec{\gamma} \in \Gamma} \texttt{integral}(H_{\vec{\gamma}}) \cdot 2^{d_{\vec{\gamma}}} \le \texttt{ub}(\phi), \tag{13}$$

where $\texttt{lb}(\phi) = \sum_{\vec{\gamma} \in \Gamma} \texttt{lb}(H_{\vec{\gamma}}) \cdot 2^{d_{\vec{\gamma}}}$ and $\texttt{ub}(\phi) = \sum_{\vec{\gamma} \in \Gamma} \texttt{ub}(H_{\vec{\gamma}}) \cdot 2^{d_{\vec{\gamma}}}$.

5 Evaluation

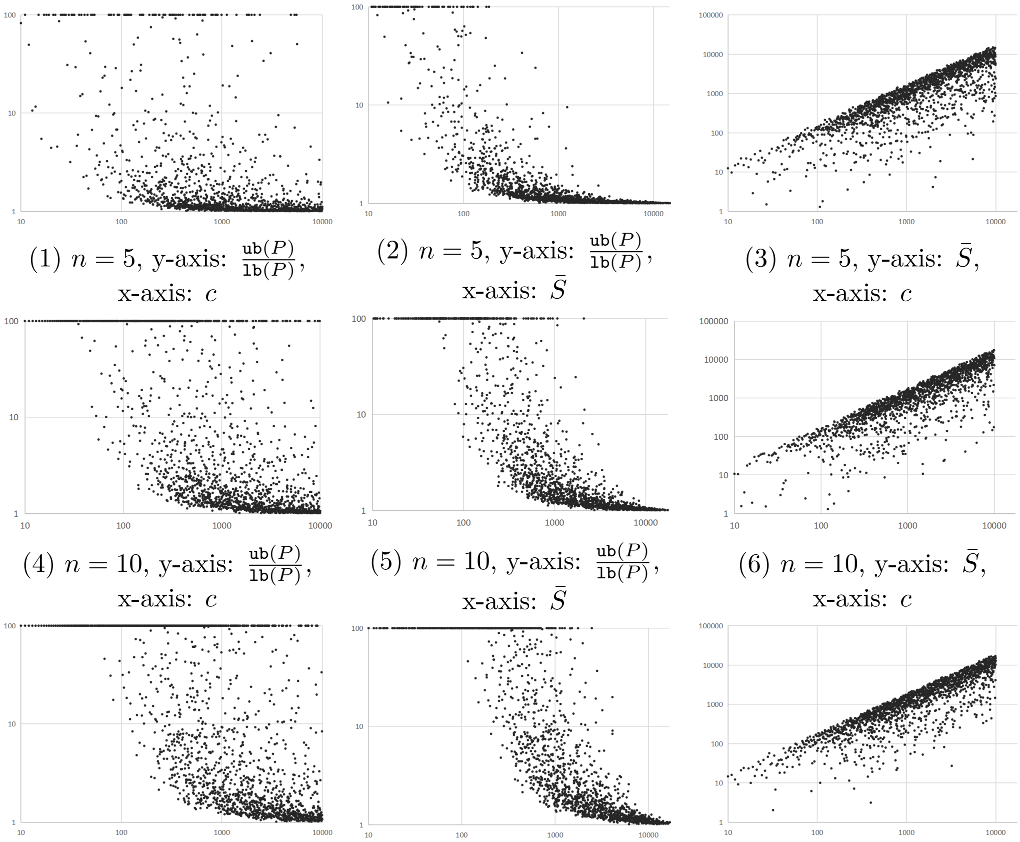
5.1 Experimental settings

Experiments were conducted on a cloud with 48 Core Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz and 128GB memory. We used a timeout of 3600 seconds and a memory limit of 16GB. The suite of benchmarks consists of two families:

- **Random Polytopes** $P(m, n, n_I, c)$, where m, n, n_I and c are the number of LCs, the number of all variables, the number of integer variables and the range size of both x_i and b_i, respectively. In detail, a polytope $P(m, n, n_I, c) = \{A\vec{x} \le \vec{b}\}$ is generated by the following steps: (1) randomly choose n_I different integer variables, (2) randomly select $a_{ij} \in A$ from -10 to 10 and b_i from $-c$ to c, (3) if P is unsatisfiable then repeat above steps. In our benchmarks, we chose $m = n \in [3, 15]$, $n_I \in [1, n]$ and $c \in [10, 10000]$.
- **Instances from program analysis:** We adopted the application benchmarks introduced by [11] which are generated by analyzing 7 programs ("cubature", "gjk", "http-parser", "muFFT", "SimpleXML", "tcas" and "timeout") ranging from $0.4k$ to $7.7k$ lines of source code via a symbolic execution bug-finding tool. There are 3803 SMT(LIA) (linear integer arithmetic) formulas in total.

Table 1 Running times and approximation results over random polytopes with $c = 1000$.

Benchmarks			ExactMI		Vinci		PolyVest		Bounds by MixIntCount			
n	n_I	$\bar{S}$	integral(F)	t (s)	vol(P)	t (s)	$\hat{\text{vol}}(P)$	t (s)	$\frac{\text{ub}(P)}{\text{vol}(P)}$	$\frac{\text{lb}(P)}{\text{vol}(P)}$	$\lvert X\rvert$	t (s)
4	1	948	1.44E+10	4.99	1.43E+10	0.007	1.45E+10	0.072	1.055	0.914	281	0.004
	2	874	6.07E+08	508	6.07E+08	0.006	6.15E+08	0.074	1.172	0.820	1177	0.006
	3	1099	—	—	1.47E+11	0.006	1.52E+11	0.067	1.037	0.919	156	0.003
5	1	369	1.64E+09	1.89	1.64E+09	0.012	1.66E+09	0.257	1.479	0.635	2791	0.011
	2	945	—	—	8.70E+12	0.019	8.59E+12	0.256	1.050	0.935	189	0.005
	3	1086	—	—	6.13E+13	0.018	6.17E+13	0.249	1.062	0.923	291	0.005
6	1	715	5.55E+09	18.3	5.55E+09	0.053	5.48E+09	0.755	2.653	0.281	5243	0.023
	2	1438	—	—	1.34E+17	0.067	1.39E+17	0.780	1.034	0.925	113	0.005
	3	229	—	—	5.91E+09	0.063	5.92E+09	0.751	1.795	0.464	3824	0.019
7	1	1101	2.47E+17	113	2.47E+17	0.659	2.50E+17	1.90	1.125	0.848	810	0.008
	2	980	—	—	2.41E+17	0.655	2.44E+17	2.05	1.136	0.867	855	0.010
	3	727	—	—	6.19E+15	0.658	6.13E+15	2.04	1.171	0.813	1180	0.011
8	1	703	6.57E+14	615	6.57E+14	7.64	6.44E+14	4.56	2.036	0.415	4643	0.035
	2	1251	—	—	6.83E+20	7.91	6.83E+20	4.17	1.099	0.873	628	0.009
	3	556	—	—	5.71E+15	8.24	5.68E+15	4.62	1.695	0.527	3706	0.030
9	1	475	—	—	—	—	2.37E+13	9.05	7.560	0.064	49295	0.390
	2	1250	—	—	—	—	3.82E+22	8.46	1.152	0.836	1016	0.015
	3	138	—	—	—	—	1.47E+10	9.71	34.50	0.000	100000	0.792
12	1	252	—	—	—	—	2.56E+15	57.4	43.24	0.001	100000	1.517
	2	439	—	—	—	—	8.18E+19	54.9	7.063	0.070	45424	0.700
	3	1408	—	—	—	—	1.43E+32	52.3	1.123	0.845	842	0.025
15	1	958	—	—	—	—	5.85E+32	235	2.120	0.408	5102	0.157
	2	1598	—	—	—	—	5.10E+38	228	1.292	0.766	1950	0.076
	3	1587	—	—	—	—	4.20E+36	224	1.469	0.635	2902	0.105



(1) $n = 5$, y-axis: $\frac{\texttt{ub}(P)}{\texttt{lb}(P)}$, x-axis: c

(2) $n = 5$, y-axis: $\frac{\texttt{ub}(P)}{\texttt{lb}(P)}$, x-axis: $\bar{S}$

(3) $n = 5$, y-axis: $\bar{S}$, x-axis: c

(4) $n = 10$, y-axis: $\frac{\texttt{ub}(P)}{\texttt{lb}(P)}$, x-axis: c

(5) $n = 10$, y-axis: $\frac{\texttt{ub}(P)}{\texttt{lb}(P)}$, x-axis: $\bar{S}$

(6) $n = 10$, y-axis: $\bar{S}$, x-axis: c

(7) $n = 15$, y-axis: $\frac{\texttt{ub}(P)}{\texttt{lb}(P)}$, x-axis: c

(8) $n = 15$, y-axis: $\frac{\texttt{ub}(P)}{\texttt{lb}(P)}$, x-axis: $\bar{S}$

(9) $n = 15$, y-axis: $\bar{S}$, x-axis: c

Figure 2 Experimental results about tightness of bounds $\frac{\texttt{ub}(P)}{\texttt{lb}(P)}$ with different c, $\bar{S}$ and n.

5.2 Experimental results

Table 1 presents experimental results on random polytopes with different values of n and n_I but fixed $c = 1000$. Due to page limit, we could only provide partial results here. In Table 1, $\bar{S} = \frac{1}{n}\sum_{i=1}^{n}(max\{x_i|\vec{x} \in P\} - min\{x_i|\vec{x} \in P\})$ is the average range size of variables in real domain, $|X|$ is the number of sample points generated. ExactMI is the implementation of the straightforward method presented in Section 4, which is the baseline. Vinci and PolyVest are tools for computing or approximating $\texttt{vol}(P)$. MixIntCount approximates the bounds of differences of $\texttt{integral}(F)$ and $\texttt{vol}(P)$. So the closer the bounds $\texttt{lb}(P)/\texttt{vol}(P)$ and $\texttt{ub}(P)/\texttt{vol}(P)$ are to 1, the better.

Table 1 shows that $\texttt{vol}(P)$ is usually very close to the exact $\texttt{integral}(F)$. We observe that ExactMI can only handle instances with only one or two integer variables because the number of integer assignments generated by ExactMI grows exponentially with respect to n_I. The scalability of ExactMI is also limited by Vinci, which runs out of memory in a few seconds when $n \geq 9$. PolyVest is more scalable as it is a polynomial time randomized approximation algorithm. Our tool MixIntCount generates approximate bounds $\texttt{lb}(P) \leq \texttt{integral}(F) \leq \texttt{ub}(P)$. The experimental results show that bounds are mostly useful while the overhead of approximating bounds is negligible compared to the cost of volume computation or approximation. There are two exceptions when $n = 9, n_I = 3$ and

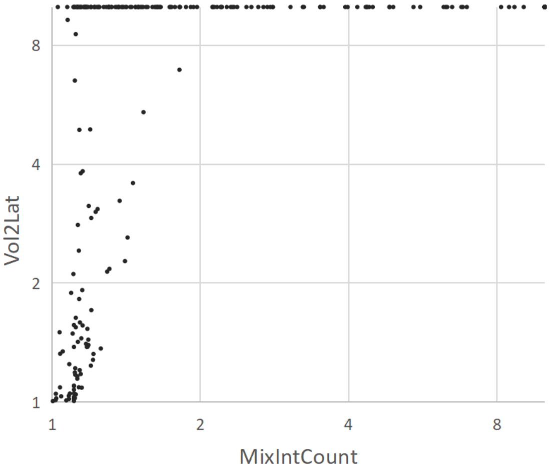


Figure 3 Comparison about the tightness of bounds $\frac{\texttt{ub}(P)}{\texttt{lb}(P)}$ (the smaller the better) between MixIntCount and Vol2Lat on random polytopes with pure-integer variables.

$n = 12, n_I = 1$ in Table 1. In these cases, the algorithm does not stop before reaching the maximum number of sample points, i.e., $|X| = N$, and bounds are also in essence meaningless. Besides, we observe that bounds are sometimes loose while $\texttt{integral}(F)$ and $\texttt{vol}(P)$ are very close, such as, when $n = 6, n_I = 1$ and $n = 8, n_I = 1$. From Table 1, we observe that the tightness of bounds by MixIntCount is related to the size of $\bar{S}$. In addition, the parameter c, which controls the domain size of variables, is fixed to 1000. So we conducted more experiments with different values of c.

Figure 2 presents results about tightness of bounds $\frac{\texttt{ub}(P)}{\texttt{lb}(P)}$ with different settings of c and n. From Figure 2 (3) (6) (9), we observe that $\bar{S}$ is highly correlated with c. Comparing Figure 2 (1) (4) (7) and Figure 2 (2) (5) (8), we find that tightness of bounds is more correlated with $\bar{S}$ than c. Besides, we also find that tightness of bounds is negatively correlated with n by comparing each row.

We compared our tool MixIntCount with Vol2Lat on pure integer instances which can be viewed as the special cases of mixed-integers. Tool Vol2Lat [11] is an approximate integer solution counter via a polytope's volume over pure integer constraints. We conducted experiments on random polytopes where their coefficients are generated in the same way as the mixed-integer cases. The results are presented in Figure 3, whose x-axis and y-axis are the tightness of bounds $\frac{\texttt{ub}(P)}{\texttt{lb}(P)}$ by two tools. Note that we force $\frac{\texttt{ub}(P)}{\texttt{lb}(P)} = 10$ when $\frac{\texttt{ub}(P)}{\texttt{lb}(P)} > 10$, thus the line of points at the top of the figure are cases that $\frac{\texttt{ub}(P)}{\texttt{lb}(P)} > 10$ for Vol2Lat.

Then we conducted experiments on application benchmarks from program analysis which are #SMT(LIA) problems. Note that both MixIntCount and Vol2Lat were incorporated into the DPLL (T)-based #SMT (LA) counter [12]. The results are presented in Table 2. Note that timeout cases by exact counter barvinok (cannot evaluate bounds without exact results) and degenerated cases (volume is zero) are excluded, so there are 3682 instances remaining. In Table 2, $\bar{e}$, $\bar{e}_l$ and $\bar{e}_u$ represent the average values of relative errors $e = \frac{|\texttt{lat}(F) - \texttt{vol}(F)|}{\texttt{lat}(F)}$, $e_l = \frac{\texttt{lb}(F)}{\texttt{lat}(F)}$ and $e_u = \frac{\texttt{ub}(F)}{\texttt{lat}(F)}$. Recall that we chose $\epsilon = 0.1$ for MixIntCount in default, so e_l and e_u will be at most 95% and at least 105%. With a smaller $\epsilon = 0.01$, MixIntCount provides a bit tighter bounds as presented in Table 2. In general, our approach provides much tighter upper bounds than Vol2Lat and consumes more time in exchange.

Table 2 Comparison about errors of bounds (the close to 100% the better) between MixIntCount (MIC) and Vol2Lat on pure integer instances from application instances.

Benchmarks	Vol2Lat			MIC ($\epsilon = 0.1$)			MIC ($\epsilon = 0.01$)		
Name	$\bar{t}$ (s)	$\bar{e}_l$	$\bar{e}_u$	$\bar{t}$ (s)	$\bar{e}_l$	$\bar{e}_u$	$\bar{t}$ (s)	$\bar{e}_l$	$\bar{e}_u$
cubature	1.65	14.9%	347.4%	6.35	13.8%	215.4%	29.9	14.4%	207.6%
gjk	0.17	<0.1%	806.9%	1.65	<0.1%	389.1%	6.49	<0.1%	375.1%
httpparser	0.44	90.5%	117.5%	1.42	87.8%	111.2%	5.14	90.2%	108.6%
muFFT	0.07	45.6%	500.6%	0.71	43.6%	246.8%	1.65	45.4%	239.2%
SimpleXML	0.21	93.8%	120.3%	0.58	90.6%	109.6%	0.95	94.0%	105.9%
tcas	0.60	>99.9%	<100.1%	0.60	95.4%	104.9%	0.59	99.5%	100.5%
timeout	9.45	97.4%	109.3%	14.0	92.8%	108.0%	19.4	97.3%	103.5%

6 Related Works

Ge et al. [11] studied the relation between the count of integer points inside a polytope and a polytope's volume. Then they proposed an algorithm to approximate the upper and lower bounds of integer solution counts via volume. They focused on the pure integer linear constraints. In this paper, we propose new algorithms for bounds approximation via a polytope's volume. The experimental results show that our approach provides tighter bounds on pure-integer cases. In addition, we extend their results and our algorithms to mixed-integer cases using Theorem 9.

The existing works on solving weighted model integration (WMI) problems [3, 17, 25, 26, 28] were focusing on formulas with Boolean and real variables, e.g., in SMT(LRA) (linear real arithmetic) representation. In this paper, we consider the solution space integration problem over mixed-integer linear constraints, which is a theory atom of SMT(LIRA) (linear mixed integer real arithmetic).

Ma et al. [24] first studied #SMT problems and proposed a DPLL(T)-based #SMT framework. Chistikov et al. [7] first extended hashing-based model counting techniques from #SAT to #SMT problems directly. Soon, Chakraborty et al. [6] proposed word-level hash functions for #SMT(BV) (bit-vector) and showed that their approach outperforms Chistikov's method through experiments. After that, Ge et al. [13] compared more recent hashing-based counters with exact integer counters in the scope of linear integer constraints. They observed that exact integer counters combined with the DPLL(T)-based #SMT framework are still more efficient for #LIA and #SMT(LIA) (linear integer arithmetic) problems. So in this paper, we also incorporate our algorithms into DPLL(T)-based #SMT(LA) (linear arithmetic) counters for approximating bounds of solution space integrations of #SMT(LA) formulas.

7 Conclusion

In this paper, we proposed an approximate solution space integration algorithm via a polytope's volume. It is based on theoretical analysis of bounds for solution space integration, volume and integer count. The upper and lower bounds provided by our approach are useful for users to decide whether to trust the approximations. We evaluated our approach's scalability and the tightness of bounds by experiments. Extending the bounds to weighted solution counting problems would be an interesting and challenging direction for future research.

References

1 Alexander I. Barvinok. Computing the volume, counting integral points, and exponential sums. *Discrete & Computational Geometry*, 10:123–141, 1993. doi:10.1007/BF02573970.

2 Alexander I. Barvinok. Computing the ehrhart polynomial of a convex lattice polytope. *Discrete & Computational Geometry*, 12:35–48, 1994. doi:10.1007/BF02574364.

3 V. Belle, A. Passerini, and Guy Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In Qiang Yang and Michael J. Wooldridge, editors, *Proc. of IJCAI 2015*, pages 2770–2776. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/392.

4 H. C. P. Berbee, C. G. E. Boender, A. H. G. Rinnooy Kan, C. L. Scheffer, R. L. Smith, and J. Telgen. Hit-and-run algorithms for the identification of nonredundant linear inequalities. *Math. Program.*, 37(2):184–207, 1987. doi:10.1007/BF02591694.

5 B. Büeler, A. Enge, and K. Fukuda. *Exact Volume Computation for Polytopes: A Practical Study*, pages 131–154. Birkhäuser, 2000. doi:10.1007/978-3-0348-8438-9_6.

6 Supratik Chakraborty, Kuldeep S. Meel, Rakesh Mistry, and Moshe Y. Vardi. Approximate probabilistic inference via word-level counting. In *Proc. of AAAI*, pages 3218–3224, 2016.

7 Dmitry Chistikov, Rayna Dimitrova, and Rupak Majumdar. Approximate counting in SMT and value estimation for probabilistic programs. In *Proc. of TACAS*, pages 320–334, 2015.

8 Ben Cousins and Santosh S. Vempala. Gaussian cooling and o*(n3) algorithms for volume and gaussian volume. *SIAM J. Comput.*, 47(3):1237–1273, 2018. doi:10.1137/15M1054250.

9 M. E. Dyer, A. M. Frieze, and R. Kannan. A random polynomial time algorithm for approximating the volume of convex bodies. In *Proc. of STOC*, pages 375–381, 1989. doi:10.1145/73007.73043.

10 C. Ge and F. Ma. A fast and practical method to estimate volumes of convex polytopes. In Jianxin Wang and Chee-Keng Yap, editors, *Proc. of FAW*, volume 9130 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2015. doi:10.1007/978-3-319-19647-3_6.

11 C. Ge, F. Ma, X. Ma, F. Zhang, P. Huang, and J. Zhang. Approximating integer solution counting via space quantification for linear constraints. In Sarit Kraus, editor, *Proc. of IJCAI*, pages 1697–1703. ijcai.org, 2019. doi:10.24963/ijcai.2019/235.

12 C. Ge, F. Ma, P. Zhang, and J. Zhang. Computing and estimating the volume of the solution space of SMT(LA) constraints. *Theor. Comput. Sci.*, 743:110–129, 2018. doi:10.1016/j.tcs.2016.10.019.

13 Cunjing Ge and Armin Biere. Decomposition strategies to count integer solutions over linear constraints. In Zhi-Hua Zhou, editor, *Proc. of IJCAI*, pages 1389–1395. ijcai.org, 2021. doi:10.24963/ijcai.2021/192.

14 Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *Proc. of ISSTA*, pages 166–176, 2012. doi:10.1145/2338965.2336773.

15 M. Grötschel, L. Lovász, and A. Schrijver. Geometric algorithms and combinatorial optimization. *Combinatorica*, 1988.

16 Amy Huang, Liam Lloyd, Mohamed Omar, and James C. Boerkoel. New perspectives on flexibility in simple temporal planning. In *Proc. of ICAPS*, pages 123–131, 2018. URL: https://aaai.org/ocs/index.php/ICAPS/ICAPS18/paper/view/17775.

17 Samuel Kolb, Pedro Zuidberg Dos Martires, and Luc De Raedt. How to exploit structure while solving weighted model integration problems. In Amir Globerson and Ricardo Silva, editors, *Proc. of UAI*, volume 115 of *Proceedings of Machine Learning Research*, pages 744–754. AUAI Press, 2019. URL: http://proceedings.mlr.press/v115/kolb20a.html.

18 Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes. *J. Symb. Comput.*, 38(4):1273–1302, 2004. doi:10.1016/j.jsc.2003.04.003.

19 L. Lovász and I. Deák. Computational results of an $O^*(n^4)$ volume algorithm. *European Journal of Operational Research*, 216(1):152–161, 2012. doi:10.1016/j.ejor.2011.06.024.

20 L. Lovász and S. Vempala. Hit-and-run from a corner. *SIAM J. Comput.*, 35(4):985–1005, 2006. doi:10.1137/S009753970544727X.

21 L. Lovász and S. Vempala. Simulated annealing in convex bodies and an $O^*(n^4)$ volume algorithm. *J. Comput. Syst. Sci.*, 72(2):392–417, 2006. doi:10.1016/j.jcss.2005.08.004.

22 László Lovász. Hit-and-run mixes fast. *Math. Program.*, 86(3):443–461, 1999. doi:10.1007/s101070050099.

23 Kasper Søe Luckow, Corina S. Pasareanu, Matthew B. Dwyer, Antonio Filieri, and Willem Visser. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *Proc. of ASE*, pages 575–586, 2014. doi:10.1145/2642937.2643011.

24 F. Ma, S. Liu, and J. Zhang. Volume computation for boolean combination of linear arithmetic constraints. In *Proc. of CADE*, pages 453–468, 2009. doi:10.1007/978-3-642-02959-2_33.

25 P. Z. Dos Martires, A. Dries, and Luc De Raedt. Exact and approximate weighted model integration with probability density functions using knowledge compilation. In *Proc. of AAAI, 2019*, pages 7825–7833. AAAI Press, 2019. doi:10.1609/aaai.v33i01.33017825.

26 P. Morettin, A. Passerini, and R. Sebastiani. Advanced SMT techniques for weighted model integration. *Artif. Intell.*, 275:1–27, 2019. doi:10.1016/j.artint.2019.04.003.

27 Gilles Pesant. Counting-based search for constraint optimization problems. In *Proc. of AAAI*, pages 3441–3448, 2016. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12065, doi:10.1609/AAAI.V30I1.10433.

28 Giuseppe Spallitta, Gabriele Masina, Paolo Morettin, Andrea Passerini, and Roberto Sebastiani. Smt-based weighted model integration with structure awareness. In James Cussens and Kun Zhang, editors, *Proc. of UAI*, volume 180 of *Proceedings of Machine Learning Research*, pages 1876–1885. PMLR, 2022. URL: https://proceedings.mlr.press/v180/spallitta22a.html.

29 Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979. doi:10.1137/0208032.

30 Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007. doi:10.1007/s00453-006-1231-0.

31 E. B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927. doi:10.1080/01621459.1927.10502953.

32 Alessandro Zanarini and Gilles Pesant. Solution counting algorithms for constraint-centered search heuristics. In *Proc. of CP*, pages 743–757, 2007. doi:10.1007/978-3-540-74970-7_52.

A CP/LS Heuristic Method for Maxmin and Minmax Location Problems with Distance Constraints

Panteleimon Iosif
University of Western Macedonia, Kozani, Greece

Nikolaos Ploskas
University of Western Macedonia, Kozani, Greece

Kostas Stergiou
University of Western Macedonia, Kozani, Greece

Dimosthenis C. Tsouros
KU Leuven, Belgium

Abstract

In facility location problems we seek to locate a set of facilities in an area, where clients may be present, so that some criterion is optimized. For instance, in the p-center problem we seek to minimize the maximum distance between any client and its closest facility, whereas in the p-dispersion problem we seek to maximize the minimum distance between any two facilities. Hence, in the former we have a minmax objective, whereas in the latter we have a maxmin objective. Recently, a variant of p-dispersion where distance constraints exist between facilities was studied from a CP and ILP perspective. An incomplete CP solver that uses a greedy heuristic to prune branches was shown to significantly outperform Gurobi and OR-Tools in terms of execution time, although it failed to discover optimal or near-optimal solutions in many instances. We enhance this work in two directions, regarding the effectiveness and the applicability of the approach. We first show how local search can be used to obtain better estimations of the bound at each node, resulting in more focused pruning, which allows for optimal or near-optimal solutions to be discovered in many more instances. Then, we demonstrate how the framework can be applied on the p-center problem with distance constraints, comparing it to ILP and CP models implemented in Gurobi and OR-Tools, respectively.

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

Keywords and phrases Constraint Programming, Local Search, facility location, distance constraints, optimization

Digital Object Identifier 10.4230/LIPIcs.CP.2024.14

1 Introduction

Facility location problems are widely studied in OR, AI, computational geometry and other disciplines. In such problems we seek to locate a set of facilities in an area, where clients may be present, so that some criterion is optimized. The optimization criterion largely depends on the type of facilities to be located. When the facilities have beneficial properties (e.g. pharmacies), we want to locate them close to clients. In contrast, when the facilities are (ob)noxious, i.e. they have hazardous effects (e.g. dump sites), we seek to locate them far from clients and/or each other. In between, we have the class of semi-obnoxious facilities that have both desirable and undesirable properties. For instance, we may wish to locate gas stations close to clients for their convenience, but not too close because of the pollution and potential danger they are associated with.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 14; pp. 14:1–14:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The p-center and the p-dispersion problems have been widely studied when modeling beneficial and obnoxious scenarios, respectively [29, 38, 34]. In the former we seek to locate p facilities in an area where clients are present, so that the maximum distance between any client and its closest facility is minimized. In the latter we seek to locate p (ob)noxious facilities, so that the minimum distance between any two facilities is maximized. Hence, in p-center we have a minmax objective, whereas in p-dispersion we have a maxmin objective. The relationship between these two problems has been identified as early as 1977 [45].

In practice, p-center problems occur when the close proximity of facilities to clients is desirable, and in addition, we seek a fair location, in the sense that no client is too far from the facilities. On the other hand, p-dispersion problems occur whenever a close proximity of facilities is dangerous or for other reasons undesirable [34, 20, 37].

Recently, a variant of p-dispersion that includes distance constraints between facilities was studied from a CP and ILP perspective [41]. Distance constraints in location problems were first studied by Moon and Chaudhry [38] and can stem from operational needs and regulations, such as clearance distances for safe chemical storage [1], separation distances between packages containing radioactive materials [49] or portable fire extinguishers [50]. Also, by placing distance constraints between facilities and/or between facilities and clients, we can model the requirements that arise when trying to locate semi-obnoxious facilities [33].

Ploskas et al. [41] described a search method that uses a heuristic to prune branches, based on a estimation of the cost at each node. If the estimated bound, provided by a greedy assignment of the unassigned variables, is not higher than the cost of the incumbent solution then a fail is forced and the sub-tree below the current node is pruned. A solver that uses this method is naturally incomplete, but the solver of [41] was shown to be much faster than standard CP and ILP solvers. However, it was able to discover the optimal solution in only 2 out of the 82 MDPLIB benchmark instances for which the optimal solution is known, and was quite far from the optimal in many cases. This is because the greedy heuristic may often underestimate the cost, resulting in exceedingly high branch pruning, and the omission of the optimal and close-to-optimal solutions.

We extend the work of [41] in two directions, regarding effectiveness and applicability. We first enhance the heuristic used to prune the search space through the use of local search (LS). Specifically, at each node we compute the greedy assignment and use it as the initial solution to the relaxed problem obtained by not considering the distance constraints. Then we try to improve this assignment through the application of a variant of the best pairwise interchange heuristic for p-dispersion (a LS method) [22]. This results in better estimations of the bound, allowing for the discovery of 40 optimal and many near-optimal solutions, and resulting in better solutions being discovered in almost all instances for which the optimal is not known. We also add a second local search component that tries to improve any new solution being discovered by searching in the neighborhood of feasible solutions.

Secondly, we demonstrate that the entire concept of building an incomplete CP solver around a heuristic that estimates the cost at each node and accordingly prunes the search space, is also applicable in other location problems with distance constraints, using the p-center problem as a demonstration. As baseline methods, we introduce ILP and CP models for this problem and implement them in Gurobi and CP-SAT OR-Tools, respectively. We use similar reasoning to the p-dispersion problem to build a CP solver that uses a simple model of the problem and applies a local search method to estimate the bound at each node, and thereby prune branches, making the solver incomplete.

We experimented with p-center problems with distance constraints of two types. In the first one the locations of the clients and the potential facility sites are randomly placed in a grid, while in the second we use p-median benchmark instances [3] as basis. Problems of the first type, that typically have few solutions, can be very hard for Gurobi, as it often does not discover any solution within 1 hour of cpu time, but mostly manageable by OR-Tools, except for the larger ones. In contrast, OR-Tools fares badly on problems of the second type, that typically have many solutions, while Gurobi finds most of these problems quite easy. But both face difficulties in handling very large problems because of the size of their models.

The incomplete CP solver's performance is much more robust. It is outperformed by OR-Tools, in terms of solution quality, in small/medium size grid-structured problems, but is much more efficient than Gurobi. It also discovers solutions in all instances, even the larger ones that are out of reach for the complete solvers. On the other hand, the solver outperforms OR-Tools in p-median based problems, finding many optimal or near-optimal solutions much faster. It can also be quite faster than Gurobi in some cases, especially when the number of facilities is small, but cannot compete in instances where we try to locate a large number of facilities in relatively few facility sites.

In the following we first review related work and define the two problems, focusing on the p-center variant which has not been considered before, in terms of solution methods. Hence, we propose ILP and CP models for this problem. Then we describe the use of local search to boost the effectiveness of a CP solver. Finally, we present experimental results.

2 Related Work

The p-center problem, which is $\mathcal{NP}$-hard in the general case, was originally proposed by Hakimi, along with the related p-median problem [29, 30]. Since then, various ILP formulations for the p-center problem have been proposed [13, 18, 6]. A review of exact and heuristic methods for the p-center problem is given in [6]. The p-dispersion problem, which is also $\mathcal{NP}$-hard on general networks for an arbitrary p [25], was originally mentioned by Shier [45]. However, the term p-dispersion first appeared in the analysis of location problems with distance constraints by Moon and Chaudhry [38]. The first ILP solution was proposed by Kuby [34] while a specialized algorithm and alternative ILP models followed [19, 44, 43].

Regarding distance constraints, Moon and Chaudhry were the first to systematically study them [38]. Both p-dispersion and p-center with distance constraints were mentioned as problems that can arise in real-life scenarios, but no approaches towards solving them were proposed. Recently, Dai et al. revisited the former as part of a study on circle dispersion in non-convex polygons [12]. Distance constraints have also been considered in the context of other location problems [9, 10, 32, 47, 8, 39, 11, 48]. For instance, Tansel et al. studied the distance constrained *p-center* problem for the case where the network is a tree [47], whereas Comley studied the problem of locating a small number of semi-obnoxious facilities that interact with each other as well as with other existing facilities [11]. More recently, Berman and Huang studied the problem of locating obnoxious facilities so as to minimize the total demand covered, so that no two facilities are closer than a pre-specified distance [4]. Drezner et al. studied obnoxious facility location problems with restrictions on the distance between facilities and demand points [16, 17].

There are very few CP-related methods for facility location problems [23, 7, 46, 41]. As mentioned above, we build on the work of [41] which is concerned with the p-dispersion problem. There are CP works outside facility location that are relevant to ours, as they too sacrifice the completeness of a CP solver to solve optimization problems faster [31, 36, 26]. However, these works typically do this through different ways, e.g. by adding extra constraints that may disallow solutions but cut off large portions of the search tree.

The *greedy* heuristic for p-dispersion (resp. p-center) solves the 2-dispersion problem first (resp. the 1-center problem) and then places the remaining facilities one by one, choosing the location point for the currently considered facility f that maximizes the minimum distance in the set of already placed facilities plus f (resp. minimizes the maximum distance between clients and their closest facility among the already located ones plus f) [35, 21]. The *best pairwise interchange* heuristic for p-dispersion starts with a random location of the facilities, then it finds the pair of facilities f_1,f_2 that are closest to each other, and then it finds the free location point v that by allocating either f_1 or f_2 to v, the value of the objective function is maximally improved. This is repeated until no further improvement is possible [22].

3 Problem Definition

We assume that p facilities in a set of facilities F are to be placed on p nodes of a weighted network G. Hence, we deal with discrete/network location problems. Each facility site can host at most one facility. In a *p-center with distance constraints problem* (pCD), a set CL of demand points (clients) to be serviced by the facilities is located at certain (known) nodes of G. In a *p-dispersion with distance constraints problem* (pDD), we are only interested in the dispersion of the facilities. In the following, we will use the terms demand points and clients interchangeably, and we will assume that the set of nodes in G where the clients are located is known, and so is the set of nodes P where the facilities can potentially be located. The weight w_{ij} of an edge denotes the symmetric service cost (typically the distance) between nodes i and j. We assume that once the facilities have been located in a pCD, each client will be serviced by its closest facility, and there are no capacity restrictions on the facilities.

Between each pair of facilities f_i and f_j there is a distance constraint $dis(f_i, f_j) > d_{ij}$ specifying that the distance $dis(f_i, f_j)$ between the points where the facilities f_i and f_j are located must be greater than d_{ij}, where d_{ij} is a constant. In a pCD there is also a distance constraint $dis(f_i, c_k) > dk_i$ between each facility f_i and any client c_k, specifying that the distance $dis(f_i, c_k)$ between the node where facility f_i is located and node k where client c_k is located must be greater than dk_i, where dk_i is a constant.

A common assumption in the relevant literature is that the distance bound d_{ij} is the same for all the constraints between facilities. This is reasonable when the facilities are homogeneous, and therefore in essence indistinguishable, but it is not always realistic, especially when the facilities have different properties, as for example in [1, 49]. In this paper we deal with the heterogeneous case where the distance bound may vary from constraint to constraint. However, we follow the homogeneity assumption with respect to clients, meaning that for a specific facility f_i, we assume that the minimum distance bound between its location and the locations of the clients is the same for all clients.

The distance between two points i and j can be given by the Euclidean distance, e.g. for the location of hazardous facilities, or by the shortest path in a street network, e.g. for the location of franchises, or by any other suitable metric. The methods we propose do not depend on any particular distance measure because, as is common in the literature, we assume that the pairwise distances between all possible client and facility location points are given in a 2-d distance matrix D. In the case of the pCD, we assume that the service cost between a client and the location of a facility, which we try to minimize, is as natural given by the length of the shortest path between the two nodes in the network. We assume that the shortest paths between all pairs of nodes have been precomputed and their lengths are stored in a 2-d matrix SP. However, if necessary, instead of precomputing the distances and storing them in a distance matrix, they could be computed "on the fly", under the condition that this operation takes constant time. This holds for Euclidean or Manhattan distances given the coordinates of the points, but it does not hold for the shortest path in a network. To summarize, in a pDD we have:

- P: the set of candidate facility locations.
- F: the set of facilities to be located.
- p: the number of facilities to be located.
- $D[i,j]$: the distance between any two points.
- d_{ij}: the lower bound in the distance between each pair of facilities (i,j).

Additionally, in a pCD we have:

- CL: the set of demand nodes.
- $SP[i,j]$: the shortest path (s.p.) distance between any two points i and j.
- dk_i: the lower bound in the distance between each facility $i \in F$ and all demand nodes.

The goal in a pCD (resp. pDD) is to locate each facility so that the maximum s.p. distance between any client and its closest facility is minimized (resp. the minimum distance between any two facilities is maximized), subject to the satisfaction of all the distance constraints.

4 ILP Model for the pCD

In this section, we present an ILP formulation for the pCD, based on a standard formulation for p-center [18]. The model is extended to deal with heterogeneous facilities and to include distance constraints. We make use of the following additional notation:

- $C = \{(i,j,f_1,f_2) | i,j \in P, f_1,f_2 \in F, D[i,j] \leq d_{f_1 f_2}\}$: the set of quadruples (i,j,f_1,f_2) s.t. facilities f_1 and f_2 cannot be placed in facility sites i and j, respectively, because i and j are not in a safe distance between each other with respect to the allowed distance between f_1 and f_2.
- $N = \{(i,j) | i \in P, j \in F, \exists k \in CL, D[i,k] \leq dk_j\}$: the set of pairs (i,j) s.t. facility j cannot be placed in facility site i because there exists a demand node k that is not in safe distance from i with respect to the allowed distance between j and the demand nodes.
- $x_{ij} = 1$ if a facility $j \in F$ is located at a facility site $i \in P$ and 0 otherwise.
- $y_{ki} = 1$ if a demand node $k \in CL$ is assigned to a facility site $i \in P$ and 0 otherwise.

As we deal with heterogeneous facilities, we need $|P| \times |F|$ variables, i.e. one variable $x_{ij}, \forall(i,j), i \in P, j \in F$, in order to know whether or not a specific facility $j \in F$ is located at a facility site $i \in P$, because facilities are not indistinguishable as in the standard case of p-center. In addition, we need $|CL| \times |P|$ variables, i.e. $y_{ki}, \forall(k,i), k \in CL, i \in P$, in order to know whether or not a demand node $k \in CL$ is assigned to a facility site $i \in P$. Variables $y_{ki}, \forall(k,i), k \in CL, i \in P$, are required in order to: (i) calculate the distance between each demand node and the facility that serves it (in the objective function of the model), and (ii) place restrictions on which facilities can serve each demand node based on the distance constraints. Variable z is a continuous variable capturing the maximum s.p. distance between clients and located facilities.

The mixed-integer linear programming model for the pCD problem can be expressed as:

$$
\begin{aligned}
\min \quad & z && && (1)\\
\text{s.t.} \quad & \sum_{i\in P} x_{ij} = 1 && \forall j \in F && (2)\\
& \sum_{i\in P}\sum_{j\in F} x_{ij} = p && && (3)\\
& \sum_{i\in P} y_{ki} = 1 && \forall k \in CL && (4)\\
& y_{ki} \leq \sum_{j\in F} x_{ij} && \forall k \in CL, \forall i \in P && (5)\\
& \sum_{i\in P} y_{ki} \times SP[k,i] \leq z && \forall k \in CL && (6)\\
& \sum_{j\in F} x_{ij} \leq 1 && \forall i \in P && (7)\\
& x_{if_1} + x_{jf_2} \leq 1 && \forall\,(i,j,f_1,f_2) \in C && (8)\\
& x_{ij} = 0 && \forall\,(i,j) \in N && (9)\\
& x_{ij} \in \{0,1\} && \forall i \in P, \forall j \in F && (10)\\
& y_{ki} \in \{0,1\} && \forall k \in CL, \forall i \in P && (11)\\
& z \in \mathbb{R} && && (12)
\end{aligned}
$$

The objective function 1 aims at minimizing the maximum s.p. distance between the clients and their nearest located facility. Constraint 2 guarantees that each facility should be hosted at exactly one facility site, while Constraint 3 specifies that p facilities are to be located. Although this constraint is subsumed by Constraint 2, our experiments showed that there are cases where it results in important speed-ups. Hence, we include it in the model. Constraint 4 ensures that each demand node will be served by one facility site, while Constraint 5 guarantees that each demand node will be served by a facility site where a facility has been located. It is a generalization of the Balinski constraint [2] to the case of heterogeneous facilities. Constraint 6 ensures that the variable z will be greater than or equal to all the s.p. distances between demand nodes and their nearest located facility.

Constraint 7 ensures that each facility site can host at most one facility. Constraint 8 models the distance constraints between facilities. It ensures that each facility is at a safe distance from all other facilities by not allowing two facilities f_1 and f_2 to be established at sites that are at a distance closer than the allowed distance between f_1 and f_2. Finally, Constraint 9 is the distance constraint between facilities and demand nodes. This constraint ensures that any located facility is at a safe distance from every demand node.

Two important decisions regarding the efficiency of the model are how to formulate the requirement that each client will be served by an open facility site, and the modeling of the distance constraints. Regarding the former, the requirement can be captured by a generalization of the Efroymson & Ray constraint, using a Big M constant. This results in fewer constraints compared to using the Balinski Constraint 5, but run times are clearly worse, and therefore the Efroymson & Ray was rejected. Regarding the distance constraints, we chose to follow and adapt the best model described in [4], where essentially the distance constraints are modeled through a special case of clique constraints. Computational results from [4], as well as our own preliminary experiments, showed that this model is better than various other models proposed in the literature [38, 40].

5 CP Models for the pCD

The pCD is modeled as a Constraint Optimization Problem (X, Dom, CL, C, O), where X is the set of decision variables, Dom is the set of finite domains, CL is the set of demand nodes, C is the set of hard constraints, and O is the optimization function. The model is as follows:

1. For each facility $i \in F$ there is a finite domain variable x_i.These p variables are the decision variables in the problem, meaning that $|X| = |F| = p$. The domain of each variable $x_i \in X$, denoted by $Dom(x_i)$, includes as values all the points where a facility can be located, i.e. $\forall x_i \in X : Dom(x_i) = P$.
2. Y_1 is a set of auxiliary variables, s.t. for each pair of variables $(x_i, x_j) \in X \times X \mid i < j$, there is a variable $y_{1ij} \in Y_1$ and a constraint $y_{1ij} = D[x_i, x_j]$. Hence, each $y_{1ij} \in Y_1$ models the distance between x_i and x_j. In CP solvers, this is implemented using the Element global constraint, i.e. $y_{1ij} = Element(D, [x_i, x_j])$.
3. Y_2 is a set of auxiliary variables, s.t. for each pair of facilities and clients $(x_i, c_k) \in X \times CL$, there is a variable $y_{2ik} \in Y_2$ and a constraint $y_{2ik} = D[x_i, c_k]$. Hence, each $y_{2ik} \in Y_2$ models the distance between x_i and c_k. In CP solvers, this is again implemented using the Element constraint, i.e. $y_{2ik} = Element(D, [x_i, c_k])$.
4. S is a set of auxiliary variables, s.t. for each pair of variables and clients $(x_i, c_k) \in X \times CL$, there is a variable $s_{ik} \in S$ and a constraint $s_{ik} = SP[x_i, c_k]$. Hence, each $s_{ik} \in S$ models the service cost (i.e. s.p. distance) between x_i and c_k. Again, this is implemented using the Element constraint, i.e. $s_{ik} = Element(SP, [x_i, c_k])$.
5. Z is a set of auxiliary variables, s.t. for each client $c_k \in CL$, there is a variable $z_k \in Z$ and a constraint $z_k = min(s_{1k}, s_{2k}, ..., s_{pk})$. Hence, each $z_k \in Z$ models the shortest path distance between each client and its nearest facility.
6. For each variable $y_{1ij} \in Y_1$, there is a distance constraint $y_{1ij} > d_{ij}$.
7. For each variable $y_{2ik} \in Y_2$, there is a distance constraint $y_{2ik} > dk_i$.
8. There is a variable z, s.t. $z = \max(Z)$, capturing the maximum shortest path distance between a client and its closest facility.
9. The objective function is $O = minimize(z)$.

The $s_{ik} = SP[x_i, c_k]$ constraints link the auxiliary variables s_{ik}, and therefore also the z variable and the objective function, with the decision variables. We also considered adding an AllDifferent constraint over all variables in X. Such a constraint is redundant, as the distance constraints already force the variables to take different values, and experiments with and without it showed no noticeable difference. Also, to reduce the number of auxiliary variables, the distance constraints between facilities can be captured using the Table constraint instead of the Element constraint. Our experiments showed that this does not make a difference in very large problems, where, as we will demonstrate, OR-Tools runs out of memory, but its effect on smaller problems remains to be experimentally investigated in detail.

A reason for the failure of OR-Tools to solve large instances of the pCD (and the pDD) is the size of the model it constructs, largely because of the very large domains (which are not uncommon in location problems), along with the auxiliary variables. This is to an extent due to the lazy clause generation mechanism of the solver, which creates a large amount of literals to represent all the variable-value combinations. To bypass this, we propose to use a much simpler model, dropping all the auxiliary variables and relevant constraints, resulting in a model with only the p decision variables and the distance constraints. The optimization function can now be handled procedurally within the solver by simply computing the cost of any new solution found so as to determine if this cost is better than the cost of the incumbent solution. If so, then the bound is tightened.

Dropping the auxiliary variables may result in a lighter model, but on the other hand we lose propagation power. If there is no link between the decision variables and the objective function, whether it is explicitly represented, as in the CP model above (items 8,9), or procedurally handled, then any improvement in the cost of the incumbent solution will not be propagated to the decision variables. To partially overcome this problem while keeping a simple model, we apply the following inference techniques:

- Assume that a solution A with better cost than all previously found ones has been located and let C_A denote its cost. We find the clients that determine the value of C_A (i.e. clients with distance C_A from their closest facility). For each one of them, we check if there exists a candidate facility point at a smaller distance than C_A. If there is no such point for some client, then there is no possible way for a facility to be located at a facility point that is closer to this specific client than the value of C_A. Hence, the value of the objective function cannot be further improved, and therefore we terminate search.
- After a variable has been assigned and the distance constraints have been propagated, we check for each client if there exists a value (i.e. available facility point) in any domain of a variable that is located at a distance smaller than the value of C_A, from this specific client. If no such value exists in any domain, then we know that in any feasible solution existing in the sub-tree below the current assignment, at least one client will be assigned to a facility located at an equal or greater distance than C_A. Hence, the cost cannot be improved, and therefore we prune the current branch and continue searching.

Both these techniques are subsumed by propagation in a CP solver that uses the full model of the problem described above. But in the simple model they (partially) compensate for the absence of a link between the optimization function and the decision variables.

6 Enhancing a Heuristic CP-Based Method Through LS

Ploskas et al. proposed a heuristic technique for the pDD that tries to prune early the parts of the search tree for which it seems unlikely that their exploration will improve the value of the optimization function. Specifically, the cost of the first feasible solution found is used as the initial lower bound. Thereafter, at each node, after the currently tried assignment $x_i = a$ has been propagated, an upper bound for the best possible solution is computed, giving an estimation of the best possible cost that can be achieved if the sub-tree rooted at the specific node is explored. If this is not higher than the current lower bound then the current branch of the search tree is abandoned and the search moves on. Each time a solution with a higher cost than the current lower bound is found, the lower bound is updated.

At each node, the bound is computed by applying the greedy heuristic for p-dispersion on the relaxed problem obtained by not considering the distance constraints. Assuming that x_i is the current variable, $x_1 \leftarrow v_1, \ldots, x_{i-1} \leftarrow v_{i-1}$ is the assignment to past variables and v_i is the value under consideration for x_i, the heuristic greedily computes the cost of the "best" assignment for the future variables $x_{i+1}, \ldots, x_p$ [41]. That is, it visits these variables one by one, starting with x_{i+1}, and for each variable x_j, $i + 1 \leq j \leq p$, and each value $v_j \in Dom(x_j)$, it finds the minimum distance between v_j and any assignment (location) among variables (facilities) $x_1, \ldots, x_{j-1}$. The value that maximizes this distance is then (temporarily) assigned to x_j. This is repeated until all variables have been assigned.

This heuristic pruning method was embedded in a custom CP solver (thus making the solver incomplete), which was compared to the complete ILP and CP-SAT solvers Gurobi and OR-Tools. Results demonstrated that it can discover much better solutions in large instances that are very hard for the complete solvers. However, the custom solver was able to discover the optimal solution in only 2 out of the 82 MDPLIB instances of smaller size that are solvable to optimality by the complete solvers. Also, in many cases the cost of the best solution it discovered was quite far from the optimal.

The reason for this is that the greedy heuristic often underestimates the true cost that can be obtained. This leads to faster exploration of the search space, as many branches are pruned, but it also very often results in the omission of optimal/near-optimal solutions. We now describe one way to improve the effectiveness of a solver that follows this reasoning using local search to obtain more accurate bounds when estimating the cost at each node.

6.1 pDD

We propose to invoke LS before the decision is taken to cut off the branch, when the greedy heuristic dictates to do so. We use the assignment obtained by the greedy heuristic as the initial solution to the relaxed problem that omits the distance constraints, and then try to improve it with local moves until either it cannot be improved any more (a local maximum) or its cost becomes higher than that of the incumbent solution. In the former case we cut off the current branch, while in the latter we accept it and continue its exploration.

We use a variant of the best pairwise interchange heuristic for p-dispersion. Our method takes as input the complete assignment $A =< x_1 \leftarrow v_1, \ldots, x_i \leftarrow v_i, x_{i+1} \leftarrow v_{i+1}, \ldots, x_p \leftarrow v_p >$, where $x_{i+1} \leftarrow v_{i+1}, \ldots, x_p \leftarrow v_p$ is the temporary assignment to future variables, the cost C_A of this assignment and the cost of the incumbent solution C_I. It then finds the pairs of facilities (x_l, x_k) that are closest to each other, and therefore have distance equal to C_A (there can be many such pairs, which we call "culprit"), it forms the set of variables X_c involved in such pairs, and for each variable in X_c, it tries to find an alternative location so as to improve the value of C_A as much as possible. Note that for any "culprit" pair, both variables will necessarily be among $x_{i+1}, \ldots, x_p$ (the future ones), and therefore we can freely change their values in the assignment. This stems from the propagation of the updated bound any time a solution that improves the current bound is discovered. CP solvers do this by adding a constraint forcing subsequent solutions to be better than the current one, while the method of [41], that we follow, does it in a slightly different way by integrating the bound's propagation in the solver's arc consistency propagation mechanism.

If at some point, C_A becomes higher than C_I then the method terminates, signaling that the current branch must not be cut off. If this does not happen then the method stops when a local maximum is reached, e.g. when there is no alternative location for any variable in X_c that can increase the value of C_A. Algorithm 1 depicts this process, which, as our experiments showed, is more effective than the best pairwise interchange heuristic which it modifies, especially when a variable is involved in more than one "culprit" pair.

$A[x_j]$ denotes the value that variable x_j takes in the assignment A. Function *Compute_Cost* takes a complete assignment of the variables and returns its cost. The identification of the pairs of facilities that are closest to each other takes place each time a local move is made because any improvement to the cost means that other pairs are now the "culprit" ones.

Naturally, more sophisticated LS methods or meta-heuristics might be able to obtain better bounds. Indeed, we tried several options, including a GRASP meta-heuristic. The LS method of Algorithm 1 was chosen because it achieves a good balance in terms of quality and cpu time. For instance, the GRASP procedure rarely offered overall quality improvements to compensate for the higher run times (and implementation complexity).

Finally, we also use LS to tighten the bound obtained each time a solution that improves the incumbent is discovered. Specifically, once a new incumbent solution $A =< x_1 \leftarrow v_1, \ldots, x_p \leftarrow v_p >$ with cost C_I is discovered, we evaluate the alternative assignments of all

Algorithm 1 $LS(X, Dom, C, A, C_A, C_I)$ for the pDD.

```
repeat until no change in value of $C_A$
  $X_c \leftarrow \{x_l, x_k \in X | D[A[x_l], A[x_k]] = C_A\}$;
  $C_{temp} \leftarrow C_A$;
  for each $x_j \in X_c$
     for each $v_j \in Dom(x_j) | v_j \neq A[x_j]$
        $A[x_j] \leftarrow v_j$;
        $temp\text{-}cost \leftarrow Compute\_Cost(X, Dom, C, A)$;
        if $temp\text{-}cost > C_{temp}$
           $C_{temp} \leftarrow temp\text{-}cost$;
           mark $A[x_j] = v_j$ as best move;
        restore $A[x_j]$;
  if $C_{temp} > C_A$
     $C_A \leftarrow C_{temp}$;
     if $C_A > C_I$ return true;
     make best move;
return false;
```

variables involved in “culprit” pairs. If an assignment that improves C_I (say $x_i \leftarrow v_k$) is found, and by changing the value of x_i to v_k in A, we get a feasible solution (i.e. the distance constraints are satisfied) then C_I is tightened and the process is repeated until no local move that results in a feasible solution can improve C_I. Thereafter, the solver continues search as usual, but with a (hopefully) tightened bound. Using LS to improve the solutions found by a CP solver in optimization problems is a standard way of integrating CP and LS [24].

6.2 pCD

The heuristic we have developed for the pCD works in a similar way. The cost of the first feasible solution found is now used as the initial upper bound, as we have a minimization problem. The initial estimation at each node is again performed by solving the relaxed problem without the distance constraints in a greedy fashion. That is, for each variable x_j, $i + 1 \leq j \leq p$, and each value $v_j \in Dom(x_j)$, we compute the maximum distance between any client and its closest facility among $x_1, \ldots, x_j$ that we would get if v_j was assigned to x_j. The value that minimizes this distance is then temporarily assigned to x_j. After all future variables have been assigned in this way, the resulting assignment A is given to a local search method that tries to improve it. The only difference between this method and the one of Algorithm 1, is that in the case of the pCD there are no “culprit” pairs of variables. Hence, the second line in Algorithm 1 is omitted and all variables are considered when looking for the best local move. Let us explain this.

In a pCD the cost C_A of any complete assignment A is due to the one or more clients that are at distance C_A from their closest facility. But any relocation of a facility x_i may potentially result in an improvement of the cost (a decrease in the value of C_A), as x_i may move closer to these clients, meaning that they could perhaps now be serviced by x_i. In contrast, in the pDD, if none of the variables in a culprit pair is relocated, the cost will never improve (but could worsen), regardless of the relocation of other facilities.

7 Experiments

We experimented with instances generated in two different ways. The first uses a benchmark library as basis to create pCDs or pDDs. For the former, we use the p-median benchmark dataset [3], while for the latter we use the p-dispersion benchmark library MDPLIB 2.0 [37], as in [41]. In the second generation method we seek to locate a number of facilities in a grid.

Computations were performed on an Intel i7 CPU 8700 with 16 GB of main memory, a clock of 3.2 GHz, an L1 cache of 348 KB, an L2 cache of 2 MB, and an L3 cache of 12 MB, running under CentOS 8.4. We set a time limit of 3,600 seconds for all the experiments reported below. The ILP model was solved using Gurobi 9.0.3 [28]. The CP model was written in the CPMpy modeling tool [27] and compiled into CP-SAT OR-Tools [15]. The heuristic CP approach, for both pDDs and pCDs, was implemented in a custom solver written in C. This solver, which is basically a MAC search algorithm [42], implements the simple CP model of Section 5, uses dom/wdeg for variable ordering [5], lexicographic value ordering, arc consistency for the propagation of distance constraints and implements the two inference techniques of Section 5. We also ran experiments with an 8-thread version of OR-Tools, which uses techniques such as large neighborhood search in parallel with the main search, as well as the CP solver Choco [14]. The ILP model is stored in compressed sparse column format, as the constraint matrix can sometimes be too large to be stored as a full array.

7.1 Problem Generation Models

The MDPLIB collects a large number of p-dispersion instances divided into various classes [37]. In the GKD, MDG, and SOM classes, the distances between the potential facility locations are given by Euclidean distances, random real numbers, and random integers, respectively. The instances we tried have 100-1000 potential facility points and 10-30 facilities.

We generated pCDs in two different ways. The *grid generation model* creates problems embedded in a $n \times n$ grid. It takes the following parameters: n, p, $|CL|$, $|P|$. We first randomly select $|CL| + |P|$ among the $n \times n$ nodes. $|CL|$ of these nodes are randomly selected to place the clients and the remaining nodes are the potential facility locations. We assume that the weight of each edge in the grid is equal to 1. Therefore, given that we have a grid, the distance of the shortest path between any two points can be at best equal to the Manhattan distance between them. For each distance constraint $dis(x_i, x_j) > d_{ij}$ between facilities x_i and x_j, d_{ij} is randomly set to an integer number in the interval $[0, max\_euc/2]$, where max_euc is the maximum Euclidean distance between two points on the grid. Accordingly, for the constraints specifying the distances between facilities and clients, a random integer is set in the (experimentally selected) interval $[0, 3]$, in order to minimize infeasibilities.

The *p-median based generator* takes instances from the p-median benchmark dataset [3], consisting of problems with 100–900 nodes and 5 to 200 facilities. We randomly select $|P|$ nodes to be candidate facilities, while the remaining nodes are clients. We have considered two cases: **1)** 80% of the nodes are candidate facility sites and the remaining 20% are clients. If the resulting number of candidate sites is less than or equal to p, then we progressively increase the number of candidate sites until $|P| > p$ and the generated instances are feasible. **2)** 20% of the nodes are candidate facility sites and the remaining 80% are clients. Similarly to the previous case, we progressively increase the number of candidate sites until $|P| > p$ and the generated instances are feasible. To set the parameter d_{ij}, we find the minimum and maximum distance between all pairs of candidate sites and we set d_{ij} equal to a random number in the range $[\min, \min + (\max - \min)/10]$. Similarly for parameter dk_i.

For each generation model and each setting of the parameters, 10 instances were generated.

7.2 Experiments with the pDD

In Table 1 we compare the LS-enhanced CP-based method, denoted CP$_{LS}$, to that of [41], denoted CP$_{G}$, using the same instances (https://github.com/ploskasnikos/pdispersion), and adding some larger ones that are very hard for the complete solvers. For each class we give the number of the MDPLIB instance on which it is based, and in brackets the numbers of potential location sites and facilities. For each solver configuration we report the total cpu time taken over the 10 instances ($\sum$cpu columns), the number of times when the optimal solution was found (#opt columns), and the mean value of the best solution found within the time limit. In the #opt column for CP$_{G}$ we give in brackets the number of instances for which the optimal is known. We also give the number of instances in each class where CP$_{LS}$ found a better solution compared to CP$_{G}$ (#imp columns). In brackets we give the number of instances where CP$_{LS}$ found a worse solution. As baseline, in the last column, for each class we give the mean value of the best solution found by the complete solver that displayed the best performance in the particular class. This is left blank (-) if no complete solver was able to find at least one solution in all instances of a class.

Table 1 Comparing CP$_{LS}$ to CP$_{G}$ on MDPLIB-generated pDDs.

Class	CP$_{G}$			CP$_{LS}$				Baseline
(p,$\|P\|$)	$\sum$cpu	#opt	cost	$\sum$cpu	#opt	#imp	cost	cost
				MDG				
a1 (100,10)	1	0 (10)	4.35	1	10	10 (0)	4.68	4.68
a1 (100,20)	94	0 (0)	1.69	170	0	8 (2)	1.79	1.17
a1 (500,10)	51	0 (0)	5.88	56	0	10 (0)	6.11	6.02
a1 (500,20)	132	0 (0)	2.91	188	0	9 (1)	2.96	1.57
a2 (100,10)	1	0 (10)	4.24	1	10	10 (0)	4.74	4.74
a2 (100,20)	156	0 (0)	1.64	166	0	7 (2)	1.71	1.4
a2 (500,10)	54	0(0)	5.94	66	0	10 (0)	6.22	5.92
b1 (100,10)	1	0 (10)	428.18	1	0	10 (0)	455.94	460.11
b1 (100,20)	63	0 (0)	181.2	133	0	8 (1)	200.49	109.35
b1 (500,10)	45	0 (0)	576.45	53	0	10 (0)	591.56	584.33
b1 (500,20)	107	0 (0)	290.64	128	0	9 (0)	302.75	-
b2 (100,10)	1	0 (9)	428.17	1	9	10 (0)	459.99	459.99
b2 (100,20)	109	0 (0)	159.93	140	0	8 (2)	170.40	113.33
b2 (500,10)	51	0 (0)	574.55	55	0	10 (0)	608.61	581.31
				GKD				
d1 (100,10)	2	1 (10)	33.42	2	0	9 (1)	33.85	34.06
d1 (250,10)	11	0 (7)	34.25	21	3	10 (0)	36.06	35.98
d1 (250,20)	>19,837	0 (0)	18.83	>24,663	0	8 (1)	19.24	10.55
d1 (500,10)	108	0 (0)	36.01	99	0	10 (0)	38.21	36.96
d1 (1000, 20)	>36,000	0 (0)	16.99	>36,000	0	6 (1)	17.95	-
d2 (100,10)	2	1 (10)	31.29	3	8	10 (0)	34.22	34.82
d2 (250,10)	10	0 (5)	35.06	22	0	10 (0)	37.1	36.31
d2 (250,20)	>24,865	0 (0)	14.93	>24,998	0	5 (0)	15.39	-
d2 (1000, 10)	846	0 (0)	36.47	1,830	0	10 (0)	39.25	32.72
				SOM-GRID				
b5 (200, 20)	11	0 (0)	2	13	0	0 (0)	2	2
g7 (80, 30)	36	10 (10)	2	51	10	0 (0)	2	1.9

We can make the following observations. CP$_{LS}$ finds better solutions than CP$_{G}$ in the vast majority of instances, discovering the optimal solution in 40 MDPLIB instances, compared to only 2 for CP$_{G}$. This gives an improvement in the mean cost in all classes of the MDG and

GKD categories, often by large margins. As a result, CP$_{LS}$ finds optimal solutions or near-optimal ones in all of the smaller classes that are within reach of the complete solvers, while improving the bounds in larger classes for which the optimal is unknown (e.g. in b1(100,20) where the value is now almost twice that obtained by the complete solvers). Importantly, the cpu time overheads of CP$_{LS}$ compared to CP$_G$ are not very significant, keeping in mind that the $\sum$cpu columns give total run times over 10 instances until termination or cut-off and that both solver configurations are orders of magnitude faster than the complete solvers [41]. CP$_{LS}$ usually reaches the best solution found by CP$_G$ in similar or faster run times, but often takes longer to terminate because it further improves this solution.

For CP$_{LS}$, we counted the number of times that the greedy heuristic made an estimation lower than the current bound, meaning that LS was then called, and the number of times that LS managed to increase the estimation to a value greater than the current bound, meaning that the current branch was not cut off. It turns out that in around 4% to 14% of the calls to LS, on average, the current branch was not cut off (details in the Appendix). Despite these relatively low percentages, CP$_{LS}$ obtains significant improvements in solution quality.

Results from the SOM MDPLIB class and from grid-structured problems are also given in [41]. These categories were both very easy in terms of run times for CP$_G$ and CP$_{LS}$. In these cases, CP$_{LS}$ found the same solutions as CP$_G$ because either CP$_G$ already discovered the (known) optimal solutions in the smaller classes, or the solutions discovered had very little room for improvement (objective values in the range $1 \ldots 5$). Hence, we only give indicative results from the hardest class from each category. In the b5 SOM class, CP$_{LS}$ and CP$_G$ display similar performance and probably both discover the optimal in all instances (Gurobi also finds solutions with cost 2, but was unable to prove optimality within the time limit). In the g7 grid class, both CP$_{LS}$ and CP$_G$ discover the known optimal solutions in all instances.

Finally, in some rare cases CP$_G$ can locate slightly better solutions than CP$_{LS}$. To put it simply, this is because both methods are heuristic. In more detail, a possible scenario is the following: Suppose that the first solution A located has objective value C$_A$. As the search continues, assume that LS improves the heuristic's estimation at some node and does not cut the current branch (as CP$_G$ would do), leading to a better solution B being later located, with value C$_B$ < C$_A$. The discovery of solution B may lead to more branches being then cut off, potentially including a branch that leads to a solution C with value C$_C$ < C$_B$. Hence, CP$_{LS}$ will not discover this solution. On the other hand, it is possible that CP$_G$ locates solution C, because using the upper bound of C$_A$ > C$_B$ to prune, may lead to weaker pruning, allowing for the branch that leads to solution C to be explored.

7.3 Experiments with the pCD

Table 2 details the classes generated for pCDs using the two generation models. For the p-median based ones, we give the name of the p-median benchmark used as basis. Each such class is defined by the parameters $< |V|, p, |P|, |CL| >$. For example, class <400,5,80,320> includes problems with 400 points, 5 facilities, 80 potential locations and 320 clients. The 2nd column gives classes where the number of clients is larger or equal to the number of candidate sites, while the 3rd gives classes where there are more candidate sites than clients. Each grid based class is defined by the parameters $< n, p, |P|, |CL| >$. For example, in class <10,20,80,20> we have a 10×10 grid, 20 facilities, 80 potential locations and 20 clients.

We have experimented with five solvers: Gurobi, OR-Tools, 8-threads OR-Tools, Choco, our custom CP solver. These are all complete solvers. We also experimented with three configurations of our solver that incorporate the pruning heuristic, making the solver incom-

Table 2 Problem classes and their characteristics.

p-median based	$\lvert CL\rvert \geq \lvert FP\rvert$	$\lvert FP\rvert > \lvert CL\rvert$	**grid based**	
pmed5	<100,33,40,60>	<100,33,80,20>	g1	<10,20,80,20>
pmed10	<200,67,100,100>	<200,67,160,40>	g2	<20,10,350,50>
pmed15	<300,100,150,150>	<300,100,240,60>	g3	<20,20,350,50>
pmed21	<500,5,100,400>	<500,5,400,100>	g4	<20,25,350,50>
pmed26	<600,5,120,480>	<600,5,480,120>	g5	<30,20,300,50>
pmed31	<700,5,140,560>	<700,5,560,140>	g6	<30,10,500,100>
pmed36	<800,10,400,400>	<800,10,640,160>	g7	<30,20,500,100>
pmed38	<900,5,450,450>	<900,5,720,180>	g8	<30,20,700,200>
			g9	<50,100,1300,200>

plete. The first (CP$_G$), only uses a greedy heuristic to estimate the cost at each node (as in [41] for the pDD), the second one uses LS to perform the estimation, and the third one (CP$_{LS}$), adds the LS component that tries to improve any solution found. Choco displayed inferior performance compared to OR-Tools and therefore was not included in extensive experiments, while the 8-thread version of OR-Tools did not demonstrate any significant benefits compared to the single thread one. The basic complete version of our solver (i.e. without the heuristic) was not competitive, despite using a simple model of the problem. Indicative results of this solver and 8-threads OR-Tools are given in the Appendix.

Among the configurations of our solver that use the bound estimation heuristic, CP$_{LS}$ displayed the best results. CP$_{LS}$ finds the optimal in 85 out of the 187 instances for which the optimal is known, as opposed to 68 for CP$_G$, and improves the cost found by CP$_G$ (resp. worsens it) in 47 (resp. 4). Regarding run times, as in the pDD, CP$_{LS}$ usually reaches the best solution found by CP$_G$ in similar or faster run times, but may take longer to terminate because it further improves this solution. It achieves similar run times in classes where CP$_{LS}$ rarely improves the cost, while it can take up to twice the time to terminate in classes where it often improves the cost. The second LS component makes a slight contribution towards the solver's performance. When turned off (i.e. LS only used for branch pruning) then there is an improvement in 43 instances (82 optimal) compared to CP$_G$.

Interestingly, in only around 0.3% to 3% of the calls to LS, the current branch was not cut off. Despite these very low percentages, CP$_{LS}$ still managed to find better solutions than CP$_G$ in many instances. It is not surprising that the corresponding percentages are higher in pDDs, as intuitively it is more likely to improve the bound's estimation by changing the location of a facility through a local move in a pDD rather than in a pCD, where a local move must place a facility f_i close to all the clients that are at maximum distance from their closest facility, so that they are now served by f_i.

In Table 3, we compare CP$_{LS}$ to Gurobi and OR-Tools. We report the total cpu times, the mean cost of the best solution found, and the number of instances where any solver was cut off (in brackets after $\sum$cpu). If a solver did not terminate, we count $3,600$ secs towards its cpu time sum and record the best solution it was able to find. Column t$_b$ gives the mean cpu time taken by CP$_{LS}$ to find its best solution. Columns t$_m$ give the mean cpu time taken by Gurobi and OR-Tools to find the first solution that matches (or improves) the cost of the best solution found by CP$_{LS}$. If CP$_{LS}$ was unable to find any solution within the time limit in some instances then t$_b$ is left blank (-). Accordingly, if Gurobi or OR-Tools did not manage to match the solution found by CP$_{LS}$ in some instances, t$_m$ is left blank. The last column gives the number of instances where CP$_{LS}$ found the optimal solution. In some classes, OR-Tools (and Gurobi in one class) suffered memory exhaustion and crashed in all instances without discovering any solutions. This is denoted with MEM in the $\sum$cpu column.

Table 3 Comparing solvers on pCD problems.

	Gur $\sum$cpu	t$_m$	cost	ORt $\sum$cpu	t$_m$	cost	CP$_{LS}$ $\sum$cpu	t$_b$	cost	#opt
$\lvert CL\rvert \geq \lvert FP\rvert$										
pmed5	2 (0)	0	100.5	1,668 (0)	164	100.5	>7,306 (2)	-	-	5
pmed10	44 (0)	4	63.4	1,963 (0)	196	63.4	4 (0)	0.2	63.4	10
pmed15	123 (0)	12	49.7	19,753 (0)	1,974	49.7	384 (0)	1.3	50.1	9
pmed21	201 (0)	11	45.8	212 (0)	18	45.8	6 (0)	0.3	46.5	6
pmed26	206 (0)	13	43.2	224 (0)	16	43.2	9 (0)	0.3	44	4
pmed31	985 (0)	67	34.7	254 (0)	19	34.7	17 (0)	0.7	35.6	3
pmed36	1,160 (0)	19	35.9	>18,468 (3)	951	35.9	478 (0)	19	37	4
pmed38	146 (0)	14	37.2	3,522 (0)	352	37.2	52 (0)	3	37.3	9
$\lvert FP\rvert > \lvert CL\rvert$										
pmed5	5 (0)	0	47	249(0)	25	47	407 (0)	40	47	10
pmed10	64 (0)	6	25.3	10,775 (0)	1,076	25.3	343 (0)	2	26.3	7
pmed15	219 (0)	21	21.6	MEM	-	-	7,544 (0)	36	22.3	7
pmed21	252 (0)	9	27.6	>19,747 (1)	225	27.6	101 (0)	5	28.6	2
pmed26	537 (0)	27	24.7	14,901 (0)	460	24.7	113 (0)	5	25.6	2
pmed31	2,535 (0)	48	21.8	>33,041 (5)	912	21.8	227 (0)	8	22.8	0
pmed36	>5,817 (1)	110	19	MEM	-	-	1,697 (0)	110	20.7	0
pmed38	4,345 (0)	69	20.2	>33,218 (6)	2,454	20.2	184 (0)	8	20.9	3
GRID										
g1	4,103 (0)	381	2.1	443 (0)	14	2.1	2 (0)	0.1	2.8	3
g2	5,191 (0)	73	4.1	7,737 (0)	16	4.1	26 (0)	0.1	6.1	0
g3	>36,000 (10)	-	-	>36,000 (10)	159	4.2	205 (0)	14	5.3	0
g4	>36,000 (10)	-	-	>33,302 (9)	-	-	9,300 (0)	922	5.4	1
g5	>36,000 (10)	-	-	>36,000 (10)	818	6.9	3,317 (0)	320	8.4	0
g6	>23,021 (4)	852	7.6	>34,641 (9)	30	7.4	99 (0)	0.5	9.9	0
g7	>36,000 (10)	-	-	>36,000 (10)	-	-	1,264 (0)	100	8.4	0
g8	>36,000 (10)	-	-	>36,000 (10)	-	-	275 (0)	10	9	0
g9	MEM	-	-	MEM	-	-	>36,000 (10)	814	13.7	0

Gurobi outperforms OR-Tools, in run times, in all classes of the p-median based problems, and in fact finds these problems quite easy, terminating within the time limit in all but one instance. CP$_{LS}$ is able to find many optimal or near-optimal solutions, and it does this very fast in most classes. As the t_b and t_m columns indicate, it can take OR-Tools orders of magnitude longer runs to match the solutions found by CP$_{LS}$ (e.g. pmed15 and pmed38). Compared to Gurobi, CP$_{LS}$ is generally able to find good solutions faster, especially in classes with few facilities (e.g. pmed31), but cannot compete in classes that include many facilities and relatively few facility points. In terms of optimal solutions found, CP$_{LS}$ is quite successful in classes with more clients than facility points, as it finds at least 5 optimal solutions in 5 classes, while it locates all of them in the pmed10 class. The performance is not as good in the $|FP| > |CL|$ category, as there are classes where no optimal solution is found. This is not surprising, considering that in such classes, domain sizes are larger.

Regarding grid based problems, the results differ significantly. OR-Tools now performs better than Gurobi, as the latter finds 6 out of the 9 classes very hard and did not manage to find any solution in any instance of these classes. OR-Tools also finds some of these classes very hard, but managed to discover solutions in at least one instance from every class, but not in all instances (hence the blank t_m and cost columns). CP$_{LS}$ is much more robust, being able to find solutions in all instances of all classes, and doing this very fast in some classes (including hard ones, such as g8).

Regarding solution quality, in classes where one or both of the complete solvers solved all instances, the cost of the solutions discovered by CP$_{LS}$ is worse than that discovered by OR-Tools (and in some cases Gurobi). Also, CP$_{LS}$ managed to find only 4 of the known optimal solutions. On the other hand, CP$_{LS}$ obtained solutions in all instances, even the very large ones with 1,300 facility points. This ability to handle the very large instances is due to both the simpler model of the problem and the pruning heuristic. The simpler model means that propagation is not very costly, and therefore an initial solution is sooner or later located. Thereafter, the heuristic takes over and prunes many branches, allowing for the solution to be quickly improved.

On the other hand, the propagation performed by OR-Tools is costly, because of the many auxiliary variables and relevant constraints, meaning that for very large grid problems, which typically have few solutions, it is either unable to reach a solution within the time limit, even if the memory requirements are manageable, or takes very long to improve the ones found. Of course, OR-Tools could also use the simpler model, but in this case the pruning heuristic should be written into the solver as a specialized constraint to make it competitive.

Regarding Gurobi, a factor that seems to affects its performance is the ratio of candidate facility sites to facilities. If this is small then Gurobi quickly locates the optimal solution and proves optimality. In contrast, when it is large then Gurobi finds the problems harder. On the other hand, the performance of CP$_{LS}$, and OR-Tools to a large extent, is dependant on the number of facilities/variables which mainly determines the size of the search space these solvers explore.

Another important factor that affects the performance of the solvers is the number of solutions. Gurobi seems to benefit from the presence of many solutions in an instance, while this does not hold for CP$_{LS}$ and OR-Tools. In contrast, Gurobi finds it hard to deal with problems that only have a few solutions, whereas the CP solvers handle such problems more efficiently. Let us note that p-median generated instances typically have a very large number of solutions. For example, a simple enumeration of the feasible solutions, using a complete CP solver without the objective function, counted 13,343,409 solutions in 10 minutes of cpu time on average for the 10 instances of pmed38 with $|CL| \geq |FP|$. The enumeration was stopped after 10 minutes, meaning that the actual number of solutions could be much higher. Further to this, focusing on two contrasting instances of class g1; one having only 58 feasible solutions and another with 87,950,294 solutions, Gurobi took 3,590 secs to match the best solution of CP$_{LS}$ in the former, while it managed this in 17 secs in the latter.

When there are few solutions then Gurobi is either unable to find any solution within the time limit and/or finds it hard to obtain a good initial bound, meaning that its progress towards the optimal is very slow. In contrast, in the presence of many solutions, it starts with a good initial bound and is able to quickly improve it. The low number of solutions in grid problems is also a possible explanation for the lower quality of solutions discovered by CP$_{LS}$ compared to OR-Tools in classes that OR-Tools can handle. In such cases, erroneous pruning by the heuristic may be costly, as the optimal and near-optimal solutions are few, whereas when there are many solutions, (as in the MDPLIB and p-median benchmarks), mistakes are not as costly, as there may be many branches that lead to good solutions.

8 Conclusion

We have enhanced a recent heuristic approach towards solving the p-dispersion problem with distance constraints in two directions. First, we showed how LS can significantly improve the effectiveness of the method by computing better bound estimations at each node, and therefore performing more focused branch pruning, and to a lesser extent by tightening the bound each time a new solution is discovered. Second, we showed that the entire method is applicable to other location problems, using the p-center problem as an example. Results demonstrated that a solver that heuristically prunes the search space during search is more robust than standard ILP and CP solvers. It would be very interesting to investigate the applicability of the approach to other types of constraint optimization problems.

References

1 T. Argo and E. Sandstrom. Separation distances in NFPA codes and standards (tech. rep.). Fire Protection Research Foundation. 2014.

2 M.L. Balinski. Integer programming: methods, uses, computations. *Management Science*, 12(3):253–313, 1965.

3 J.E. Beasley. A note on solving large p-median problems. *European Journal of Operational Research*, 21(2):270–273, 1985.

4 O. Berman and R. Huang. The minimum weighted covering location problem with distance constraints. *Computers and Operations Research*, 35(12):356–372, 2008. doi:10.1016/j.cor.2006.03.003.

5 F. Boussemart, F. Heremy, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 482–486, 2004.

6 H. Calik and B.C. Tansel. Double bound method for solving the p-center location problem. *Comput. Oper. Res.*, 40(12):2991–2999, 2013. doi:10.1016/j.cor.2013.07.011.

7 H. Cambazard, D. Mehta, B. O'Sullivan, and L. Quesada. A computational geometry-based local search algorithm for planar location problems. In *Proceedings of the 9th International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2012)*, pages 97–112, 2012.

8 S. Chaudhry, T. McCormick, and I. D. Moon. Locating independent facilities with maximum weight: Greedy heuristics. *International Journal of Management Science*, 14(5):383–389, 1986.

9 R. L. Church and M. E. Meadows. Results of a new approach to solving the p-median problem with maximum distance constraints. *Geographical Analysis*, 9(4):364–378, 1977.

10 V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979. doi:10.1287/moor.4.3.233.

11 W. Comley. The location of ambivalent facilities: Use of a quadratic zero-one programming algorithm. *Applied Mathematical Modeling*, 19(1):26–29, 1995.

12 Z. Dai, K. Xu, and M. Ornik. Repulsion-based p-dispersion with distance constraints in non-convex polygons. *Annals of Operations Research*, 307:75–91, 2021. doi:10.1007/s10479-021-04281-z.

13 M.S. Daskin. *Network and discrete location: models, algorithms, and applications, 2nd edn.* Wiley, Hoboken, 2013.

14 Choco development team. An Open-Source java library for constraint programming. https://choco-solver.org/.

15 OR-Tools development team. OR-Tools, CP-SAT solver. https://developers.google.com/optimization/cp/cp_solver.

16 T. Drezner, Z. Drezner, and A. Schöbel. The weber obnoxious facility location model: A big arc small arc approach. *Computers and Operations Research*, 98:240–250, 2018. doi:10.1016/j.cor.2018.06.006.

17 Z. Drezner, P. Kalczynski, and S. Salhi. The planar multiple obnoxious facilities location problem: A Voronoi based heuristic. *Omega*, 87:105–116, 2019.

18 S. Elloumi, M. Labbé, and Y. Pochet. A new formulation and resolution method for the p-center problem. *INFORMS J. Comput.*, 16(1):83–94, 2004. doi:10.1287/ijoc.1030.0028.

19 E. Erkut. The discrete p-dispersion problem. *European Journal of Operational Research*, 46(1):48–60, 1990.

20 E. Erkut and S. Neuman. Analytical models for locating undesirable facilities. *European Journal of Operational Research*, 40(3):275–291, 1989.

21 E. Erkut and S. Neuman. Comparison of four models for dispersing facilities. *Information Systems and Operational Research*, 29:68–86, 1991.

22 E. Erkut, Y. Ülküsal, and O. Yeniçerioglu. A comparison of p-dispersion heuristics. *Computers & Operations Research*, 21(10):1103–1113, 1994. doi:10.1016/0305-0548(94)90041-8.

23 M. M. Fazel-Zarandi and J. C. Beck. Solving a location-allocation problem with logic-based benders' decomposition. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP 2009)*, pages 344–351, 2009.

24 F. Focacci, F. Laburthe, and A. Lodi. Local search and constraint programming. In F.W. Glover and G.A. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 369–403. Kluwer / Springer, 2003. doi:10.1007/0-306-48056-5_13.

25 J. B. Ghosh. Computational aspects of the maximum diversity problem. *Operations Research Letters*, 19(4):175–181, 1996. doi:10.1016/0167-6377(96)00025-9.

26 C. Gomes and M. Sellmann. Streamlined constraint reasoning. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2004)*, pages 274–289, 2004.

27 T. 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*, 2019.

28 LLC Gurobi Optimization. Gurobi optimizer reference manual. , 2023. URL: https://www.gurobi.com.

29 S.L. Hakimi. Optimum locations of switching centers and the absolute centers and medians of a graph. *Operations Research*, 12(3):450–459, 1964.

30 S.L. Hakimi. Optimum distribution of switching centers in a communication network and some related graph theoretic problems. *Operations Research*, 13(3):462–475, 1965.

31 N. Isoart and J.-C. Régin. A k-Opt Based Constraint for the TSP. In *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, 2021.

32 B. M. Khumawala. An efficient algorithm for the p-median problem with maximum distance constraints. *Geographical Analysis*, 5(4):309–321, 1973.

33 J. Krarup, D. Pisinger, and F. Plastria. Discrete location problems with push-pull objectives. *Discrete Applied Mathematics*, 123(1-3):363–378, 2002. doi:10.1016/S0166-218X(01)00346-8.

34 M. J. Kuby. Programming models for facility dispersion: The p-dispersion and maxisum dispersion problems. *Mathematical and Computer Modelling*, 10(10):792, 1988.

35 A.A. Kuehn and M.J. Hamburger. A heuristic program for locating warehouses. *Management Science*, 9:643–666, 1963.

36 M.Z. Lagerkvist and M. Rattfeldt. Half-checking propagators. In *Proceedings of the 19th Workshop on Constraint Modelling and Reformulation*, 2020.

37 R. Marti, A. Martinez-Gavara, S. Perez-Pelo, and J. Sanchez-Oro. A review on discrete diversity and dispersion maximization from an OR perspective. *European Journal of Operational Research*, 299(3):795–813, 2022. doi:10.1016/j.ejor.2021.07.044.

38 I. D. Moon and S. Chaudhry. An analysis of network location problems with distance constraints. *Management Science*, 30(3):290–307, 1984.

39 I. D. Moon and L. Papayanopoulos. Minimax location of two facilities with minimum separation: Interactive graphical solutions. *Journal of the Operations Research Society*, 42:685–694, 1991.

40 A.T. Murray, R.L. Church, R.A. Gerrard, and W.S. Tsui. Impact models for siting undesirable facilities. *Papers in Regional Science*, 77(1):19–36, 1998.

41 N. Ploskas, K. Stergiou, and D.C. Tsouros. The p-dispersion problem with distance constraints. 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 30:1–30:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CP.2023.30.

42 D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence, Amsterdam, The Netherlands, August 8-12, 1994*, pages 125–129, 1994.

43 D. Sayah and S. Irnich. A new compact formulation for the discrete p-dispersion problem. *European Journal of Operational Research*, 256(1):62–67, 2017. doi:10.1016/j.ejor.2016.06.036.

44 F. Sayyady and Y Fathi. An integer programming approach for solving the p-dispersion problem. *European Journal of Operational Research*, 253(1):216–225, 2016. doi:10.1016/j.ejor.2016.02.026.

45 D. R. Shier. A min-max theorem for p-center problems on a tree. *Transportation Science*, 11:243–252, 1977.

46 S.Y.D. Sorkhabi, D.A. Romero, J.C. Beck, and C. Amon. Constrained multi-objective wind farm layout optimization: Novel constraint handling approach based on constraint programming. *Renewable Energy*, 126(C):341–353, 2018.

47 B.C. Tansel, R.L. Francis, T.J. Lowe, and M.L. Chen. Duality and distance constraints for the nonlinear p-center problem and covering problem on a tree network. *Operations Research*, 30(4):725–744, 1982. doi:10.1287/opre.30.4.725.

48 S.B. Welch and S. Salhi. The obnoxious p facility network location problem with facility interaction. *European Journal of Operations Research*, 102:302–319, 1997.

49 49 C.F.R. §175.701. Separation distance requirements for packages containing class 7 (radioactive) materials in passenger-carrying aircraft. Title 49 Code of Federal Regulations, Part 175. 2021.

50 29 C.F.R. §1910.157. Portable fire extinguishers. Title 29 Code of Federal Regulations, Part 157. 2021.

Appendix

In Table 4 we give mean results regarding the pruning of the heuristic and the effect of LS. CP$_{LS_h}$ is the solver configuration that uses LS only within the branch pruning heuristic to perform a potential improvement in the estimation of the bound. We denote by R_a the ratio of branches accepted (not cut off) by LS (within the branch pruning heuristic) to the total number of times that it was called. We denote by R_b the ratio of the total number of branches pruned by the heuristic to the number of times the heuristic was called. We denote by R_i, the ratio of the number of times that LS further improved a feasible solution to the number of times that it was called to do so. The ratios are presented as percentages and are computed as means over all the instances of a problem category (MDG and GKD in pDDs - p-median and grid in pCDs).

The high percentages of R_b explain the efficiency of the solver that applies this type of pruning, as large amounts of the search tree are cut off (more than 90% of the calls to the heuristic, on average, result in branch pruning, in both categories of the pCD and one of the pDD). As expected, the basic CP$_G$ variant usually displays a higher pruning ratio compared to CP$_{LS}$, but not always, because in some instances, especially p-median based pCDs, the latter quickly finds very good solutions, tightening the bound, and thereby resulting in heavy pruning. Regarding the effect of LS as a means to improve newly discovered solutions, in the GKD classes of pDDs and the p-median classes of pCDs there was an improving ratio R_i of around 60%, meaning that LS was able to improve an incumbent solution more than half of the times it was called to do so.

Table 4 Ratios for pDDs and pCDs.

	R_a	R_b	R_i	R_a	R_b	R_i
pDD	MDG			GKD		
CP$_G$	-	91.78%	-	-	85.18%	-
CP$_{LS}$	4.16%	91.22%	35.64%	14.15%	79.94%	58.89%
CP$_{LS_h}$	4.19%	90.82%	-	14.12%	79.54%	-
pCD	p-median			Grid		
CP$_G$	-	90.55%	-	-	95.35%	-
CP$_{LS}$	0.36%	91.78%	60.91%	2.76%	93.79%	31.34%
CP$_{LS_h}$	0.61%	90.24%	-	3.08%	92.96%	-

In Table 5 an indicative comparison between the basic solver fo pCDs (denoted CP$_{de}$), i.e. the solver with the branch pruning heuristic deactivated, and CP$_{LS}$ is presented. For each class, we report the total cpu times taken by the solvers, the mean cost of the best solution found, and the number of instances where any solver reached the cut off limit of 1 hour without terminating (in brackets after the total cpu time). Column t$_b$ gives the mean cpu time taken by the solvers to find their best solution. Column t$_m$ gives the mean cpu time taken by CP$_{de}$ to find the first solution that matches (or improves) the cost of the best solution found by CP$_{LS}$. If CP$_{de}$ did not manage to match the solution found by CP$_{LS}$ in some instances, t$_m$ is left blank. Finally, we also report the total nodes visited by each solver.

Table 5 Comparing CP solver with/without the branch pruning heuristic in pCDs.

	CP$_{de}$					CP$_{LS}$			
Class	$\sum$cpu	t$_b$	t$_m$	cost	$\sum$nodes	$\sum$cpu	t$_b$	cost	$\sum$nodes
$\lvert CL\rvert \geq \lvert FP\rvert$									
pmed21	2,775 (0)	19	8	45.9	72,202,355	6 (0)	0.3	46.5	2,995
pmed36	>21,600 (6)	406	-	37.8	307,856,397	478 (0)	19	37	15,731
pmed38	>12,355 (3)	286	-	37.5	269,869,194	52 (0)	3	37.3	8,960
$\lvert FP\rvert > \lvert CL\rvert$									
pmed21	>36,000 (10)	1,292	-	29.4	3,479,429,138	101 (0)	5	28.6	26,045
pmed36	>36,000 (10)	643	-	22.7	1,365,670,278	1,697 (0)	110	20.7	74,489
pmed38	>36,000 (10)	430	-	21.1	2,996,757,611	184 (0)	8	20.9	24,921
$GRID$									
g3	>32,788 (9)	381	17	4.1	67,219,121	205 (0)	14	5.3	154,500
g6	>36,000 (10)	751	88	9.2	250,773,552	99 (0)	0.5	9.9	11,230
g8	>31,509 (8)	918	20	7.6	23,653,359	275 (0)	10	9	55,194

Not surprisingly, the behaviour of CP$_{de}$ resembles somewhat that of OR-Tools. It is by far slower than its incomplete version CP$_{LS}$, and typically discovers worse solutions, in p-median based problems. On the other hand, it is still slower in grid problems (though by smaller margins), but it discovers better solutions than CP$_{LS}$. The huge numbers of visited nodes compared to CP$_{LS}$, demonstrate the pruning power of the branch pruning heuristic, although results from the grid classes indicate that this massive pruning can sometimes be costly, in terms of solution quality, when problems have a relatively small amount of solutions.

In Table 6 we give an indicative comparison between two different configurations of OR-Tools, one single threaded and one with 8 threads. For each class, we report the total cpu times taken, the mean cost of the best solution found, and the number of instances where the cut off limit of 1 hour without terminating is reached (in brackets after the total cpu time).

Table 6 Comparing ORtools with 1 and 8 threads.

	ORt$_1$ $\sum$cpu	t$_b$	cost	ORt$_8$ $\sum$cpu	t$_b$	cost
$\lvert CL\rvert \geq \lvert FP\rvert$						
pmed5	1,668 (0)	167	100.5	872 (0)	86	100.5
pmed10	1,963 (0)	196	63.4	1,902 (0)	188	63.4
pmed21	212 (0)	18	45.8	203 (0)	16	45.8
pmed26	224 (0)	19	43.2	195 (0)	18	43.2
$\lvert FP\rvert > \lvert CL\rvert$						
pmed5	249 (0)	25	47	249 (0)	24	47
pmed10	10,775 (0)	1,077	25.3	10,726 (0)	1,071	25.3
pmed21	>19,747 (1)	497	27.6	10,173 (0)	319	27.6
pmed26	14,901 (0)	604	24.7	9,775 (0)	525	24.7
GRID						
g1	443 (0)	30	2.1	272 (0)	14	2.1
g2	7,737 (0)	98	4.1	8,178 (0)	158	4.1
g3	>36,000 (10)	396	4.2	>35,977 (9)	354	4.1
g6	>34,641 (9)	358	7.4	>35,334 (9)	732	7.4

Results demonstrate that the 8 thread configuration for OR-Tools did not result in significantly improved run times (sometimes it is detrimental). In most classes, the mean times taken for the discovery of the best solution were close. But importantly, the 8 thread variant very rarely manages to improve the cost of the best solution found. Finally, the two versions crashed in the same classes that are denoted with MEM in Table 3.

CSPs with Few Alien Constraints

Peter Jonsson ✉
Department of Computer and Information Science, Linköping University, Sweden

Victor Lagerkvist ✉
Department of Computer and Information Science, Linköping University, Sweden

George Osipov ✉
Department of Computer and Information Science, Linköping University, Sweden

Abstract

The *constraint satisfaction problem* asks to decide if a set of constraints over a relational structure $\mathcal{A}$ is satisfiable (CSP($\mathcal{A}$)). We consider CSP($\mathcal{A} \cup \mathcal{B}$) where $\mathcal{A}$ is a structure and $\mathcal{B}$ is an *alien* structure, and analyse its (parameterized) complexity when at most k alien constraints are allowed. We establish connections and obtain transferable complexity results to several well-studied problems that previously escaped classification attempts. Our novel approach, utilizing logical and algebraic methods, yields an FPT versus pNP dichotomy for arbitrary finite structures and sharper dichotomies for Boolean structures and first-order reducts of $(\mathbb{N}, =)$ (equality CSPs), together with many partial results for general ω-categorical structures.

2012 ACM Subject Classification Theory of computation → Complexity theory and logic; Mathematics of computing → Discrete mathematics

Keywords and phrases Constraint satisfaction, parameterized complexity, hybrid theories

Digital Object Identifier 10.4230/LIPIcs.CP.2024.15

Funding *Peter Jonsson and George Osipov*: Supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.
Peter Jonsson: Partially supported by the Swedish research council under grant VR-2021-04371.
Victor Lagerkvist: Partially supported by the Swedish research council under grant VR-2022-03214.

Acknowledgements We thank the anonymous reviewers for helpful suggestions for how to improve the presentation of the paper.

1 Introduction

The *constraint satisfaction problem* over a structure $\mathcal{A}$ (CSP($\mathcal{A}$)) is the problem of verifying whether a set of constraints over $\mathcal{A}$ admits at least one solution. This problem framework is vast, and, just to name a few, include all Boolean satisfiability problems as well as k-coloring problems, and for infinite domains we may formulate both problems centrally related to model checking first-order formulas and qualitative reasoning. Notable examples where complete complexity dichotomies are known (separating tractable from NP-hard problems) include *all* finite structures [12, 24] and first-order definable relations over well-behaved base structures like $(\mathbb{N}, =)$ and $(\mathbb{Q}, <)$ [2]. While impressive mathematical achievements, these dichotomy results are still somewhat unsatisfactory from a practical perspective since we are unlikely to encounter instances which are based on *purely* tractable constraints. Could it be possible to extend the reach of these powerful theoretical results by relaxing the basic setting so that we may allow greater flexibility than purely tractable constraints while still obtaining something simpler than an arbitrary NP-hard CSP?

We consider this problem in a *hybrid* setting via problems of the form CSP($\mathcal{A} \cup \mathcal{B}$) where $\mathcal{A}$ is a "stable", tractable background structure and $\mathcal{B}$ is an *alien* structure. We focus on the case when CSP($\mathcal{A} \cup \mathcal{B}$) is NP-hard (thus, richer than a polynomial-time solvable problem) but where we have comparably few constraints from the alien structure $\mathcal{B}$. This problem is

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 15; pp. 15:1–15:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

compatible with the influential framework of *parameterized complexity* which has been used with great effect to study *structurally* restricted problems (e.g., based on tree-width) but where comparably little is known when one simultaneously restricts the allowed constraints.

We begin (in Section 3) by relating the CSP problem with alien constraints to other problems, namely, (1) *model checking*, (2) the problem of checking whether a constraint in a CSP instance is *redundant*, (3) the *implication* problem and (4) the *equivalence* problem. We prove that the latter three problems are equivalent under Turing reductions and provide a general method for obtaining complexity dichotomies for all of these problems via a complexity dichotomy for the CSP problem with alien constraints. Importantly, all of these problems are well-known in their own right, but have traditionally been studied with wildly disparate tools and techniques, but by viewing them under the unifying lens of alien constraints we not only get four dichotomies for the price of one but also open the powerful toolbox based on *universal algebra*. For non-Boolean domains this is not only a simplifying aspect but an absolute necessity to obtain general results. We expand upon the algebraic approach in Section 4 and relate alien constraints to *primitive positive definitions* (pp-definitions) and the important notion of a *core*. As a second general contribution we explore the case when each relation in $\mathcal{B}$ can be defined via an *existential positive formula* over $\mathcal{A}$. This results in a general *fixed-parameter tractable* (FPT) algorithm (with respect to the number of alien constraints) applicable to both finite, and, as we demonstrate later, many natural classes of structures over infinite domains.

In the second half of the paper we attack the complexity of alien constraints more systematically. We begin with structures over finite domains where we obtain a general tractability result by combining the aforementioned FPT algorithm together with the CSP dichotomy theorem [12, 24]. In a similar vein we obtain a general hardness result based on a universal algebraic gadget. Put together this yields a general result: if $\mathcal{A} \cup \mathcal{B}$ is a core (which we may assume without loss of generality) then either $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is FPT, or $\mathrm{CSP}_{\leq p}(\mathcal{A} \cup \mathcal{B})$ is NP-hard for some $p \geq 0$, i.e., is *para-NP-hard* (pNP-hard). Thus, from a parameterized complexity view we obtain a complete dichotomy (FPT versus pNP-hardness) for finite-domain structures. However, to also obtain dichotomies for implication, equivalence, and the redundancy problem, we need sharper bounds on the parameter p. We concentrate on two special cases. We begin with Boolean structures in Section 5.2 and obtain a complete classification which e.g. states that $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is FPT if $\mathcal{A}$ is in one of the classical *Schaefer* classes, and give a precise characterization of $\mathrm{CSP}_{\leq p}(\mathcal{A} \cup \mathcal{B})$ for all relevant values of p if $\mathcal{A}$ is not Schaefer. For example, if we assume that $\mathcal{A}$ is Horn, we may thus conclude that $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is FPT for *any* alien Boolean structure $\mathcal{B}$. More generally this dichotomy is sufficiently sharp to also yield dichotomies for implication, equivalence, and redundancy. Compared to the proofs by Schnoor & Schnoor [22] for implication and Böhler [11] for equivalence, we do not use an exhaustive case analysis over Post's lattice.

In Section 6 we consider structures over infinite domains. If we assume that $\mathcal{A}$ and $\mathcal{B}$ are ω-categorical, then we manage to lift the FPT algorithm based on existential positive definability from Section 4 to the infinite setting. Another important distinction is that the notion of a core, and subsequently the common trick of singleton expansion, works differently for ω-categorical languages. Here we follow Bodirsky [2] and use the notion of a *model-complete core*, which means that all n-ary orbits are pp-definable, where an orbit is defined as the action of the automorphism group over a fixed n-ary tuple. This allows us to, for example, prove that $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is FPT whenever $\mathcal{A}$ is an ω-categorical model-complete core and $\mathrm{CSP}(\mathcal{A})$ is in P such that the orbits of the automorphism group of $\mathcal{B}$ are included in the orbits of the automorphism group of $\mathcal{A}$. This forms a cornerstone for the dichotomy for equality languages since the only remaining cases are when $\mathcal{A}$ is 0-valid (meaning that each relation contains a constant tuple) but not Horn (defined similarly to the Boolean domain), and when $\mathcal{B}$ is not 0-valid. The remaining cases are far from trivial, however, and we require the algebraic machinery from Bodirsky et al. [4] which provides a characterization of equality languages in terms of their *retraction* to finite domains. We rely on this description via a recent classification result by Osipov & Wahlström [19]. Importantly, our dichotomy result is sufficiently sharp to additionally obtain complexity dichotomies for the implication, equivalence, and redundancy problems. To the best of our knowledge, these dichotomies are the first of their kind for arbitrary equality languages.

We finish the paper with a comprehensive discussion in Section 7. Most importantly, we have opened up the possibility to systematically study not only alien constraints, but also related problems that have previously escaped complexity classifications. For future research the main open questions are whether (1) sharper results can be obtained for arbitrary finite domains and (2) which further classes of infinite domain structures should be considered.

Proofs of statements marked with ($\star$) have been removed due to space constraints.

2 Preliminaries

We begin by introducing the basic terminology and the fundamental problems under consideration. We assume throughout the paper that the complexity classes P and NP are distinct. We let $\mathbb{Q}$ denote the rationals, $\mathbb{N} = \{0, 1, 2, \ldots\}$ the natural numbers, $\mathbb{Z} = \{\ldots, -2.-1, 0, 1, 2, \ldots\}$ the integers, and $\mathbb{Z}_+ = \{1, 2, 3, \ldots\}$ the positive integers. For every $c \in \mathbb{Z}_+$, we let $[c] = \{1, 2, \ldots, c\}$.

A *parameterized problem* is a subset of $\Sigma^* \times \mathbb{N}$ where Σ is the input alphabet, i.e., an instance is given by $x \in \Sigma^*$ of size n and a natural number k, and the running time of an algorithm is studied with respect to both k and n. The most favourable complexity class is FPT (*fixed-parameter tractable*), which contains all problems that can be decided in $f(k) \cdot n^{O(1)}$ time with f being some computable function. An *fpt-reduction* from a parameterized problem $L_1 \subseteq \Sigma_1^* \times \mathbb{N}$ to $L_2 \subseteq \Sigma_2^* \times \mathbb{N}$ is a function $P : \Sigma_1^* \times \mathbb{N} \to \Sigma_2^* \times \mathbb{N}$ that preserves membership (i.e., $(x, k) \in L_1 \Leftrightarrow P((x, k)) \in L_2$), is computable in $f(k) \cdot |x|^{O(1)}$ time for some computable function f, and there exists a computable function g such that for all $(x, k) \in L_1$, if $(x', k') = P((x, k))$, then $k' \leq g(k)$. It is easy to verify that if L_1 and L_2 are parameterized problems such that L_1 fpt-reduces to L_2 and L_2 is in FPT, then it follows that L_1 is in FPT, too. There are many parameterized classes with less desirable running times than FPT but we focus on pNP-hard problems: a problem is pNP-hard under fpt-reductions if it is NP-hard for some constant parameter value, implying such problems are not in FPT unless $\mathsf{P} = \mathsf{NP}$.

We continue by defining *constraint satisfaction problems*. First, a *constraint language* is a (typically finite) set of relations $\mathcal{A}$ over a universe A, and for a relation $R \in \Gamma$ we write $\mathrm{ar}(R) = k$ to denote its arity k. It is sometimes convenient to associate a constraint language with a relational signature, and thus obtaining a *relational structure*: a tuple $(A; \tau, I)$ where A is the *domain*, or *universe*, τ is a relational signature, and I is a function from σ to the set of all relations over D which assigns each relation symbol R a corresponding relation $R^{\mathcal{A}}$ over D. We write $\mathrm{ar}(R)$ for the arity of a relation R, and if $R = \emptyset$ then $\mathrm{ar}(R) = 0$. All structures in this paper are relational and we assume that they have a finite signature unless otherwise stated. Typically, we do not need to make a sharp distinction between relations and the corresponding relation symbols, so we usually simply write $(A; R_1, \ldots, R_m)$, where each R_i is a relation over A, to denote a structure. We also sometimes do not make a sharp distinction between structures and sets of relations when the signature is not important. For arbitrary structures $\mathcal{A}$ and $\mathcal{A}'$ with domains A and A', we let $\mathcal{A} \cup \mathcal{A}'$ denote the structure with domain $A \cup A'$ and containing the relations in $\mathcal{A}$ and $\mathcal{A}'$.

For a constraint language (or structure) $\mathcal{A}$ an instance of the *constraint satisfaction problem* over $\mathcal{A}$ ($\mathrm{CSP}(\mathcal{A})$) is then given by $I = (V, C)$ where V is a set of variables and C a set of constraints of the form $R(x_1, \ldots, x_k)$ where $x_1, \ldots, x_k \in V$ and $R \in \mathcal{A}$, and the question is whether there exist a function $f \colon V \to A$ that satisfies all constraints (a *solution*), i.e., $(f(x_1), \ldots, f(x_k)) \in R$ for all $R(x_1, \ldots, x_k) \in C$. The *CSP dichotomy theorem* says that all finite-domain CSPs are either in P or are NP-complete [12, 24]. Given an instance $I = (V, C)$ of $\mathrm{CSP}(\mathcal{A})$, we let $\mathrm{Sol}(I)$ be the set of solutions to I. We now define CSPs with alien constraints in the style of Cohen et al. [14].

$\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$

Instance: A natural number k and an instance $I = (V, C_1 \cup C_2)$ of $\mathrm{CSP}(\mathcal{A} \cup \mathcal{B})$, where (V, C_1) is an instance of $\mathrm{CSP}(\mathcal{A})$ and (V, C_2) is an instance of $\mathrm{CSP}(\mathcal{B})$ with $|C_2| \leq k$.
Question: Does there exist a satisfying assignment to I?

Throughout the paper, we assume without loss of generality that the structures $\mathcal{A}$ and $\mathcal{B}$ can be associated with disjoint signatures. The parameter in $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is the *number of alien constraints* (abbreviated #ac). We let $\mathrm{CSP}_{\leq k}(\mathcal{A} \cup \mathcal{B})$ denote the $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ problem restricted to a fixed value k of parameter #ac. Note that if $\mathrm{CSP}(\mathcal{A})$ is not in P, then $\mathrm{CSP}_{\leq 0}(\mathcal{A} \cup \mathcal{B})$ is not in P; moreover, if $\mathrm{CSP}(\mathcal{A} \cup \mathcal{B})$ is in P, then $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is in P. Thus, it is sensible to always require that $\mathrm{CSP}(\mathcal{A})$ is in P and $\mathrm{CSP}(\mathcal{A} \cup \mathcal{B})$ is not in P. In many natural cases (e.g., all finite-domain CSPs), $\mathrm{CSP}(\mathcal{A} \cup \mathcal{B})$ not being polynomial-time solvable implies that $\mathrm{CSP}(\mathcal{A} \cup \mathcal{B})$ is NP-hard.

A k-ary relation R is said to have a *primitive positive definition* (pp-definition) over a constraint language $\mathcal{A}$ if $R(x_1, \ldots, x_k) \equiv \exists y_1, \ldots, y_{k'} : R_1(\mathbf{x_1}) \land \ldots \land R_m(\mathbf{x_m})$ where each $R_i \in \mathcal{A} \cup \{=_A\}$ and each $\mathbf{x_i}$ is a tuple of variables over $x_1, \ldots, x_k$, $y_1, \ldots, y_{k'}$ matching the arity of R_i. Here, and in the sequel, $=_A$ is the equality relation over A, i.e. $\{(a, a) \mid a \in A\}$. If $\mathcal{A}$ is a constraint language, then we let $\langle \mathcal{A} \rangle$ be the inclusion-wise smallest set of relations containing $\mathcal{A}$ closed under pp-definitions.

▶ **Theorem 1** ([16]). *Let $\mathcal{A}$ and $\mathcal{B}$ be structures with the same domain. If every relation of $\mathcal{A}$ has a primitive positive definition in $\mathcal{B}$, then there is a polynomial-time reduction from CSP($\mathcal{A}$) to CSP($\mathcal{B}$).*

When working with problems of the form $\mathrm{CSP}_{\leq k}(\mathcal{A} \cup \mathcal{B})$ we additionally introduce the following simplifying notation: $\langle \mathcal{A} \cup \mathcal{B} \rangle_{\leq k}$ denotes the set of all pp-definable relations over $\mathcal{A} \cup \mathcal{B}$ using at most k atoms from $\mathcal{B}$. We now describe the corresponding algebraic objects. An operation $f \colon D^m \to D$ is a *polymorphism* of a relation $R \subseteq D^k$ if, for any choice of m tuples $(t_{11}, \ldots, t_{1k}), \ldots, (t_{m1}, \ldots, t_{mk})$ from R, it holds that $(f(t_{11}, \ldots, t_{m1}), \ldots, f(t_{1k}, \ldots, t_{mk}))$ is in R. An *endomorphism* is a polymorphism with arity one. If f is a polymorphism of R, then we sometimes say that R is *invariant* under f. A constraint language $\mathcal{A}$ has the polymorphism f if every relation in $\mathcal{A}$ has f as a polymorphism. We let $\mathrm{Pol}(\mathcal{A})$ and $\mathrm{End}(\mathcal{A})$ denote the sets of polymorphisms and endomorphisms of $\mathcal{A}$, respectively. If F is a set of functions over D, then $\mathrm{Inv}(F)$ denotes the set of relations over D that are invariant under every function in F. There are close algebraic connections between the operators $\langle \cdot \rangle$, $\mathrm{Pol}(\cdot)$, and $\mathrm{Inv}(\cdot)$. For instance, if $\mathcal{A}$ has a finite domain (or, more generally, if $\mathcal{A}$ is ω-categorical; see below), then we have a Galois connection $\langle \mathcal{A} \rangle = \mathrm{Inv}(\mathrm{Pol}(\mathcal{A}))$ [9, Theorem 5.1].

Polymorphisms enable us to compactly describe the tractable cases of Boolean CSPs.

▶ **Theorem 2** ([21]). *Let $\mathcal{A}$ be a constraint language over the Boolean domain. The problem CSP($\mathcal{A}$) is decidable in polynomial time if $\mathcal{A}$ is invariant under one of the following six operations: (1) the constant unary operation 0 ($\mathcal{A}$ is 0-valid), (2) the constant unary operation 1 ($\mathcal{A}$ is 1-valid), (3) the binary min operation $\sqcap$ ($\mathcal{A}$ is Horn), (4) the binary max operation $\sqcup$ ($\mathcal{A}$ is anti-Horn), (5) the ternary majority operation $M(x,y,z) = (x \sqcap y) \sqcup (x \sqcap z) \sqcup (y \sqcap z)$ ($\mathcal{A}$ is 2-SAT), or (6) the ternary minority operation $m(x,y,z) = x \oplus y \oplus z$ where $\oplus$ is the addition operator in $GF(2)$ ($\mathcal{A}$ is affine). Otherwise, the problem CSP($\mathcal{A}$) is NP-complete.*

A Boolean constraint language that satisfies condition (3), (4), (5), or (6) is called *Schaefer*.

A finite-domain structure $\mathcal{A}$ is a *core* if every $e \in \mathrm{End}(\mathcal{A})$ is a bijection. We let $f(R) = \{(f(t_1), \ldots, f(t_n)) \mid (t_1, \ldots, t_n) \in R\}$ when $f \colon A \to A$ and $R \in \mathcal{A}$. If $e \in \mathrm{End}(\mathcal{A})$ has minimal range, then $e(\mathcal{A}) = \{e(R) \mid R \in \mathcal{A}\}$ is a core and this core is unique up to isomorphism. We can thus speak about *the core* $\mathcal{A}^c$ of $\mathcal{A}$. It is easy to see that CSP($\mathcal{A}$) and CSP($\mathcal{A}^c$) are equivalent under polynomial-time reductions (indeed, even log-space reductions suffice). Another useful equivalence concerns constant relations. Let $\mathcal{A}^+$ denote the structure $\mathcal{A}$ expanded by all unary singleton relations $\{(a)\}$, $a \in A$. If $\mathcal{A}$ is a core, then CSP($\mathcal{A}$) and CSP($\mathcal{A}^+$) are equivalent under polynomial-time reductions [1].

We will frequently consider ω-*categorical* structures. An *automorphism* of a structure $\mathcal{A}$ is a permutation α of its domain A such that both α and its inverse are homomorphisms. The set of all automorphisms of a structure $\mathcal{A}$ is denoted by Aut($\mathcal{A}$), and forms a group with respect to composition. The *orbit* of $(a_1, \ldots, a_n) \in A^n$ in Aut($\mathcal{A}$) is the set $\{(\alpha(a_1), \ldots, \alpha(a_n)) \mid \alpha \in \mathrm{Aut}(\mathcal{A})\}$. Let Orb($\mathcal{A}$) denote the set of orbits of n-tuples in Aut($\mathcal{A}$) (for all $n \geq 1$). A structure $\mathcal{A}$ with countable domain is ω-categorical if and only if Aut($\mathcal{A}$) is *oligomorphic*, i.e., it has only finitely many orbits of n-tuples for all $n \geq 1$.

Two important classes of ω-categorical structures are *equality languages* (respectively, *temporal languages*) where each relation can be defined as the set of models of a first-order formula over $(\mathbb{N}; =)$ (respectively, $(\mathbb{Q}; <)$). Importantly, Aut($\mathcal{A}$) is the full symmetric group if $\mathcal{A}$ is an equality language. A relation in an equality language is said to be *0-valid* if it contains *any* constant tuple. This is justified since if the relation is invariant under one constant operation, then it is invariant under all constant operations.The computational complexity of CSP for equality languages was classified by Bodirsky and Kára [7, Theorem 1]: for any equality language $\mathcal{A}$, CSP($\mathcal{A}$) is solvable in polynomial time if $\mathcal{A}$ is 0-valid or invariant under a binary injective operation, and is NP-complete otherwise.

3 Applications of Alien Constraints

We will now demonstrate how alien constraints can be used for studying the complexity of CSP-related problem: Section 3.1 contains an example where we analyse the complexity of *redundancy*, *equivalence*, and *implication* problems, and we consider connections between the model checking problem and CSPs with alien constraints in Section 3.2. To relate problem complexity we use *Turing reductions*: a problem L_1 is *polynomial-time Turing reducible* to L_2 (denoted $L_1 \leq^p_T L_2$) if it can be solved in polynomial time using an oracle for L_2. Two problems L_1 and L_2 are *polynomial-time Turing equivalent* if $L_1 \leq^p_T L_2$ and $L_2 \leq^p_T L_1$.

3.1 The Redundancy Problem and its Relatives

We will now study the complexity of a family of well-known computational problems. We begin by some definitions. Let $\mathcal{A}$ denote a constraint language and assume that $I = (V, C)$ is an instance of CSP($\mathcal{A}$). We say that a constraint $c \in C$ is *redundant* in I if $\mathrm{Sol}((V, C)) = \mathrm{Sol}((V, C \setminus \{c\}))$. We have the following computational problems.

Redundant($\mathcal{A}$)

Instance: An instance (V,C) of CSP($\mathcal{A}$) and a constraint $c \in C$.
Question: Is c redundant in (V,C)?

Impl($\mathcal{A}$)

Instance: Two instances $(V,C_1),(V,C_2)$ of CSP($\mathcal{A}$).
Question: Does (V,C_1) imply (V,C_2), i.e., is it the case that $\mathrm{Sol}((V,C_1)) \subseteq \mathrm{Sol}((V,C_2))$?

Equiv($\mathcal{A}$)

Instance: Two instances $(V,C_1),(V,C_2)$ of CSP($\mathcal{A}$).
Question: Is it the case that $\mathrm{Sol}((V,C_1)) = \mathrm{Sol}((V,C_2))$?

Before we start working with alien constraints, we exhibit a close connection between Redundant($\cdot$), Equiv($\cdot$), and Impl($\cdot$).

▶ **Lemma 3.** *Let $\mathcal{A}$ be a structure. The problems Equiv($\mathcal{A}$), Impl($\mathcal{A}$), and Redundant($\mathcal{A}$) are polynomial-time Turing equivalent.*

Proof. We show that (1) Equiv($\mathcal{A}$) $\leq^p_T$ Impl($\mathcal{A}$), (2) Impl($\mathcal{A}$) $\leq^p_T$ Redundant($\mathcal{A}$), and (3) Redundant($\mathcal{A}$) $\leq^p_T$ Equiv($\mathcal{A}$).

(1). Let $((V,C_1),(V,C_2))$ be an instance of Equiv($\mathcal{A}$). We need to check whether $\mathrm{Sol}((V,C_1)) = \mathrm{Sol}((V,C_2))$. This is true if and only if the two Impl instances $((V,C_1),(V,C_2))$ and $((V,C_2),(V,C_1))$ are yes-instances.

(2). Let $((V,C_1),(V,C_2))$ be an instance of Impl($\mathcal{A}$). For each constraint $c \in C_2$, first check whether C_1 implies $\{c\}$ by (a) checking if $c \in C_1$, in which case C_1 trivially implies $\{c\}$, (b) if not, then check whether c is redundant in $C_1 \cup \{c\}$, in which case we answer yes, and otherwise no. If C_1 implies $\{c\}$ for every $c \in C_2$ then C_1 implies C_2 and we answer yes, and otherwise no.

(3). Let $I = ((V,C),c)$ be an instance of Redundant($\mathcal{A}$). It is obvious that I is a yes-instance if and only if the instance $((V,C),(V,C\setminus\{c\}))$ is a yes-instance of Equiv($\mathcal{A}$). ◀

Next, we show how the complexity of Redundant($\mathcal{A}$) can be analysed by exploiting CSPs with alien constraints. If R is a k-ary relation over domain D, then we let $\bar{R}$ denote its *complement*, i.e. $\bar{R} = D^k \setminus R$.

▶ **Theorem 4** ($\star$). *Let $\mathcal{A}$ be a structure with domain A. If CSP($\mathcal{A}$) is not in* P*, then Redundant($\mathcal{A}$) is not in* P*. In particular, Redundant($\mathcal{A}$) is* NP*-hard (under polynomial-time Turing reductions) whenever CSP($\mathcal{A}$) is* NP*-hard. Otherwise, Redundant($\mathcal{A}$) is in* P *if and only if for every relation $R \in \mathcal{A}$, CSP$_{\leq 1}(\mathcal{A} \cup \{\bar{R}\})$ is in* P.

Combining Theorem 4 with the forthcoming complexity classification of Boolean CSPs with alien constraints (Theorem 14) shows that Boolean Redundant($\mathcal{A}$) is in P if and only if $\mathcal{A}$ is Schaefer. We have not found this result in the literature but we view it as folklore since it follows from other classification results (start from [11] or [22] and transfer the results to Redundant($\mathcal{A}$) with the aid of Lemma 3). However, we claim that our proof is very different when compared to the proofs in [11] and [22]): Böhler et al. use a lengthy case analysis while Schnoor & Schnoor in addition uses the so-called weak base method, which scales poorly since not much is known about this construction for non-Boolean domains. We do not claim that our proof is superior, but we do not see how to generalize the classifications by Böhler et al. and Schnoor & Schnoor to larger (in particular infinite) domains since they are fundamentally based on Post's classification of Boolean clones. Such a generalization, on the other hand, is indeed possible with our approach. We demonstrate in Section 6.2

that we can obtain a full understanding of the complexity of CSPs with alien constraints for equality languages. This result carries over to Redundant$(\cdot)$ via Theorem 4, implying that we have a full complexity classification of Redundant$(\cdot)$ for equality languages. This result can immediately be transferred to Impl$(\cdot)$ and Equiv$(\cdot)$ by Lemma 3.

3.2 Model Checking

We follow [18] and view the *model checking* problem as follows: given a logic $\mathscr{L}$, a structure $\mathcal{A}$, and a sentence ϕ of $\mathscr{L}$, decide whether $\mathcal{A} \models \phi$. The main motivation for this problem is its connection to databases [23]. From the CSP perspective, we consider a slightly reformulated version: given an instance $I = (V, C)$ of CSP$(\mathcal{A})$ and a formula ϕ with free variables in V, we ask if there is a tuple in Sol(I) that satisfies ϕ. If ϕ can be expressed as an instance I' of CSP$(\mathcal{B})$ for some structure $\mathcal{B}$, then this is the same thing as if asking whether $I \cup I'$ has a solution or not. In the model-checking setting, we want to check whether ϕ is true in all solutions of I. If $\neg\phi$ can be expressed as an instance I' of CSP$(\mathcal{B})$ for some structure $\mathcal{B}$, then we are done: every solution to I satisfies ϕ if and only if CSP$(I \cup I')$ is not satisfiable, and this clarifies the connection with CSPs with alien constraints. For instance, one may view Impl$(\mathcal{A})$ (and consequently the underlying CSP$_{\leq 1}(\mathcal{A} \cup \bar{R})$ problems by Lemma 3 and Theorem 4) as the model checking problem restricted to queries that are $\mathcal{A}$-sentences constructed using the operators $\forall$ and $\vee$. Naturally, one wants the ability to use more complex queries such as (1) queries extended with other relations, i.e. queries constructed over an expanded structure, or (2) queries that are built using other logical connectives.

In both cases, it makes sense to study the fixed-parameter tractability of CSP$_{\leq}(\mathcal{A} \cup \mathcal{B})$ with parameter $\#\mathsf{ac}$ since the query is typically much smaller than the structure $\mathcal{A}$. The connection is quite obvious in the first case (one may view $\#\mathsf{ac}$ as measuring how "complex" the given query is) while it is more hidden in the second case. Let us therefore consider the negation operator. From a logical perspective, one may view a constraint $\bar{R}(x_1, \ldots, x_k)$ as the formula $\neg R(x_1, \ldots, x_k)$. Needless to say, the relation $\bar{R}$ is often not pp-definable in a structure $\mathcal{A}$ containing R but it may be existential positive definable in $\mathcal{A}$. Assume that the preconditions of the example hold and that CSP$(\mathcal{A})$ is in P. We know that $\bar{R}$ has an existential positive definition in $\mathcal{A}$ for every $R \in \mathcal{A}$. Let $\bar{\mathcal{A}} = \{\bar{R} \mid R \in \mathcal{A}\}$ and consider the problem CSP$_{\leq}(\mathcal{A} \cup \bar{\mathcal{A}})$. The forthcoming Theorem 15 is applicable so this problem is in FPT parameterized by $\#\mathsf{ac}$. Now, the corresponding model checking problem is to decide if $\mathcal{A} \models \phi$ where ϕ is an $\mathcal{A}$-sentence constructed using the operators $\forall$ and $\vee$ and where we are additionally allowed to use negated relations $\neg R(x_1, \ldots, x_m)$. It follows that this problem is in FPT parameterized by the number of negated relations.

4 General Tools for Alien Constraints

We analyze the complexity of CSP$_{\leq k}(\mathcal{A} \cup \mathcal{B})$, starting in Section 4.1 by investigating which of the classic algebraic tools are applicable to the alien constraint setting, and continuing in Section 4.2 by presenting a general FPT result. We will use these observations for proving various results but also for obtaining a better understanding of alien constraints.

4.1 Alien Constraints and Algebra

First, we have a straightforward generalization of Theorem 1 in the alien constraint setting.

▶ **Theorem 5** ($\star$). *Let $\mathcal{A}$ and $\mathcal{B}$ be two structures with disjoint signatures. There exists a polynomial time many-one reduction f from CSP$_{\leq}(\mathcal{A}^* \cup \mathcal{B}^*)$ to CSP$_{\leq}(\mathcal{A} \cup \mathcal{B})$ for any finite $\mathcal{A}^* \subseteq \langle \mathcal{A} \rangle$ and $\mathcal{B}^* \subseteq \langle \mathcal{A} \cup \mathcal{B} \rangle$. If $I = (V, C, k)$ is an instance of CSP$_{\leq}(\mathcal{A}^* \cup \mathcal{B}^*)$ and $f(I) = (V', C', k')$, then k' only depends on k, $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{B}^*$, so f is an fpt-reduction.*

This claim is, naturally, in general not true for $\mathrm{CSP}_{\leq k}(\mathcal{A}^* \cup \mathcal{B})$ for finite $\mathcal{A}^* \subseteq \langle \mathcal{A} \cup \mathcal{B} \rangle$. The idea underlying Theorem 5 can be used in many different ways and we give one example.

▶ **Proposition 6.** *If $\mathcal{A}, \mathcal{B}$ are structures and $R \in \langle \mathcal{A} \cup \mathcal{B} \rangle_{\leq 1}$, then $CSP_{\leq k}(\mathcal{A} \cup (\mathcal{B} \cup \{R\}))$ is polynomial-time reducible to $CSP_{\leq k}(\mathcal{A} \cup \mathcal{B})$.*

We proceed by relating $\mathrm{CSP}_{\leq k}(\mathcal{A} \cup \mathcal{B})$ to the important idea of reducing to a core (recall Section 2). Recall that $\mathcal{A}^c$ denotes the (unique up to isomorphism) core of a finite-domain structure $\mathcal{A}$. For two structures $\mathcal{A} \cup \mathcal{B}$ we similarly write $(\mathcal{A} \cup \mathcal{B})^c$ for the core. Specifically, if $e \in \mathrm{End}(\mathcal{A} \cup \mathcal{B})$ has minimal range, then the core consists of $\{e(R) \mid R \in \mathcal{A}\} \cup \{e(R) \mid R \in \mathcal{B}\}$ of the same signature as $\mathcal{A}$ and $\mathcal{B}$, and the problem $\mathrm{CSP}_{\leq}((\mathcal{A} \cup \mathcal{B})^c)$ is thus well-defined.

▶ **Theorem 7** ($\star$)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be two structures over a finite universe A. Then $CSP_{\leq}(\mathcal{A} \cup \mathcal{B})$ and $CSP_{\leq}((\mathcal{A} \cup \mathcal{B})^c)$ are interreducible under both polynomial-time and fpt reductions.*

In general, it is *not* possible to reduce from $\mathrm{CSP}_{\leq k}(\mathcal{A} \cup \mathcal{B})$ to $\mathrm{CSP}_{\leq k}(\mathcal{A}^c \cup \mathcal{B})$ or from $\mathrm{CSP}_{\leq k}(\mathcal{A} \cup \mathcal{B})$ to $\mathrm{CSP}_{\leq k}(\mathcal{A} \cup \mathcal{B}^c)$. This can be seen as follows. Consider the Boolean relation $R(x_1, x_2, x_3) \equiv x_1 = x_2 \vee x_2 = x_3$, and let $\mathcal{A} = \{R\}$, $\mathcal{B} = \{\neq\}$. Then, $\mathrm{CSP}_{\leq 1}(\mathcal{A} \cup \mathcal{B})$ is NP-hard (see e.g. Exercise 3.24 in [13]) so $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is pNP-hard. However, $\mathcal{A}$ is 0-valid, so $\mathcal{A}^c = \{\{(0,0,0)\}\}$, implying that $\mathrm{CSP}_{\leq}(\mathcal{A}^c \cup \mathcal{B})$ is in P.

4.2 Fixed-Parameter Tractability

We present an algorithm in this section that underlies many of our fixed-parameter tractability results and it is based on a particular notion of definability. The *existential fragment* of first-order logic consists of formulas that only use the operations negation, conjunction, disjunction, and existential quantification, while the *existential positive* fragment additionally disallows negation. We emphasize that it is required that the equality relation is allowed in existential (positive) definitions. We can view existential positive in a different way that is easier to use in our algorithm. Let $\mathcal{A}$ be a structure with domain A and assume that $R \subseteq A^m$ is defined via a existential positive definition over $\mathcal{A}$, i.e., $R(x_1, \ldots, x_m) \equiv \exists y_1, \ldots, y_n \colon \phi(x_1, \ldots, x_m, y_1, \ldots, y_n)$ where ϕ is a quantifier-free existential positive $\mathcal{A}$-formula. Since ϕ can be written in disjunctive normal form without introducing negation or quantifiers, it follows that R is a finite union of relations in $\langle \mathcal{A} \rangle$.

▶ **Theorem 8.** *Assume the following.*

1. *$\mathcal{A}, \mathcal{B}$ are structures with the same domain A,*
2. *every relation in $\mathcal{B}$ is existential positive definable in $\mathcal{A}$, and*
3. *$CSP(\mathcal{A})$ is in P.*

Then $CSP_{\leq}(\mathcal{A} \cup \mathcal{B})$ is in FPT *parameterized by* #ac.

Proof. Assume $\mathcal{B} = \{A; B_1, \ldots, B_m\}$. Condition 2. implies that B_i, $i \in [m]$, is a finite union of relations $B_i = R_i^1 \cup \cdots \cup R_i^{c_i}$ where $R_i^1, \ldots, R_i^{c_i}$ are in $\langle \mathcal{A} \rangle$. Let the structure $\mathcal{A}^*$ contain the relations in $\mathcal{A} \cup \{R_i^j \mid i \in [m] \text{ and } j \in [c_i]\}$. Clearly, $\mathcal{A}^*$ has a finite signature and the problem $\mathrm{CSP}(\mathcal{A}^*)$ is in P by Theorem 1 since every relation in $\mathcal{A}^*$ is a member of $\langle \mathcal{A} \rangle$. Let $b = \max\{c_i \mid i \in [m]\}$.

Let $((V, C), k)$ denote an arbitrary instance of $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B}))$. The satisfiability of (V, C) can be checked via the following procedure. If C contains no $\mathcal{B}$-constraint, then check the satisfiability of (V, C) with the polynomial-time algorithm for $\mathrm{CSP}(\mathcal{A})$. Otherwise, pick one constraint $c = B_i(x_1, \ldots, x_q)$ with $B_i \in \mathcal{B}$ and check recursively the satisfiability of the following instances:

$$(V, (C \setminus \{c\}) \cup \{R_i^1(x_1, \ldots, x_q)\}), \ldots, (V, (C \setminus \{c\}) \cup \{R_i^{c_i}(x_1, \ldots, x_q)\}).$$

If at least one of the instances is satisfiable, then answer "yes" and otherwise "no". This is clearly a correct algorithm for $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$.

We continue with the complexity analysis. Note that the leaves in the computation tree produced by the algorithm are $\mathrm{CSP}(\mathcal{A}^*)$ instances and they are consequently solvable in polynomial time. The depth of the computation tree is at most k (since (V, C) contains at most k $\mathcal{B}$-constraints) and each node has at most b children. Thus, the problem can be solved in $b^k \cdot \mathrm{poly}(|I|)$ time. We conclude that $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is in FPT parameterized by #ac since b is a fixed constant that only depends on the structures $\mathcal{A}$ and $\mathcal{B}$. ◀

5 Finite-Domain Languages

This section is devoted to CSPs over finite domains. We begin in Section 5.1 by studying how the definability of constants affect the complexity of finite-domain CSPs with alien constraints, and we use this as a cornerstone for a parameterized FPT versus pNP dichotomy result for of $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$. We show a sharper result for Boolean structures in Section 5.2.

5.1 Parameterized Dichotomy

We begin with a simplifying result. For a finite set A, let $\mathcal{C}_A$ be the structure whose relations are the constants over A.

▶ **Lemma 9** ($\star$). *Let $\mathcal{A}$ be a structure over a domain A. For every $\mathcal{C} \subseteq \mathcal{C}_A$, CSP$(\mathcal{A} \cup \mathcal{C})$ is polynomial-time reducible to CSP$_{\leq |\mathcal{C}|}(\mathcal{A} \cup \mathcal{C})$.*

Lemma 9 together with the basic algebraic results from Section 4.1 allows us to prove the following result that combines a more easily formulated fixed-parameter result (compared to Theorem 8) with a powerful hardness result.

▶ **Theorem 10** ($\star$). *Let $\mathcal{A}, \mathcal{B}$ be structures with finite domain D. Assume that $\mathcal{A} \cup \mathcal{B}$ is a core. If CSP$(\mathcal{A} \cup \mathcal{C}_A)$ is in* P*, then CSP$_{\leq}(\mathcal{A} \cup \mathcal{B})$ is in* FPT *with parameter* #ac*. Otherwise, CSP$_{\leq p}(\mathcal{A} \cup \mathcal{B})$ is* NP*-hard for some p that only depends on the structures $\mathcal{A}$ and $\mathcal{B}$.*

Proof. We provide a short proof sketch. Using the dichotomy of finite domain CSPs [12, 24], we first assume $\mathrm{CSP}(\mathcal{A} \cup \mathcal{C}_D)$ is in P. One can prove that every tuple over D is pp-definable over $\mathcal{A} \cup \mathcal{C}_D$ and then that each relation in $\mathcal{B}$ is existential positive definable over $\mathcal{A} \cup \mathcal{C}_D$. We can now apply Theorem 8, and $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is in FPT.

For the NP-hard case, we assume $\mathrm{CSP}(\mathcal{A} \cup \mathcal{C}_D)$ is NP-hard and construct a polynomial-time reduction from $\mathrm{CSP}(\mathcal{A} \cup \mathcal{C}_D)$ to $\mathrm{CSP}_{\leq p}(\mathcal{A} \cup \mathcal{B})$. We use the endomorphisms of $\mathcal{A} \cup \mathcal{B}$ to construct a pp-definable relation E which allow us to simulate the constant relations, and a reduction to $\mathrm{CSP}_{\leq 1}(\mathcal{A} \cup \{E\})$ to establish the claim via Lemma 9 and Theorem 5. ◀

Theorem 10 has broad applicability. Let us, for instance, consider a structure $\mathcal{A}$ with finite domain A and containing a finite number of relations from $\mathrm{Inv}(f)$ where $f \colon A^m \to A$ is idempotent ($f : A^m \to D$ is *idempotent* if $f(a, \ldots, a) = a$ for all $a \in A$.) If $\mathrm{CSP}(\mathcal{A})$ is in P, then $\mathrm{CSP}(\mathcal{A} \cup \mathcal{C}_A)$ is in P since constant relations are invariant under f. Hence, $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is in FPT parameterized by #ac for *every* finite structure $\mathcal{B}$ with domain A by Theorem 10. Idempotent functions that give rise to polynomial-time solvable CSPs are fundamental and well-studied in the literature; see e.g. the survey by Barto et al. [1].

Via Theorem 7 we obtain the following parameterized complexity dichotomy separating problems in FPT from pNP-hard problems.

▶ **Corollary 11.** *Let $\mathcal{A}, \mathcal{B}$ be structures over the finite domain A. Then, CSP$_{\leq}(\mathcal{A} \cup \mathcal{B})$ is either in* FPT *or* pNP*-hard (in parameter* #ac*).*

Proof. Let $e \in \text{End}(\mathcal{A} \cup \mathcal{B})$ have minimal range and let $\mathcal{A}' = \{e(R) \mid R \in \mathcal{A}\}$ and $\mathcal{B}' = \{R \mid R \in \mathcal{B}\}$ be the two components of the core $(\mathcal{A} \cup \mathcal{B})^c$, and let $A' = \{e(a) \mid a \in A\}$ be the resulting domain. The problems CSP$_{\leq}(\mathcal{A} \cup \mathcal{B})$ and CSP$_{\leq}(\mathcal{A}' \cup \mathcal{B}')$ are fpt-interreducible by Theorem 7. The problem CSP$(\mathcal{A}' \cup \mathcal{C}_{A'})$ is either in P or is NP-hard by the CSP dichotomy theorem [12, 24]. In the first case, CSP$_{\leq}(\mathcal{A}' \cup \mathcal{B}')$ (and thus CSP$_{\leq}(\mathcal{A} \cup \mathcal{B})$) is in FPT with parameter #ac. Otherwise, CSP$_{\leq}(\mathcal{A}' \cup \mathcal{B}')$ is pNP-hard, and the fpt-reduction from CSP$_{\leq}(\mathcal{A}' \cup \mathcal{B}')$ to CSP$_{\leq}(\mathcal{A} \cup \mathcal{B})$ establishes pNP-hardness for the latter. ◀

Corollary 11 must be used with caution: it does not imply that CSP$_{\leq 1}(\mathcal{A} \cup \mathcal{B})$ is NP-hard and results such as Theorem 4 may not be applicable. This encourages the refinement of coarse complexity results based on Theorem 10. We use Boolean relations as an example of this in the next section.

5.2 Classification of Boolean Languages

We present a complexity classification of CSP$_{\leq}(\mathcal{A} \cup \mathcal{B})$ when $\mathcal{A}$ and $\mathcal{B}$ are Boolean structures (Theorem 14). We begin with two auxiliary results and we define relations $c_0 = \{(0)\}$ and $c_1 = \{(1)\}$.

▶ **Lemma 12** ($\star$)**.** *Let $\mathcal{A}$ be a Boolean structure where $c_0 \in \langle \mathcal{A} \rangle$. If an n-ary Boolean $R \neq \emptyset$ is not 0-valid then $c_1 \in \langle \mathcal{A} \cup \{R\} \rangle_{\leq 1}$.*

We say that a Boolean relation R is *invariant under complement* if it is invariant under the operation $\{0 \mapsto 1, 1 \mapsto 0\}$. This is equivalent to $(t_1, \ldots, t_k) \in R$ if and only if $(1 - t_1, \ldots, 1 - t_k) \in R$.

▶ **Lemma 13** ($\star$)**.** *Let $\mathcal{A}$ be a Boolean structure with finite signature. If $\mathcal{A}$ is invariant under complement, then CSP$(\mathcal{A} \cup \{c_0, c_1\})$ is polynomial-time reducible to CSP$_{\leq 1}(\mathcal{A} \cup \{\neq\})$.*

We are now ready for analysing the complexity of CSP$_{\leq}(\mathcal{A} \cup \mathcal{B})$ when $\mathcal{A}$ and $\mathcal{B}$ are Boolean structures. We use a simplifying concept: a *0/1*-pair (R_0, R_1) contains two Boolean relations where R_0 is 0-valid but not 1-valid and R_1 is 1-valid but not 0-valid.

▶ **Theorem 14.** *Let $\mathcal{A}$ and $\mathcal{B}$ be Boolean structures such that CSP$(\mathcal{A})$ is in* P *and CSP$(\mathcal{A} \cup \mathcal{B})$ is* NP*-hard. Then the following holds.*

1. *If $\mathcal{A}$ is Schaefer, then CSP$_{\leq}(\mathcal{A} \cup \mathcal{B})$ is in* FPT *with parameter* #ac*.*
2. *If (i) $\mathcal{A}$ is not Schaefer, (ii) $\mathcal{A}$ is both 0- and 1-valid, (iii) $\mathcal{B}$ contains a* 0/1*-pair, and (iv) $\mathcal{B}$ is 0- or 1-valid, then CSP$_{\leq 2}(\mathcal{A} \cup \mathcal{B})$ is* NP*-hard and CSP$_{\leq 1}(\mathcal{A} \cup \mathcal{B})$ is in* P*.*
3. *Otherwise, CSP$_{\leq 1}(\mathcal{A} \cup \mathcal{B})$ is* NP*-hard.*

Proof. Assume $\mathcal{A}$ is Schaefer and let $\mathcal{A}^+ = \mathcal{A} \cup \{c_0, c_1\}$. The structure $\mathcal{A}^+$ is clearly a core and $\mathcal{A}^+ \cup \mathcal{B}$ is a core, too. The problem CSP$(\mathcal{A}^+)$ is in P by Theorem 2 so Theorem 10 implies that CSP$_{\leq}(\mathcal{A}^+ \cup \mathcal{B})$ (and naturally CSP$_{\leq}(\mathcal{A} \cup \mathcal{B})$) is in FPT parameterized by #ac. Since CSP$(\mathcal{A})$ is in P, we know from Theorem 2 that $\mathcal{A}$ is 0-valid, 1-valid or Schaefer. We assume henceforth that $\mathcal{A}$ is 0-valid and not Schaefer; the other case is analogous. If $\mathcal{B}$ is 0-valid, then CSP$(\mathcal{A} \cup \mathcal{B})$ is trivially in P and this is ruled out by our initial assumptions. We assume henceforth that $\mathcal{B}$ is not 0-valid and consider two cases depending on whether c_0 is pp-definable in $\mathcal{A}$ or not.

Case 1. c_0 is pp-definable in $\mathcal{A}$. We know that $\mathrm{CSP}(\mathcal{A} \cup \{c_0, c_1\})$ is NP-hard by Theorem 2 since $\mathcal{A}$ is not Schaefer. We can thus assume that $\mathrm{CSP}(\mathcal{A} \cup \{c_1\})$ is NP-hard. Lemma 9 implies that $\mathrm{CSP}_{\leq 1}(\mathcal{A} \cup \{c_1\})$ is NP-hard. The relation c_1 is in $\langle \mathcal{A} \cup \mathcal{B} \rangle_{\mathcal{B} \leq 1}$ by Lemma 12 so we conclude that $\mathrm{CSP}_{\leq 1}(\mathcal{A} \cup \mathcal{B})$ is NP-hard.

Case 2. c_0 is not pp-definable in $\mathcal{A}$. This implies that every relation in $\mathcal{A}$ is simultaneously 0- and 1-valid. To see this, assume to the contrary that $\mathcal{A}$ contains a relation that is not 1-valid. Then, $x = 0 \Leftrightarrow R(x, \ldots, x)$ and c_0 is pp-definable in $\mathcal{A}$. This implies that $\mathcal{B}$ contains (a) a relation that is not invariant under any constant operation or (b) every relation is closed under a constant operation and $\mathcal{B}$ contains a 0/1-pair. Note that if (a) and (b) does not hold, then $\mathcal{B}$ is invariant under a constant operation and $\mathrm{CSP}(\mathcal{A} \cup \mathcal{B})$ is trivially in P.

Case 2(a). There is a a relation R in $\mathcal{B}$ that is not invariant under any constant operation, i.e. $(0, \ldots, 0) \notin R$ and $(1, \ldots, 1) \notin R$. The relation R has arity $a \geq 2$. Let t be the tuple in R that contains the maximal number b of 0:s. Clearly, $b < a$. We assume that the arguments are permuted so that t begins with b 0:s and continues with $a - b$ 1:s. Consider the pp-defintion

$$S(x, y) \equiv R(\underbrace{x, \ldots, x}_{b \text{ occ.}}, \underbrace{y, \ldots, y}_{a-b \text{ occ.}}).$$

There are two possibilities: either $S(x, y) \Leftrightarrow x = 0 \wedge y = 1$ or $S(x, y) \Leftrightarrow x \neq y$. In the first case we are done since $\mathrm{CSP}(\mathcal{A} \cup \{c_0, c_1\})$ is NP-hard (recall that $\mathcal{A}$ is not Schaefer) and $\mathrm{CSP}_{\leq 1}(\mathcal{A} \cup \mathcal{B})$ is easily seen to be NP-hard by Lemma 9. Let us consider the second case. If $\mathcal{A}$ is invariant under complement, then $\mathrm{CSP}_{\leq 1}(\mathcal{A} \cup \mathcal{B})$ is NP-hard by Lemma 13. If $\mathcal{A}$ is not invariant under complement, then we claim that c_0 and c_1 can be pp-defined with the aid of $\neq$. Arbitrarily choose a relation T in $\mathcal{A}$ that contains a tuple $t = (t_1, \ldots, t_a)$ such that $(1 - t_1, \ldots, 1 - t_a) \notin T$ – note that t cannot be a constant tuple since both $(0, \ldots, 0)$ and $(1, \ldots, 1)$ are in T. Assume that t contains b 0:s and that the arguments are permuted so that t begins with b 0:s followed by $a - b$ 1:s. Consider the pp-definition

$$U(x, y) \equiv x \neq y \wedge T(\underbrace{x, \ldots, x}_{b \text{ occ.}}, \underbrace{y, \ldots, y}_{a-b \text{ occ.}}).$$

The relation U contains the single tuple $(0, 1)$. We know that $\mathrm{CSP}(\mathcal{A} \cup \{c_0, c_1\})$ is NP-hard (recall that $\mathcal{A}$ is not Schaefer) and Lemma 9 implies that $\mathrm{CSP}_{\leq 2}(\mathcal{A} \cup \{c_0, c_1\})$ is NP-hard, too. It is now easy to see that $\mathrm{CSP}_{\leq 1}(\mathcal{A} \cup \mathcal{B})$ is NP-hard via the definition of U.

Case 2(b). Every relation in $\mathcal{B}$ is closed under at least one constant operation and $\mathcal{B}$ contains a 0/1-pair (R_0, R_1). Since $\mathcal{A}$ is both 0- and 1-valid, it follows that $\mathrm{CSP}_{\leq 1}(\mathcal{A} \cup \mathcal{B})$ is in P. The constant relations c_0 and c_1 are pp-definable in $\{R_0, R_1\}$ since $x = 0 \Leftrightarrow R_0(x, \ldots, x)$ and $x = 1 \Leftrightarrow R_1(x, \ldots, x)$. This implies with the aid of Lemma 9 that $\mathrm{CSP}_{\leq 2}(\mathcal{A} \cup \mathcal{B})$ is NP-hard since $\mathcal{A}$ is not Schaefer. ◀

Theorem 14 carries over to Boolean Redundant(·), Equiv(·) and Impl(·) by Lemma 3 combined with Theorem 4, so these problems are in P if and only if $\mathcal{A}$ is Schaefer (case 2. in Theorem 14 is not applicable when analysing these problems since it requires $|\mathcal{B}| \geq 2$). Otherwise, they are NP-complete under polynomial-time Turing reductions. The *meta-problem* for Boolean CSPs with alien constraints is decidable, i.e., there is an algorithm that decides for Boolean structures $\mathcal{A}, \mathcal{B}$ whether $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is in case 1., 2., or 3. of Theorem 14. This is obvious since we have polymorphism descriptions of the Schaefer languages.

6 Infinite-Domain Languages

We focus on infinite-domain CSPs in this section. We begin Section 6.1 by discussing certain problems when CSPs with alien constraints are generalized to infinite domains. Our conclusion is that restricting ourselves to ω-categorical structures is a viable first step: ω-categorical structures constitute a rich class of CSPs and we can generalize at least some of the machinery from Section 5 to this setting. We demonstrate this in Section 6.2 where we obtain a complete complexity classification for equality languages.

6.1 Orbits and Infinite-Domain CSPs

It is not straightforward to tranfer the results in Section 5 to the infinite-domain regime. First, let us consider Theorem 8. In contrast to finite domains, relations in $\mathcal{B}$ may not be finite unions of relations in $\langle\mathcal{A}\rangle$ or, equivalently, not being definable with an existential positive formula. Second, let us consider Theorem 10: the proof is based on structures expanded with symbols for each domain value and this leads to problematic structures with infinite signatures. The proof is also based on the assumption that CSPs are either polynomial-time solvable or NP-complete, and this is no longer true [5]. It is thus necessary to restrict our attention to some class of structures with sufficiently pleasant properties. A natural choice is ω-categorical structures that allows us to reformulate Theorem 8 as follows.

▶ **Theorem 15** ($\star$). *Assume the following.*

1. *$\mathcal{A}, \mathcal{B}$ are structures with the same countable (not necessarily infinite) domain A,*
2. *$\mathcal{A}$ and $\mathcal{B}$ are ω-categorical,*
3. *every relation in* $\mathrm{Orb}(\mathcal{B})$ *is existential primitive definable in $\langle\mathcal{A}\rangle$, and*
4. *CSP($\mathcal{A}$) is in* P

Then CSP$_{\leq}(\mathcal{A} \cup \mathcal{B})$ is in FPT *parameterized by* #ac.

▶ **Example 16.** Results related to Theorem 15 have been presented in the literature. Recall that RCC5 and RCC8 are spatial formalism with binary relations that are disjunctions of certain basic relations [20]. Li et al. [17] prove that if $\mathcal{A}$ is a polynomial-time solvable RCC5 or RCC8 constraint language containing all basic relations, then Redundant($\mathcal{A}$) is in P. This immediately follows from combining Theorem 4 and Theorem 15 since RCC5 and RCC8 can be represented by ω-categorical constraint languages [3, 10] and every RCC5/RCC8 relation is existential primitive definable in the structure of basic relations by definition. This result can be generalized to a much larger class of relations in the case of RCC5 since the orbits of k-tuples are pp-definable in the structure of basic relations [6, Proposition 35].

A general hardness result based on the principles behind Theorem 10 does not seem possible in the infinite-domain setting, even for ω-categorical structures. The hardness proof in Theorem 10 utilizes variables given fixed values and a direct generalization would lead to groups of variables that together form an orbit of an n-tuple. Such gadgets behave very differently from variables given fixed values: in particular, they do not admit a result similar to Lemma 9. Thus, hardness results needs to be constructed in other ways.

We know from Section 4.1 that $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ and $\mathrm{CSP}_{\leq}((\mathcal{A} \cup \mathcal{B})^c)$ are the same when $\mathcal{A}$ and $\mathcal{B}$ has the same finite domain. We now consider a generalisation of cores to infinite domains from Bodirsky [2]: an ω-categorical structure $\mathcal{A}$ with countable domain is a *model-complete core* if every relation in $\mathrm{Orb}(\mathcal{A})$ is pp-definable in $\mathcal{A}$. There is an obvious infinite-domain analogue of Theorem 7: if $\mathcal{A}' \cup \mathcal{B}'$ is the model-complete core of $\mathcal{A} \cup \mathcal{B}$ (where $\mathcal{A}, \mathcal{B}$ are ω-categorical structures over a countable domain A), then $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ polynomial-time reduces to $\mathrm{CSP}_{\leq}(\mathcal{A}' \cup \mathcal{B}')$. Model-complete cores share many other properties with cores, too. With this said, it is interesting to understand model-complete cores in the context of $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$, simply because they are so well-studied and exhibit useful properties. We merely touch upon this subject by making an observation that we use in Section 6.2.

▶ **Lemma 17** ($\star$). *Let $\mathcal{A}$ and $\mathcal{B}$ denote ω-categorical structures with a countable domain A. Assume that $\mathcal{A}$ is a model-complete core and CSP($\mathcal{A}$) is in* P*. Then, $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is in* FPT *parameterized by* #ac *for every structure $\mathcal{B}$ such that $\mathrm{Orb}(\mathcal{B}) \subseteq \mathrm{Orb}(\mathcal{A})$.*

6.2 Classification of Equality Languages

We present a complexity classification of $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ for equality languages $\mathcal{A}$, $\mathcal{B}$. Essentially, there are two interesting cases: when $\mathcal{A}$ is Horn, and when $\mathcal{A}$ is 0-valid and not Horn. In the former case, $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is in FPT parameterized by #ac, while in the second case it is pNP-hard. It turns out that the ability to pp-define the arity-c disequality relation, where c depends only on $\mathcal{A}$, using at most k alien constraints, determines the complexity. A dichotomy for Redundant($\cdot$), Impl($\cdot$), and Equiv($\cdot$) follows: these problems are either in P or NP-hard under polynomial-time Turing reductions.

Recall that CSP($\mathcal{A}$) for a finite equality constraint language $\mathcal{A}$ is in P if $\mathcal{A}$ is 0-valid or preserved by a binary injective operation, and NP-hard otherwise, and that the automorphism group for equality languages is the symmetric group Σ on $\mathbb{N}$, i.e. the set of permutations on $\mathbb{N}$. It is easy to see that an orbit of a k-tuple $(a_1, \ldots, a_k)$ is pp-definable in $\{=, \neq\}$. For instance, the orbit of $(0, 0, 1, 2)$ is defined by $O(x_1, x_2, x_3, x_4) \equiv x_1 = x_2 \wedge x_2 \neq x_3 \wedge x_2 \neq x_4 \wedge x_3 \neq x_4$. Observe that $\neq$ is invariant under every binary injective operation, so if $\mathcal{A}$ is Horn, then $\neq \in \langle \mathcal{A} \rangle$ and every orbit of n-tuples under Σ is pp-definable in $\mathcal{A}$. Thus, $\mathcal{A}$ is a model-complete core as pointed out in Section 6.1. Lemma 17 now implies the following.

▶ **Corollary 18.** *Let $\mathcal{A}$ and $\mathcal{B}$ be equality languages. If $\mathcal{A}$ is Horn, then $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is in* FPT *parameterized by* #ac.

Thus, we need to classify the complexity of $\mathrm{CSP}_{\leq k}(\mathcal{A} \cup \mathcal{B})$ for every k, where $\mathcal{A}$ is 0-valid and not Horn, and $\mathcal{B}$ is not 0-valid. We will rely on results about the complexity of singleton expansions of equality constraint languages. Let $\mathcal{A}$ be a constraint language over the domain $\mathbb{N}$. By $\mathcal{A}^+_c$ we denote the expansion of $\mathcal{A}$ with c singleton relations, i.e. $\mathcal{A}^+_c = \mathcal{A} \cup \{\{1\}, \ldots, \{c\}\}$. The complexity of $\mathrm{CSP}(\mathcal{A}^+_c)$ for equality constraint languages $\mathcal{A}$ and all constants c was classified by Osipov & Wahlström [19, Section 7], building on the detailed study of polymorphisms of equality constraint languages by Bodirsky et al. [4].

The connection between $\mathrm{CSP}_{\leq k}(\mathcal{A} \cup \mathcal{B})$ and $\mathrm{CSP}(\mathcal{A}^+_c)$ is the following. In one direction, we can augment every instance of CSP($\mathcal{A}$) with c fresh variables $z_1, \ldots, z_c$ and, assuming k is large enough and $\mathcal{B}$ is not 0-valid, use $\mathcal{B}$-constraints to ensure that $z_1, \ldots, z_c$ attain distinct values in every satisfying assignment. Given that $\mathcal{A}$ is invariant under every permutation of $\mathbb{N}$, we can now treat $z_1, \ldots, z_c$ as constants, e.g. as $1, \ldots, c$, and transfer hardness results from the singleton expansion to our problem. In the other direction, if the relation $\mathrm{NEQ}_{c+1} \notin \langle \mathcal{A} \cup \mathcal{B} \rangle_{\leq k}$, then every satisfiable instance of $\mathrm{CSP}_{\leq k}(\mathcal{A} \cup \mathcal{B})$ has a solution with range $[c]$, and $\mathcal{A}^+_c$ is tractable: indeed, a satisfiable instance without such a solution would be a pp-definition of $\mathrm{NEQ}_{c'}$ for some $c' > c$. These connections are formalized in Lemmas 23 and 24. We will leverage the following hardness result.

▶ **Lemma 19** (Follows from Theorem 54 in [19]). *Let $\mathcal{A}$ be a finite equality language. If $\mathcal{A}$ is not Horn, then $\mathrm{CSP}(\mathcal{A}^+_c)$ is* NP*-hard for some $c = c(\mathcal{A})$.*

Our main tool for studying singleton expansions are *retractions*.

▶ **Definition 20.** *Let $\mathcal{A}$ be an equality language. An operation $f\colon \mathbb{N} \to [c]$ is a* retraction of $\mathcal{A}$ to $[c]$ *if f is an endomorphism of $\mathcal{A}$ where $f(i) = i$ for all $i \in [c]$. If $\mathcal{A}$ admits a retraction f to $[c]$, then we say that $\mathcal{A}$* retracts to $[c]$, *and $\mathcal{A}_f$ is* a retract (of $\mathcal{A}$ to $[c]$)*.*

We obtain a useful characterization of retracts.

▶ **Lemma 21.** *Let $\mathcal{A}$ be an equality language and f be a retraction from $\mathcal{A}$ to $[c]$. Then $f(R) = R \cap [c]^{\mathrm{ar}(R)}$ for all $R \in \mathcal{A}$.*

Proof. First, observe that $f(R) \subseteq R \cap [c]^{\mathrm{ar}(R)}$: indeed, f is an endomorphism, so $f(R) \subseteq R$, and $f(R) \subseteq [c]^{\mathrm{ar}(R)}$ because the range of f is $[c]$. Moreover, we have $R \cap [c]^{\mathrm{ar}(R)} \subseteq f(R)$ because f is constant on $[c]$, so it preserves every tuple in $[c]^{\mathrm{ar}(R)}$. ◀

The finite-domain language $\{R \cap [c]^{\mathrm{ar}(R)} : R \in \mathcal{A}\}$ is called a *c-slice of $\mathcal{A}$* in [19, Section 7]. Lemma 21 shows that a c-slice of $\mathcal{A}$ is the retract $\mathcal{A}_f$ under any retraction f from $\mathcal{A}$ to $[c]$. Note that the definition of the c-slice does not depend on f, so we can talk about *the retract of $\mathcal{A}$ to* $[c]$. We will use this fact implicitly when transferring results from Theorem 57 in [19].

▶ **Lemma 22** (Follows from Theorem 57 in [19]). *Let $\mathcal{A}$ be an equality language that is 0-valid and not Horn, and let c be a positive integer. Then exactly one of the following holds:*

- *$\mathcal{A}$ does not retract to $[c]$, and CSP($\mathcal{A}_c^+$) is* NP*-hard.*
- *$\mathcal{A}$ retracts to $[c]$, and CSP($\mathcal{A}_c^+$) is* NP*-hard for all $c \geq 2$.*
- *$\mathcal{A}$ retracts to $[c]$, and both CSP(Δ_c^+) for the retract Δ and CSP($\mathcal{A}_c^+$) are in* P*.*

Let $\mathrm{NEQ}_r = \{(t_1, \ldots, t_r) \in \mathbb{N}^r : |\{t_1, \ldots, t_r\}| = r\}$, i.e. the relation that contains every tuple of arity r with all entries distinct.

▶ **Lemma 23** ($\star$). *Let $\mathcal{A}$ and $\mathcal{B}$ be equality languages and $c \in \mathbb{Z}_+$. If $\mathrm{NEQ}_{c+1} \notin \langle \mathcal{A} \cup \mathcal{B} \rangle_{\leq k}$, then every satisfiable instance of CSP$_{\leq k}$($\mathcal{A} \cup \mathcal{B}$) has a solution whose range is in $[c]$.*

▶ **Lemma 24** ($\star$). *Let $\mathcal{A}$, $\mathcal{B}$ be two equality constraint languages, and let $c \in \mathbb{Z}_+$ be an integer. CSP($\mathcal{A}_c^+$) is polynomial-time reducible to CSP$_{\leq k}$($\mathcal{A} \cup \mathcal{B}$) whenever $\mathrm{NEQ}_c \in \langle \mathcal{A} \cup \mathcal{B} \rangle_{\leq k}$.*

We are ready to present the classification.

▶ **Theorem 25.** *Let $\mathcal{A}$ and $\mathcal{B}$ be equality languages such that CSP($\mathcal{A}$) is in* P *and CSP($\mathcal{A} \cup \mathcal{B}$) is* NP*-hard.*

1. *If $\mathcal{A}$ is Horn, CSP$_{\leq}$($\mathcal{A} \cup \mathcal{B}$) is in* FPT *parameterized by* #ac*.*
2. *If $\mathcal{A}$ is not Horn, CSP$_{\leq}$($\mathcal{A} \cup \mathcal{B}$) is* pNP*-hard parameterized by* #ac*. Moreover, there exists an integer $c = c(\mathcal{A})$ such that CSP$_{\leq k}$($\mathcal{A} \cup \mathcal{B}$) is in* P *whenever $\mathrm{NEQ}_c \notin \langle \mathcal{A} \cup \mathcal{B} \rangle_{\leq k}$, and is* NP*-hard otherwise.*

Proof. CSP($\mathcal{A}$) is in P so $\mathcal{A}$ is Horn or 0-valid. If $\mathcal{A}$ is Horn, then Corollary 18 applies, proving part 1 of the theorem. Suppose $\mathcal{A}$ is 0-valid and not Horn. By applying Lemma 19 to $\mathcal{A}$, we infer that there is a minimum positive integer c such that CSP($\mathcal{A}_c^+$) is NP-hard. Since $\mathcal{A}$ is 0-valid, we have $c \geq 2$. Using Lemma 24, we can reduce CSP($\mathcal{A}_c^+$) to CSP$_{\leq k}$($\mathcal{A} \cup \mathcal{B}$) in polynomial time whenever $\mathrm{NEQ}_c \in \langle \mathcal{A} \cup \mathcal{B} \rangle_{\mathcal{B} \leq k}$, proving that the latter problem is NP-hard. Observe that $\mathcal{B}$ is not 0-valid because CSP($\mathcal{A} \cup \mathcal{B}$) is NP-hard, so $\neq\, \in \langle \mathcal{B} \rangle$ and $\mathrm{NEQ}_c \in \langle \mathcal{A} \cup \mathcal{B} \rangle_{\mathcal{B} \leq k}$ for some finite $k \leq \binom{c}{2}$. This show the pNP-hardness result in part 2.

To complete the proof of part 2, it suffices to show that we can solve CSP$_{\leq k}$($\mathcal{A} \cup \mathcal{B}$) in polynomial time whenever $\mathrm{NEQ}_c \notin \langle \mathcal{A} \cup \mathcal{B} \rangle_{\mathcal{B} \leq k}$. To this end, observe that, by the choice of c, if $c' < c$, then CSP($\mathcal{A}_{c'}^+$) is in P. Then, by Lemma 22, $\mathcal{A}$ retracts to the finite domain $[c']$, and the retract Δ is such that CSP($\Delta_{c'}^+$) is in P. We will use the algorithm for CSP($\Delta_{c'}^+$) in our algorithm for CSP$_{\leq k}$($\mathcal{A} \cup \mathcal{B}$) that works for all k such that $\mathrm{NEQ}_c \notin \langle \mathcal{A} \cup \mathcal{B} \rangle_{\leq k}$.

Let I be an instance of $\mathrm{CSP}_{\leq k}(\mathcal{A} \cup \mathcal{B})$. Since $\mathrm{NEQ}_c \notin \langle \mathcal{A} \cup \mathcal{B} \rangle_{\mathcal{B} \leq k}$, Lemma 23 implies that I is satisfiable if and only if it admits a satisfying assignment with range $[c-1]$. Let X be the set of variables in I that occur in the scopes of the alien constraints. Note that $|X| \in O(k)$. Enumerate all assignments $\alpha : X \to [c-1]$, and check if it satisfies all $\mathcal{B}$-constraints in I. If not, reject it, otherwise remove the $\mathcal{B}$-constraints and add unary constraints $x = \alpha(x)$ for all $x \in X$ instead. This leads to an instance of $\mathrm{CSP}(\Delta^+_{c-1})$, which is solvable in polynomial time. If we obtain a satisfiable instance for some α, then accept I, and otherwise reject it. Correctness follows by Lemma 23 and the fact that the algorithm considers all assignments from X to $[c]$. We make $2^{O(k)}$ calls to the algorithm for $\mathrm{CSP}(\Delta^+_{c-1})$, where k is a fixed constant, and each call runs in polynomial time. This completes the proof. ◀

Theorem 14 implies that $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ is pNP-hard if and only if $\mathrm{CSP}_{\leq k}(\mathcal{A} \cup \mathcal{B})$ is NP-hard for some k, and it is in FPT parameterized by #ac otherwise. Theorem 25 now implies a dichotomy for Redundant($\cdot$), Impl($\cdot$), and Equiv($\cdot$) over finite equality languages.

▶ **Theorem 26** (⋆). *Let $\mathcal{A}$ be a finite equality language. Then* Redundant($\mathcal{A}$), Impl($\mathcal{A}$), *and* Equiv($\mathcal{A}$) *are either in* P *or* NP*-hard (under polynomial-time Turing reductions).*

Algebraically characterizing the exact borderline between tractable and hard cases of the problem seems difficult. In particular, given a 0-valid non-Horn equality language $\mathcal{A}$, answering whether $\mathrm{CSP}_{\leq 1}(\mathcal{A} \cup \bar{\mathcal{A}})$ is in P, i.e. whether $\mathrm{NEQ}_c \in \langle \mathcal{A} \cup \bar{R} \rangle_{\leq 1}$ for some $R \in \mathcal{A}$ and large enough c, requires a deeper understanding of such languages. However, one can show that the answer to this, and even a more general question is decidable.

▶ **Proposition 27** (⋆). *There is an algorithm that takes two equality constraint languages $\mathcal{A}$ and $\mathcal{B}$ and outputs minimum $k \in \mathbb{N} \cup \{\infty\}$ such that $CSP_{\leq k}(\mathcal{A} \cup \mathcal{B})$ is* NP*-hard.*

7 Discussion

We have focused on structures with finite signatures in this paper. This is common in the CSP literature since relational structures with infinite signature cause vexatious representational issues. It may, though, be interesting to look at structures with infinite signatures, too. Zhuk [25] observes that the complexity of the following problem is open: given a system of linear equations mod 2 and a single linear equation mod 24, find a satisfying assignment over the domain $\{0, 1\}$. The equations have unbounded arity so this problem can be viewed as a $\mathrm{CSP}_{\leq 1}(\mathcal{A} \cup \mathcal{B})$ problem where $\mathcal{A}, \mathcal{B}$ have infinite signatures. This question is thus not directly answered by Theorem 14. Second, let us also remark that when considering $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$, we have assumed that both $\mathcal{A}$ and $\mathcal{B}$ are taken from some nice “superstructure”. For example, in the equality language case we assume that both structures are first-order reducts of $(\mathbb{N}; =)$. One could choose structures more freely and, for example, let $\mathcal{A}$ be an equality language and $\mathcal{B}$ a finite-domain language. This calls for modifications of the underlying theory since (for instance) the algorithm that Theorem 8 is based on breaks down.

For finite domains we obtained a *coarse* parameterized dichotomy for $\mathrm{CSP}_{\leq}(\mathcal{A} \cup \mathcal{B})$ separating FPT from pNP-hardness. Sharper results providing the exact borderline between P and NP-hardness for the pNP-hard cases are required for classifying implication, equivalence, and redundancy. Via Theorem 7 and Theorem 10 the interesting case is when $\mathrm{CSP}(\mathcal{A})$ is in P, $\mathcal{A} \cup \mathcal{B}$ is core but $\mathcal{A}$ is not core. This question may be of independent algebraic interest and could be useful for other problems where the core property is not as straightforward as in the CSP case. For example, in *surjective* CSP we require the solution to be surjective, and this problem is generally hardest to analyze when the template is not a core [8].

Any complexity classification of the first-order reducts of a structure includes by necessity a classification of equality CSPs. Thus, our equality language classification lay the foundation for studying first-order reducts of more expressive structures. A natural step is to study *temporal languages*, i.e. first-order reducts of $(\mathbb{Q}; <)$. Our classification of equality constraint languages relies on the work in [4] via [19], who studied the clones of polymorphisms of equality constraint languages in more detail. One important result, due to Haddad & Rosenberg [15], is that after excluding several easy cases, every equality constraint language we end up with is only closed under operations with range $[c]$ for some constant c. Then, pp-defining the relation NEQ_{c+1} brings us into pNP-hard territory. Similar characterizations of the polymorphisms for reducts of other infinite structures, e.g. $(\mathbb{Q}; <)$, would imply corresponding pNP-hardness results, and this appear to be a manageable way forward.

References

1 Libor Barto, Andrei A. Krokhin, and Ross Willard. Polymorphisms, and how to use them. In *The Constraint Satisfaction Problem: Complexity and Approximability*, volume 7 of *Dagstuhl Follow-Ups*, pages 1–44. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.

2 Manuel Bodirsky. *Complexity of Infinite-Domain Constraint Satisfaction*. Cambridge University Press, 2021.

3 Manuel Bodirsky and Hubie Chen. Qualitative temporal and spatial reasoning revisited. *Journal of Logic and Computation*, 19(6):1359–1383, 2009.

4 Manuel Bodirsky, Hubie Chen, and Michael Pinsker. The reducts of equality up to primitive positive interdefinability. *The Journal of Symbolic Logic*, 75(4):1249–1292, 2010. doi:10.2178/jsl/1286198146.

5 Manuel Bodirsky and Martin Grohe. Non-dichotomies in constraint satisfaction complexity. In *Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP-2008)*, pages 184–196, 2008.

6 Manuel Bodirsky and Peter Jonsson. A model-theoretic view on qualitative constraint reasoning. *Journal of Artificial Intelligence Research*, 58:339–385, 2017.

7 Manuel Bodirsky and Jan Kára. The complexity of equality constraint languages. *Theory of Computing Systems*, 43(2):136–158, 2008.

8 Manuel Bodirsky, Jan Kára, and Barnaby Martin. The complexity of surjective homomorphism problems - a survey. *Discrete Applied Mathematics*, 160(12):1680–1690, 2012.

9 Manuel Bodirsky and Jaroslav Nešetřil. Constraint satisfaction with countable homogeneous templates. In *Proc. 17th International Workshop on Computer Science Logic (CSL-2003)*, pages 44–57, 2003.

10 Manuel Bodirsky and Stefan Wölfl. RCC8 is polynomial on networks of bounded treewidth. In *Proc. 22nd International Joint Conference on Artificial Intelligence (IJCAI-2011)*, pages 756–761, 2011.

11 Elmar Böhler, Edith Hemaspaandra, Steffen Reith, and Heribert Vollmer. Equivalence and isomorphism for Boolean constraint satisfaction. In *Proc. 16th International Workshop on Computer Science Logic (CSL-2002)*, pages 412–426, 2002.

12 Andrei A. Bulatov. A dichotomy theorem for nonuniform CSPs. In *Proc. 58th IEEE Annual Symposium on Foundations of Computer Science (FOCS-2017)*, pages 319–330, 2017.

13 Hubie Chen. A rendezvous of logic, complexity, and algebra. *ACM SIGACT News*, 37(4):85–114, 2006.

14 David A. Cohen, Peter Jeavons, Peter Jonsson, and Manolis Koubarakis. Building tractable disjunctive constraints. *Journal of the ACM*, 47(5):826–853, 2000.

15 Lucien Haddad and Ivo G. Rosenberg. Finite clones containing all permutations. *Canadian Journal of Mathematics*, 46(5):951–970, 1994.

16 Peter G. Jeavons. On the algebraic structure of combinatorial problems. *Theoretical Computer Science*, 200:185–204, 1998.

17 Sanjiang Li, Zhiguo Long, Weiming Liu, Matt Duckham, and Alan Both. On redundant topological constraints. *Artificial Intelligence*, 225:51–76, 2015.

18 Florent R. Madelaine and Barnaby Martin. On the complexity of the model checking problem. *SIAM Journal on Computing*, 47(3):769–797, 2018.

19 George Osipov and Magnus Wahlström. Parameterized complexity of equality MinCSP. *arXiv preprint arXiv:2305.11131*, 2023. This is the report version of a paper that appears in Proc. 31st Annual European Symposium on Algorithms (ESA-2023), pp. 86:1-86:17.

20 David A. Randell, Zhan Cui, and Anthony G. Cohn. A spatial logic based on regions and connection. In *Proc. 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR-1992)*, pages 165–176, 1992.

21 Thomas J. Schaefer. The complexity of satisfiability problems. In *Proc. 10th Annual ACM Symposium on Theory of Computing (STOC-1978)*, pages 216–226, 1978.

22 Henning Schnoor and Ilka Schnoor. Partial polymorphisms and constraint satisfaction problems. In *Complexity of Constraints - An Overview of Current Research Themes [Result of a Dagstuhl Seminar]*, pages 229–254. Springer, 2008.

23 Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proc. 14th Annual ACM Symposium on Theory of Computing (STOC-1982)*, pages 137–146, 1982.

24 Dmitriy Zhuk. A proof of the CSP dichotomy conjecture. *Journal of the ACM*, 67(5):30:1–30:78, 2020.

25 Dmitriy Zhuk. Constraint satisfaction problem: what makes the problem easy. In *Proc. International Conference of Mathematicians 2022 (ICM-2022)*, pages 1530–1553, 2022.

A New Optimization Model for Multiple-Control Toffoli Quantum Circuit Design

Jihye Jung[1]

H. Milton Stewart School of Ind. and Syst. Engineering, Georgia Institute of Technology, Atlanta, GA, USA

Kevin Dalmeijer

H. Milton Stewart School of Ind. and Syst. Engineering, Georgia Institute of Technology, Atlanta, GA, USA

Pascal Van Hentenryck

H. Milton Stewart School of Ind. and Syst. Engineering, Georgia Institute of Technology, Atlanta, GA, USA

Abstract

As quantum technology is advancing, the efficient design of quantum circuits has become an important area of research. This paper provides an introduction to the MCT quantum circuit design problem for reversible Boolean functions without assuming a prior background in quantum computing. While this is a well-studied problem, optimization models that minimize the true objective have only been explored recently. This paper introduces a new optimization model and symmetry-breaking constraints that improve solving time by up to two orders of magnitude compared to earlier work when a Constraint Programming solver is used. Experiments with up to seven qubits and using up to 15 quantum gates result in several new best-known circuits, obtained by any method, for well-known benchmarks. Finally, an extensive comparison with other approaches shows that optimization models may require more time but can provide superior circuits with optimality guarantees.

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

Keywords and phrases Constraint Programming, Quantum Circuit Design, Reversible Circuits

Digital Object Identifier 10.4230/LIPIcs.CP.2024.16

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2404.14384`

Funding This research was partly funded by the NSF AI Institute for Advances in Optimization (Award 2112533).

1 Introduction

As quantum technology is advancing, the efficient design of quantum circuits has become an important area of research. The primary challenge in quantum circuit design is to implement a *target function* using gates from a *preset gate library* to minimize the *circuit costs* according to some metric. This paper focuses on three choices that are often considered in the literature:

1. Target function: *Reversible Boolean function*, a key component that embeds the input data in most quantum algorithms.
2. Preset gate library: *Multiple-Control Toffoli (MCT) gate*, a typical high-level gate commonly used to represent reversible Boolean functions.
3. Circuit cost: *Quantum cost*, the number of low-level quantum gates required to realize the high-level gates in the circuit.

[1] Corresponding author

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 16; pp. 16:1–16:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

These concepts will be introduced in detail in Section 2, without assuming a prior background in quantum computing. The goal of this paper is to introduce a new optimization model to design quantum circuits within the setting defined above. For brevity, this problem will be referred to as the *MCT quantum circuit design problem*.

Literature Review. In similar settings to the one described above, early methods for quantum circuit design were developed based on intuitive observations and preconfigured circuit templates. Studies in this stage constructed a base library of small-scaled circuits to heuristically synthesize larger circuits for reversible Boolean functions [17, 9]. Post-synthesis algorithms, such as relocation algorithms [1, 22, 18, 19], were introduced to further improve these circuits, although the improved results cannot guarantee optimality either.

Several papers have used different representations of reversible Boolean functions to develop efficient synthesis algorithms. Cycle representation was used to devise several decomposition-based approaches [24, 31]. In particular, reference [24] reports an average cost improvement of 20% for benchmark functions with up to 20 qubits. Other approaches have leveraged the Reed-Muller expansion to decompose reversible boolean functions into exclusive-OR terms of Boolean products. The Reed-Muller expansion and the corresponding decision diagrams have appeared in reference [11] and [15] to address functions with up to 30 and 15 qubits, respectively. Reference [15] demonstrates a cost improvement of approximately 35% compared to previous studies, using a time limit of 600 seconds. A comparative analysis of decision diagram approaches for Reed-Muller expansion is also proposed [27].

Another heuristic approach was proposed via a quantum multiple-valued decision diagram, an efficient representation for matrices, to handle both reversible and irreversible functions [32]. The authors demonstrate the high scalability of the algorithm, handling functions with states up to length 156. A* algorithm was applied to the problem with an approximate heuristic function deduced from observations on state transitions [6], while others used heuristics based on isomorphic subgraph matching [14] and window optimization [26]. Evolutionary algorithms such as genetic algorithms [4], adaptive genetic algorithms [25], genetic programming [2], tabu search [8], and particle swarm optimization [7] were also suggested to obtain near-optimal solutions. While these methods offer different trade-offs between computation time and circuit quality, they are all heuristic and do not provide optimality guarantees.

Meanwhile, exact synthesis methodologies have been proposed to obtain optimal quantum circuits. Reference [10] iteratively solves satisfiability problems to obtain a quantum circuit with the minimum number of gates. This exact approach handles benchmark functions with three up to six qubits within a maximum of 5,000 seconds of computing time. The method based on quantified Boolean formula satisfiability, a generalized version of Boolean satisfiability, is also proposed to handle the same problem [29]. The authors report results for functions with four up to six qubits using a 2,000-second time limit. The exact methods notably handle relatively small functions but find better solutions within this space. Both references [10] and [29] report results in terms of quantum cost, which is the total number of low-level quantum gates required to implement a sequence of high-level logical gates. However, they do not directly minimize this objective; instead, they minimize the number of high-level gates as a proxy.

An optimization model was introduced to directly minimize the quantum cost in reference [13]. The authors use a multi-commodity flow-based model that is solved with a Mixed Integer Programming (MIP) solver. The method is applied to functions with three to six qubits, and significant improvements in quantum cost between 18.8% and 68.6% are observed. This provides a strong motivation to optimize quantum costs directly. While good results are obtained for small functions, the method does not scale well beyond seven gates due to the exponential number of binary variables in the model.

Contributions. This paper introduces a new optimization model to minimize quantum cost directly. Compared to [13], the new model is easier to implement, requires exponentially fewer binary variables, and has a beneficial block-angular structure. Furthermore, the paper demonstrates the advantage of Constraint Programming (CP) in solving this model. The key contributions can be summarized as follows:

- The paper introduces a new optimization model and new symmetry-breaking constraints for MCT quantum circuit design.
- The new model allows both CP and MIP solvers to significantly improve solving time, with up to *two orders of magnitude* speedup when the CP solver is used.
- Experiments with up to seven qubits and using up to 15 quantum gates result in several new best-known circuits for well-known benchmarks.
- An extensive comparison with other approaches shows that optimization models may require more time but can provide superior circuits with guaranteed optimality.

The remainder of the paper is organized as follows. Section 2 presents the necessary terminology and provides the problem description. Section 3 introduces the new optimization model, while Section 4 introduces new symmetry-breaking constraints. The computational results are presented by Section 5, and Section 6 concludes the paper.

2 Terminology

As discussed in the introduction, this paper designs quantum circuits for *reversible Boolean functions* using *MCT gates* to minimize the *quantum costs* of the resulting circuit. The relevant definitions are introduced here without assuming a prior knowledge of quantum computing. Example 1 presents a running example that is used throughout this section.

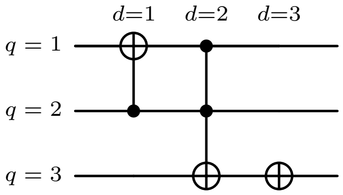
Basics of Quantum Computing. A state of a quantum system is represented by *qubits*, analogous to classical bits in classical computers. While bits assume values of 0 or 1 to define a single *basis state* (i.e., a binary vector), qubits may represent a *superposed state* (i.e., a complex vector) formed as a convex combination of the basis states. A *quantum gate* operates on qubits to transition the system to a new state based on the specification. Not every state transition can be realized by a single elementary gate, and multiple quantum gates may be combined into a *quantum circuit* to represent more complicated functions.

Reversible Boolean Function. A reversible Boolean function is a bijective function where inputs and outputs are provided as binary strings of fixed length. This function corresponds to a unique permutation and is often presented in the form of a truth table. It is noteworthy that reversible Boolean functions have been recognized as fundamental operators in quantum computing, thus explored extensively in prior research on quantum circuit synthesis [23].

Example 1a provides an example of a three-qubit reversible Boolean function. The specification defines a one-to-one mapping for each of the $2^3 = 8$ basis states. For instance, the input state (`qubit 1, qubit 2, qubit 3`)$= (1, 1, 0)$ is mapped to the output state (`qubit 1, qubit 2, qubit 3`) $= (0, 1, 1)$, or $110 \to 011$ for short. It is sufficient to only specify the function for the basis states: when superposed states are involved, they can simply be decomposed into a convex combination of basis states, after which the function can be applied to each basis state according to the specification.

Input	Output	Input	Output
000	001	100	101
001	000	101	100
010	110	110	011
011	111	111	010

(a) Truth Table (completely specified).



(b) Implementing Circuit.

Example 1 Specification and Implementing Circuit (interactive: `algassert.com/quirk`).

Multiple Control Toffoli (MCT) Circuit. *MCT circuits* consist of a sequence of *MCT gates*. Example 1b provides a circuit that meets the specification of Example 1a. It has three horizontal lines (one for each qubit q) and three MCT gates (one per column d). An MCT gate consists of one *target qubit* with the $\oplus$ symbol and zero or more *control qubits* with the $\bullet$ symbol. Control qubits do not have to be adjacent, and vertical lines connect the control qubits to the target qubit. For a given input, the circuit is read from left to right, and the MCT gates are applied one at a time. Transitions follow the following rule: *if all the control qubits are in state 1, then the target qubit is flipped.*

For example, consider the input 110 and start from the very left of the given circuit Example 1b. The top line has value 1, the middle line has value 1 and the bottom line has value 0. The first gate has one control qubit on line two. It follows that all control qubits are in state 1. As a result, the target qubit (`qubit 1`) is flipped, changing the state to 010 after the first gate. The second gate has two control qubits, but they are not all in state 1 (`qubit 1` is in state 0) so nothing happens. The third gate does not have any control bits, so the target qubit is flipped. This results in the output 011. That is, the total transition is 110 → 010 → 010 → 011, meeting the specification. It can be checked that the circuit meets the specifications for the other input states as well.

An important property of MCT circuits is that they are *reversible*, i.e., they perform the inverse operation when read from right to left [24]. Therefore, MCT circuits are a natural candidate to represent reversible Boolean functions. In fact, it is well-known that every reversible Boolean function can be represented in this way.

Quantum Costs. To implement an MCT circuit in practice, each MCT gate is decomposed into elementary quantum gates. The number of elementary quantum gates is a well-established proxy for the cost of the MCT circuit, known as the *quantum cost.* Table 1 summarizes the best-known quantum cost $f(c)$ for an MCT gate that uses a total of $c \geq 0$ control qubits [5, 16, 10]. Note that the costs change based on the number of *slack qubits* that are available and that are not used in the MCT gate otherwise. The cost of the circuit in Example 1 is $f(1) + f(2) + f(0) = 7$. Note that the costs in the table may go down in the future as better decompositions are found. It can also be seen that the quantum cost of an MCT gate tends to increase rapidly as more control qubits are added.

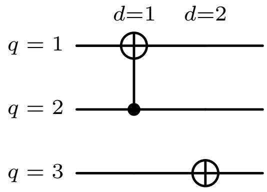
Remarks on Incomplete Specification. In Example 1, the truth table was completely specified, but this does not always have to be the case: depending on the application, there may be specific qubits that are used in the computation but for which the output is uninteresting (*don't care* qubits). However, every circuit *implementation* still represents a bijective function that assigns specific states to the don't cares, due to the reversible nature of quantum operators. Note that don't cares apply only to the outputs, whereas the inputs are completely specified in practice.

Table 1 Quantum Costs for MCT Gates (dots indicate the same cost as above; slack qubits are qubits that are available but not used in the MCT gate).

	Control qubits p							
Slack qubits	0	1	2	3	4	5	6	≤ 7
0	1	1	5	13	29	62	125	$2^{p+1} - 3$
1	·	·	·	·	·	52	80	·
2	·	·	·	·	26	·	·	·
3	·	·	·	·	·	38	·	·
≤ 4	·	·	·	·	·	·	50	·

Input	Output	Impl. 2b	Input	Output	Impl. 2b
000	00-	001	100	101	101
001	00-	000	101	100	100
010	11-	111	110	011	011
011	---	110	111	010	010

(a) Incompletely Specified Truth Table.



(b) Implementing Circuit.

Example 2 Specification and Implementing Circuit (interactive: `algassert.com/quirk`).

Example 2 turns the complete specification of Example 1 into an incomplete specification by replacing some of the output qubits by "-" (don't care) instead of 0 or 1. Note that the circuit in Example 1b is still valid, but with the additional freedom it might be possible to find a circuit with a cost lower than 7. Example 2b shows that a better circuit can indeed be found. The outputs of this implementation are added to Example 2a under "Impl. 2b". These outputs are different from the circuit in Example 1, but they both meet the incomplete specification. At a cost of only $f(1) + f(0) = 2$, the new circuit is a better implementation.

3 Optimization Model

This section presents a new optimization model to design minimum-cost MCT quantum circuits that meet a given specification. The new model is provided as Model (1) in Figure 3, and the different components will be introduced over the next several paragraphs. Constraints (1h)-(1i) are introduced as logical constraints first to clearly state their intent, after which a linear implementation is provided as just one possible implementation. An overview of the nomenclature is provided in Appendix A for convenience. While [13] and this paper both use network flows to model quantum state transitions, there are significant differences between the models that will be discussed at the end of this section.

Circuit Design. The design of a quantum circuit is modeled through variables that describe a quantum circuit diagram such as Example 1b or Example 2b. The set of qubits $Q = \{1, \ldots, n\}$ describes the rows, while the set of gates $D = \{1, \ldots, m\}$ describes the columns. Note that the maximum number of gates m is an input to the problem. Variables t_q^d and w_q^d are binary variables in Model (1) that indicate whether (q, d), contains a target or control qubit, respectively, for any $q \in Q$, $d \in D$. Constraints (1b) state that each spot contains a target qubit, a control qubit, or neither; but not both. Constraints (1c) enforce at most one target qubit per gate. Gates without a target qubit are forced to be empty through Constraints (1d). The variables are defined in Equation (1j).

$$
\begin{array}{llll}
\min & \displaystyle\sum_{d\in D}\sum_{j\in Q} f(j-1)y^d_j, & & \text{(1a)}\\
\text{s.t.} & t^d_q + w^d_q \leq 1 & \forall q\in Q, d\in D, & \text{(1b)}\\
& \displaystyle\sum_{q\in Q} t^d_q \leq 1 & \forall d\in D, & \text{(1c)}\\
& w^d_q \leq \displaystyle\sum_{r\in Q} t^d_r & \forall q\in Q, d\in D, & \text{(1d)}\\
& \displaystyle\sum_{j\in Q} j y^d_j = \sum_{q\in Q} t^d_q + \sum_{q\in Q} w^d_q & \forall d\in D, & \text{(1e)}\\
& \displaystyle\sum_{j\in Q} y^d_j \leq 1 & \forall d\in D, & \text{(1f)}\\
& \displaystyle\sum_{a\in\delta^+_k(v)} x^k_a - \sum_{a\in\delta^-_k(v)} x^k_a = \begin{cases} |\Omega^{in}_k| & \text{if } v=S\\ -|\Omega^{in}_k| & \text{if } v=T\\ 0 & \text{else}\end{cases} & \forall k\in K, v\in V, & \text{(1g)}\\
& \displaystyle\bigvee_{q^0\in Q^0_{\sigma(a)}} \left(w^{d(a)}_{q^0}=1\right) \bigvee \left(t^{d(a)}_{q(a)}=0\right) \implies x^k_a = 0 & \forall k\in K, a\in A^{flip}_k, & \text{(1h)}\\
& \displaystyle\bigwedge_{q^0\in Q^0_{\sigma(a)}} \left(w^{d(a)}_{q^0}=0\right) \bigwedge \left(\sum_{q\in Q} t^{d(a)}_q = 1\right) \implies x^k_a = 0 & \forall k\in K, a\in A^{keep}_k, & \text{(1i)}\\
& t^d_q, w^d_q, y^d_j \in \{0,1\} & \forall q,j\in Q, d\in D, & \text{(1j)}\\
& x^k_a \in \{0,1\} & \forall k\in K, a\in A_k. & \text{(1k)}
\end{array}
$$

Figure 3 Optimization Model.

Quantum Cost. An MCT gate with $c \geq 0$ control qubits incurs a quantum cost of $f(c)$, as defined in Table 1. Let y^d_j be a binary indicator that takes value one if gate $d \in D$ contains exactly $j \in Q$ target and control qubits, or zero otherwise. Objective (1a) then calculates the total quantum cost over all gates. Note that the input $q - 1$ subtracts the single target qubit, as $f(.)$ is defined in terms of control qubits only. When gate $d \in D$ is empty, all indicators y^d_j are zero (note $j \geq 1$) and there is no contribution to the objective. The y-variables are forced to take the correct values by Constraints (1e) and (1f). The former ensures that the indicators together represent the total number of target and control qubits, and the latter ensures that at most one indicator is active.

Intuition Flow Networks. The new model uses network flows to model the state transitions that are caused by the circuit. Before formally introducing this part of the model, this paragraph aims to give some intuition through Figure 4. The graph in this figure has vertices (σ, d) that indicate being in state $\sigma \in \Omega$ before gate $d \in D$ is applied. The arrows show the transitions when state 010 is provided as an input to Circuit 2b. That is, gate $d = 1$ carries out the transition $010 \to 110$ (vertex $(010, 1)$ to $(110, 2)$), and gate $d = 2$ carries out the transition $110 \to 111$ (vertex $(110, 2)$ to $(111, 3)$), for a total transition of $010 \to 111$.

Which state transitions are available depends on the design of the circuit. The dotted lines in Figure 4 show the state transitions that are allowed when the circuit design variables represent Circuit 2b. Because of the reversible nature of MCT gates, the dotted lines create

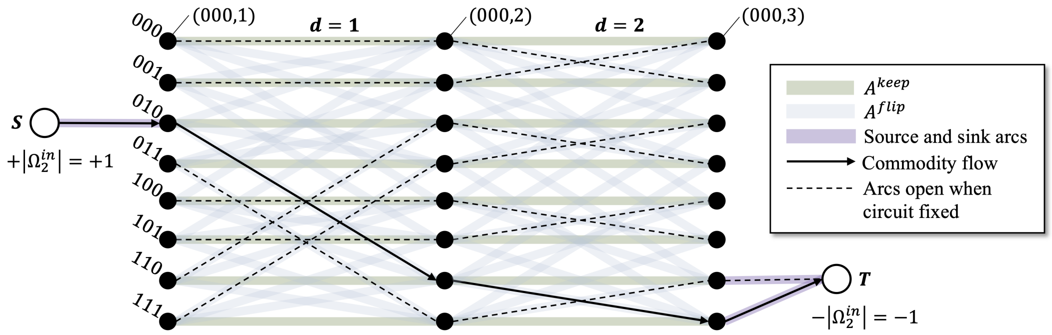


Figure 4 Flow Network Example for $k = 2$ corresponding to Example 2.

bijections between the states. As such, flow entering vertex $(\sigma, 1)$ is pushed to a unique vertex $(\sigma', m + 1)$, which represents the transition $\sigma \to \sigma'$ that is caused by the circuit. To make sure that the state transitions match the specification, a source S and a sink T are added. The source pushes the flow to a particular input state, while the sink only accepts flow from the correct output states. In the example, input state 010 has output specification 11− (Table 2a). As such, the source pushes flow to 010, and the sink only accepts flow from 110 and 111. If no flow from S to T is possible, then the circuit fails to meet the specification, and the current assignment of the circuit design variables is infeasible.

A separate flow network is introduced for each input state to ensure that the circuit meets all the specifications. The model is further improved by grouping input states with the same output specification into *commodities*. For a completely specified function, each commodity is associated with a single input state. In contrast, for an incomplete function, multiple input states can be grouped together based on the output specification. The source pushes a unit flow to each of the input states in the group. The flows remain separated due to the bijections, and are eventually collected by the sink. Note that this grouping is only possible when input states have the same output specification, and therefore the arcs connecting to the sink are the same.

Quantum States and Flow Commodities. Given the intuition, this paragraph formalizes the quantum states and the flow commodities. The set Ω of basis states is given by the 2^n binary vectors of length n. For example, $\Omega = \{000, 001, 010, 011, 100, 101, 110, 111\}$ in Example 2. States with the same (possibly incomplete) output specification will be modeled together and grouped into commodities $k \in K$. Each commodity is represented by the set $\Omega_k^{in} \subseteq \Omega$ of corresponding input states. Again, using Example 2: $\Omega_1^{in} = \{000, 001\}$ (output specification 00−), $\Omega_2^{in} = \{010\}$ (output specification 11−), etc. In addition, the set Ω_k^{out} defines the permitted output states for each commodity $k \in K$. In the example this yields $\Omega_1^{out} = \{000, 001\}$ (both match 00−), $\Omega_2^{out} = \{110, 111\}$ (both match 11−), etc. Note that the sets Ω_k^{in} partition Ω by definition, while the sets Ω_k^{out} may overlap and together cover Ω.

Flow Networks. Next, flow networks are defined that model the state transitions throughout the circuit. Figure 4 continues to provide a running example for this paragraph. A graph $G_k = (V, A_k)$ is defined for each commodity $k \in K$. The vertex set V consists of a source S, a sink T, and the vertices (σ, d) for $\sigma \in \Omega$, $d \in D \cup \{m + 1\}$. Each vertex (σ, d) represents being in state σ right after gate $d - 1$ or right before gate d, i.e., gate d is to be applied next. The arc set A_k of commodity $k \in K$ consists of four components:

- *Source arcs* that connect the source to the commodity input states: $(S, (\sigma, 1))\ \forall \sigma \in \Omega_k^{in}$.
- *Sink arcs* that connect permitted output states to the sink: $((\sigma, m+1), T)$, $\forall \sigma \in \Omega_k^{out}$.
- *Flip arcs* A_k^{flip} that represent the cases when gate $d \in D$ acts on state $\sigma \in \Omega$ by flipping bit $q \in Q$. Let $\sigma \oplus q$ denote state σ with bit q flipped. Then A_k^{flip} consists of arcs $((\sigma, d), (\sigma \oplus q, d+1))$, $\forall d \in D, \sigma \in \Omega, q \in Q$.
- *Keep arcs* A_k^{keep} that represent the cases when gate $d \in D$ keeps state $\sigma \in \Omega$ the same. That is, the keep arcs are given by $((\sigma, d), (\sigma, d+1))$, $\forall d \in D, \sigma \in \Omega$.

The state transitions for commodity $k \in K$ are represented by a network flow in G_k. Equation (1k) defines flow variables x_a^k that take on value one if arc $a \in A_k$ is used, and zero otherwise. The required flow is a flow from the source to the sink of size $|\Omega_k^{in}|$. This flow is distributed to all the input states Ω_k^{in} by the source arcs, after which the flip and keep arcs model the state transitions. The only way to reach the sink is through the sink arcs that start from one of the permitted output states Ω_k^{out}. The flow balance constraints (1g) enforce that the x_a^k variables represent such a flow. Here $\delta_k^+(v)$ and $\delta_k^-(v)$ denote the out-arcs and in-arcs of vertex $v \in V$, respectively.

It remains to connect the circuit design decisions to the network flows, i.e., ensure that arcs can only be used if they match the circuit specification. For convenience, the following shorthands are used for properties of flip and keep arcs $a \in A_k^{flip} \cup A_k^{keep}$: $d(a) \in D$ is the gate associated with arc a, $q(a) \in Q$ is the qubit that is flipped by arc a (flip arcs only), and $\sigma(a) \in \Omega$ is the state that the arc a transitions from. Furthermore, for a given state $\sigma \in \Omega$ it will be convenient to define Q_σ^0 as the set of qubits that are in state 0, e.g., $Q_{010}^0 = \{1, 3\}$. Constraints (1h) and (1i) eliminate the flow from all flip and keep arcs that do not match the circuit specification. This can be seen by considering the outgoing arcs of an arbitrary vertex (σ, d), $\sigma \in \Omega$, $d \in D$. That is, the set of one keep arc and n flip arcs that model the state transition due to gate d.

- *Case 1: Gate d flips some qubit $\bar{q} \in Q$.* Based on the transition rule (Section 2) this means that $\bar{q}$ is the target qubit ($t_{\bar{q}}^d = 1$) and all controls are on qubits with value 1 in state σ. Or alternatively, none of the controls are on qubits with value 0 in state σ ($w_{q^0}^d = 0\ \forall q^0 \in Q_\sigma^0$). It follows that the antecedent of (1i) holds and that the keep arc is excluded as expected. Flip arcs a are eliminated by (1h) as soon as they flip the wrong qubit, i.e., $q(a) \neq \bar{q}$, which implies $t_{q(a)}^d = 0$. The arc that flips $\bar{q}$ is the only flip arc that is *not* excluded by (1h), as the target is in the right place ($t_{q(a)}^d = 1$) and all controls are on the zero states ($w_{q^0}^d = 0$).
- *Case 2: Gate d keeps state σ the same.* Either there is no target qubit, in which case all flip arcs have $t_{q(a)}^d = 0$ and get eliminated by (1h) while the keep arc is unaffected by (1i). Or there is a target qubit $\bar{q}$, but at least one of the controls is on a zero state, i.e., $w_{q^0}^d = 1$ for some $q^0 \in Q_\sigma^0$. It follows again that all flip arcs are eliminated while the keep arc is unaffected.

It is concluded that Constraints (1h)-(1i) close the correct arcs to match the design of the circuit, which completes the model. It should also be noted that the flow variables may be relaxed to the continuous domain $x_a^k \in [0, 1]\ \forall k \in K, a \in A_k$. This follows from the fact that for fixed values of the t, w, and y-variables the remaining problem decomposes into $|K|$ independent minimum-cost flow problems, which are known to have the integrality property [3]. This means that the new model requires only $\mathcal{O}(nm)$ binaries.

Implementation of (1h)-(1i). There are multiple ways to implement Constraints (1h)-(1i), depending on the solver. This paper reformulates the implications as linear constraints, which are widely supported. The conversion is straightforward (e.g., see [30]), and results in the following inequalities that replace (1h)-(1i):

$$x_a^k \leq t_{q(a)}^{d(a)} \qquad \forall k \in K, a \in A_k^{flip}, \tag{2a}$$

$$x_a^k \leq 1 - w_{q^0}^{d(a)} \qquad \forall k \in K, a \in A_k^{flip}, q^0 \in Q_{\sigma(a)}^0, \tag{2b}$$

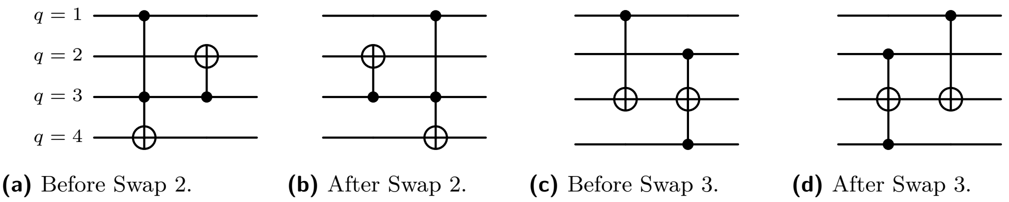
$$x_a^k \leq 1 - \sum_{q \in Q} t_q^{d(a)} + \sum_{q^0 \in Q_{\sigma(a)}^0} w_{q^0}^{d(a)} \quad \forall k \in K, a \in A_k^{keep}. \tag{2c}$$

Comparison to [13]. The new optimization model differs from [13] in a number of significant ways. First, [13] separately treat four cases for each state transition: empty gate (no flip), zero control qubits (flip), one or more control qubits with flip, one or more control qubits without flip. The new model captures all these cases in the same framework, significantly simplifying the formulation. Another important difference is in how the circuit is connected to opening and closing the flow arcs. [13] define a binary variable for each state $\sigma \in \Omega$ and gate $d \in D$ that identifies whether this state is modified by the gate, which introduces $\mathcal{O}(2^n m)$ binary variables. The new model provides a much more direct way to close arcs through Constraints (1h) and (1i). Compared to [13], this multiplies the number of constraints by a factor of $\mathcal{O}(n^2)$, *but no additional binary variables are necessary.* As a result, the new model requires only $\mathcal{O}(nm)$ binary variables. The number of flow variables remains at $\mathcal{O}(2^n nm|K|)$. However, as mentioned before, the new model decomposes into smaller independent minimum-cost flow problems for a fixed design. This implies that the formulation has a block-angular structure that may be exploited by decomposition methods in future work.

4 Symmetry-Breaking Constraints

It has been observed previously that the optimal quantum circuit design is not necessarily unique. The paper by [13] observes that empty gates do not affect the overall circuit, and constraints are added to force empty gates to appear at the end. The work by [12] presents multiple transformations that lead to new circuits with equivalent outputs (but not necessarily the same quantum cost). Inspired by these observations, this section defines three different swap operations – *Swap 1*, *Swap 2*, and *Swap 3* – that result in a different but equivalent circuit with the same cost. It will be proven that repeatedly applying these operations eventually results in a circuit that is *unswappable*, i.e., no further swaps of these types can be applied. This makes it possible to introduce symmetry-breaking constraints that limit the search to unswappable circuits, without loss of optimality. After all, every swappable quantum circuit is associated with an unswappable quantum circuit that has the same cost. The swaps used in this paper are introduced below, and future works may expand the list to eliminate additional symmetries.

- *Swap 1: Empty Gate.* If gate $d \in D$ is empty ($\sum_{q \in Q} t_q^d = 0$) and gate $d+1$ is not empty ($\sum_{q \in Q} t_q^{d+1} = 1$), then swap the two gates.
- *Swap 2: Different Target.* If the target qubit $q \in Q$ of gate $d \in D$ is at a higher line than the target qubit $r \in Q$ of gate $d+1$ ($q > r$, lower in the diagram), and the target qubits do not neighbor a control qubit ($w_q^{d+1} = 0$ and $w_r^d = 0$), then swap the two gates. It was observed by [12] that the target qubits do not affect the control qubits in either direction, and hence the gates can safely be swapped. Example 5a-5b provides a visualization.



(a) Before Swap 2. **(b)** After Swap 2. **(c)** Before Swap 3. **(d)** After Swap 3.

Example 5 Swap Operations.

- *Swap 3: Same Target.* If gate $d \in D$ and gate $d+1$ have the same target qubit $q \in Q$ ($t_q^d = 1$ and $t_q^{d+1} = 1$), and gate d has fewer control bits ($\sum_{r\in Q} w_r^d < \sum_{r \in Q} w_r^{d+1}$), then swap the two gates. Again, the target qubits do not affect the neighboring control qubits, which justifies the swap. A visualization is provided by Example 5c-5d.

▶ **Proposition 1.** *Any swappable circuit can be turned into an unswappable circuit by repeatedly applying Swap 1-3.*

Proof. See Appendix B. ◀

Constraints. Proposition 1 justifies symmetry-breaking constraints that enforce that the circuit is unswappable. The three swaps are translated into the following three classes of inequalities:

$$\sum_{q\in Q} t_q^d \geq \sum_{q \in Q} t_q^{d+1} \qquad \forall d \in D, \tag{3a}$$

$$t_q^d + t_r^{d+1} \leq 1 + w_q^{d+1} + w_r^d \qquad \forall d \in D, q, r \in Q, q > r, \tag{3b}$$

$$\sum_{r\in Q} w_r^d - \sum_{r \in Q} w_r^{d+1} \geq (n-1)(t_q^d + t_q^{d+1} - 2) \qquad \forall d \in D, q \in Q. \tag{3c}$$

Constraints (3a) prevent Swap 1 by forcing gate $d \in D$ to be non-empty when gate $d+1$ is non-empty. A similar constraint is used in [13]. Constraints (3b) model that if the target bits are set up correctly for Swap 2 (left-hand side of the equation equals two), then $w_q^{d+1} = 1$ or $w_r^d = 1$. This is necessary because $w_q^{d+1} = w_r^d = 0$ would allow Swap 2 to be applied. The constraint is automatically satisfied when the left-hand side is less than two. Finally, consider Constrains (3c). If the gates have targets on the same qubit ($t_q^d = t_q^{d+1} = 1$) then the constraint reduces to $\sum_{r\in Q} w_r^d \geq \sum_{r\in Q} w_r^{d+1}$ to prevent Swap 3. The term $n-1$ (the maximum number of control bits per gate) is sufficiently large to make the constraint inactive when the targets are not on the same qubit.

5 Computational Experiments

Computational experiments on well-known benchmarks are presented to demonstrate the performance of the new optimization model. The new model and the symmetry-breaking constraints are implemented in Python 3.11 and run on a Linux machine with dual Intel Xeon Gold 6226 CPUs (24 cores in total) on the PACE Phoenix cluster [20]. CP-SAT 9.8.3296 [21] is used as the CP solver with 24 workers (threads), and Gurobi 11.0.0 is used for the MIP approach. The instances are sourced from RevLib [28], a common benchmark for reversible and quantum circuit design. This paper selects boolean functions with up to seven qubits that have known circuit implementations in fewer than 100 gates. After removing easy three-qubit functions, a benchmark suite of 49 functions remains. A time limit of 3600

Table 2 Performance New Optimization Model compared to [13].

	Average Runtime (s)				Solved Instances		
	$m = 6$	$m = 7$	$m = 8$	Limit	$m = 6$	$m = 7$	$m = 8$
[13] (MIP)	6,614	21,126	29,895	36,000	36/38	20/38	7/38
New Model (MIP)	160	1252	2541	3,600	38/38	38/38	15/38
New Model (CP)	12	115	1193	3,600	38/38	38/38	28/38

Table 3 Performance New Optimization Model with CP on Large Instances.

	$m = 6$	$m = 7$	$m = 8$	$m = 9$	$m = 10$
Average Runtime (s)	14	111	1,101	2,140	2,502
Solved Instances	49/49	49/49	37/49	23/49	18/49
	$m = 11$	$m = 12$	$m = 13$	$m = 14$	$m = 15$
Average Runtime (s)	2,754	2,757	2,753	2,761	2,758
Solved Instances	13/49	13/49	13/49	13/49	13/49

seconds per instance is imposed in all experiments. The circuit design problem is considered to be *solved* if an optimal circuit implementation is found and proven to be optimal, or if it is proven that the problem is infeasible. If no circuit is found, or if optimality cannot be proven within the time limit, then the time limit is reported as the runtime.

Performance New Optimization Model. The performance of the new optimization model is first compared to the optimization model by [13]. To the best of our knowledge, [13] is the first MIP approach to the MCT quantum circuit design problem and therefore closests to the current work. Results are compared for the 38 reversible functions that overlap with the benchmark suite in this paper and experiments are conducted for $m \in \{6, 7, 8\}$ number of gates.

Table 2 demonstrates that the new optimization model completely outperforms previous work. Even accounting for the difference in hardware (6 cores vs. 24 cores), the new model is an order of magnitude faster when solved with Gurobi. When solved with CP-SAT, the new runtimes even improve by another magnitude. For $m = 6$ for example, CP is over 500x times faster than the runtime reported in [13]. It can also be seen that the new model solves significantly more instances. All benchmarks with $m = 7$ gates can now be solved, where only 20 out of 38 were solved previously. Further inspection of the results reveals that every instance that is solved by [13] is solved by the new model as well, regardless of whether MIP or CP was used. When CP fails to solve the problem (which only happens for $m = 8$), the best feasible solution is never worse than the best feasible solution obtained by [13]. The best performance is clearly obtained by the new model with CP, and this is the setting that is used for the remainder of the experiments.

Larger Instances. The improved performance of the new optimization model motivates experiments on the full benchmark suite for $m = 6$ up to $m = 15$ gates. This includes reversible functions such as `rd53` and `decod24-enable` that were not previously considered, and a number of gates that far exceeds the $m = 8$ gates in the experiments by [13].

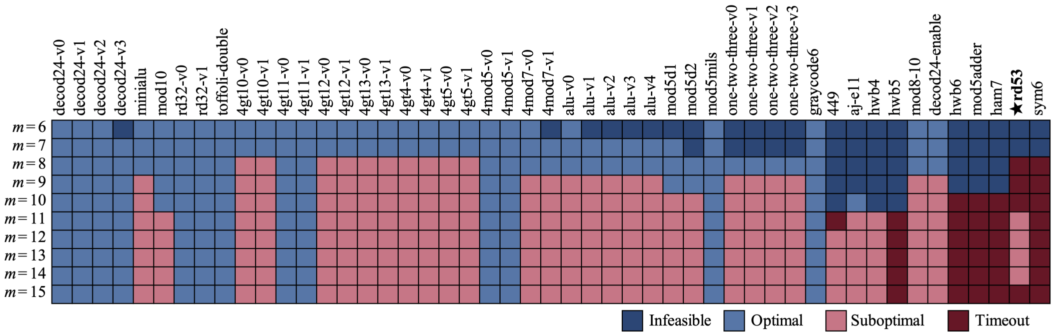


Figure 6 Solution Status of New Model with CP at 1-Hour Time Limit.

Table 3 shows that a big step was made in handling larger instances. All instances with up to $m = 7$ gates can now be solved in a matter of minutes on average, including the more complicated circuits. While most instances with $m = 8$ gates can still be solved within one hour, the average runtime starts to rise sharply at this point. It is clear that more work remains to be done to solve the largest instances, although 13 out of the 49 instances with $m = 15$ gates can already be solved.

Figure 6 details the final solution status (infeasible, optimal, suboptimal, or timeout) for each of the large instances. The newly added reversible functions are `449` up to `sym6` on the right side of the figure. It is interesting to observe that these functions require a relatively large number of gates. For example, `rd53` (marked with a $\star$) is proven infeasible for $m = 6$ and $m = 7$, and the solver was unable to find a feasible solution for $m \in \{8, 9, 10\}$, which are presumably infeasible. At $m = 11$, however, the solver fails to prove optimality but does find a feasible circuit. This circuit, shown in Appendix C, improves the state of the art with a quantum cost of 47, using only seven qubits. Compare this to [14], for example, who obtain a circuit with quantum cost 86 that requires 12 qubits. This demonstrates the benefit of using a model that can handle a larger number of gates, as good results may be obtained even when optimality cannot be proven.

Effect of Symmetry-Breaking Constraints. Figure 7 demonstrates the benefit of using symmetry-breaking constraints (3a)-(3c). Without symmetry-breaking constraints, all instances with $m = 6$ and $m = 7$ gates can still be solved, but the average solution times are 29% and 139% longer, respectively. For $m \geq 8$ gates, the difference in solvability becomes apparent. Out of the largest instances with $m = 15$ gates, only `graycode6` can be solved without breaking symmetries, while 13 instances can be solved when the constraints are included. It is not surprising that the symmetry-breaking constraints are more effective for longer circuits (i.e., large m values), as they are expected to have more symmetric solutions. More interestingly, adding the constraints outperforms the built-in symmetry detection in CP-SAT, which suggests that the symmetries observed in this paper are not obvious to detect automatically.

Comparative Analysis. A comparative analysis is provided to show how optimization-based methods fit in with other methods considered in the literature. Papers are selected that synthesize the entire circuit from scratch (as opposed to post-processing), that report quantum cost and computation time for every experiment, and for which the benchmark suite overlaps significantly with the current paper. This results in five studies that are summarized by

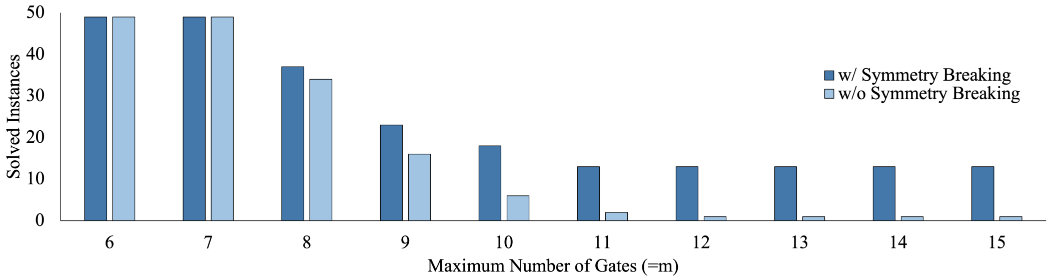


Figure 7 Comparison Symmetry-Breaking Constraints.

Table 4 Summary of Papers for Comparative Analysis.

Paper	Method	Objective	Type	Gate Lib.	Max Time
[15]	Reed-Muller + decision diagram	Gate count	Heuristic	MCT	600s
[14]	Subgraph matching + decision diagram	Qubit count	Heuristic	MCT up to two controls	<1s
[10]	Satisfiability problem	Gate count	Exact	MCT	5,000s
[29]	Quantified Boolean satisfiability problem	Gate count	Exact	MCT	2,000s
[13]	Optimization model + MIP solver	Quantum cost	Exact	MCT	36,000s
Current	Optimization model + CP solver	Quantum cost	Exact	MCT	3,600s

Table 4. The papers provide a mix of exact and heuristic methods that provide different trade-offs in terms of solution time and solution quality. Also note that while all papers report quantum cost, the methods themselves often use a different objective function as a proxy, such as minimizing the number of gates or the number of qubits.

Figure 8 includes 42 plots that show the performance of each method with a circle on the two-dimensional *time-quantum cost* plane. As for the results of this study, we select a solution with the smallest m value that brings the lowest quantum costs regardless of the proven optimality. Thus, circles with black borders indicate the current paper, and blue circles indicate that the solution was proven to be optimal in the selected m in terms of quantum cost, which is only optimized directly by [13] and this paper. While the current paper only considers circuits for a given number of qubits, [14] and [15] introduce additional qubits to obtain a feasible design. The number of qubits used is indicated by the relative size of the circle in Figure 8. A circuit is considered better if it has a lower quantum cost, and methods are preferred when they have a shorter solution time and introduce fewer ancilla qubits. That is, small circles in the lower-left are preferred.

Figure 8 shows that the current method outperforms the other methods in quantum cost (or in solution time if quantum cost is tied) in 29 out of 42 cases. These function names are marked by a blue box in the top right corner. Even when the new model does not reduce quantum cost or solution time, it can still provide a benefit of guaranteed optimality (e.g., `4gt11-v0`, `4gt11-v0`, `4mod5-v0`, `4mod5-v1`, `alu-v2`, `decod24-v0`, `decod24-v3`, `graycode6`, `mod5d1`, `mod5d2`) or fewer ancilla qubits (e.g., `449`). It should be noted, however, that the improved performance comes at the cost of a longer solution time: if only limited time is available, some of the other methods are better suited to provide good solutions quickly.

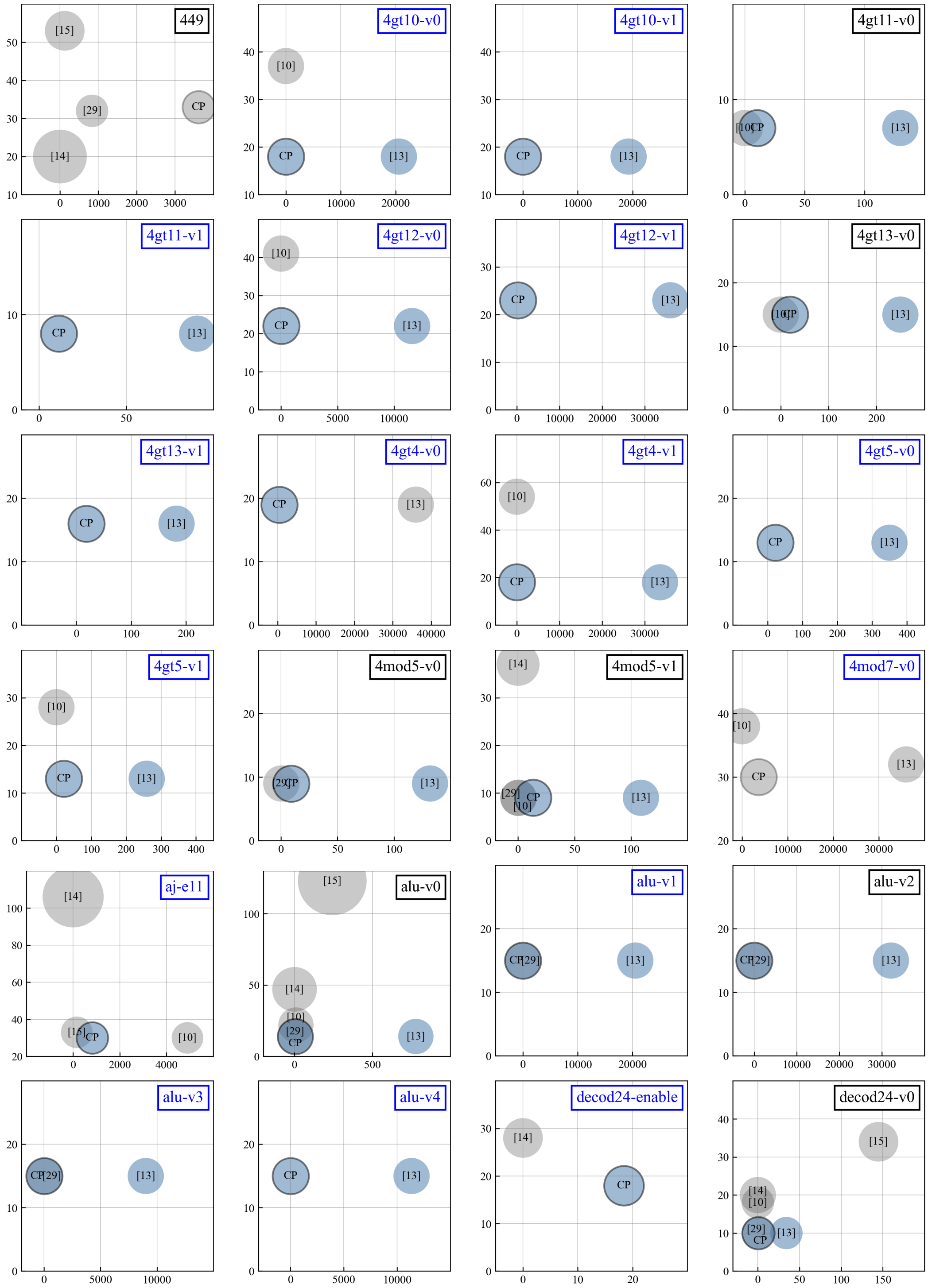


Figure 8 Comparison of Best Results with Previous Studies.

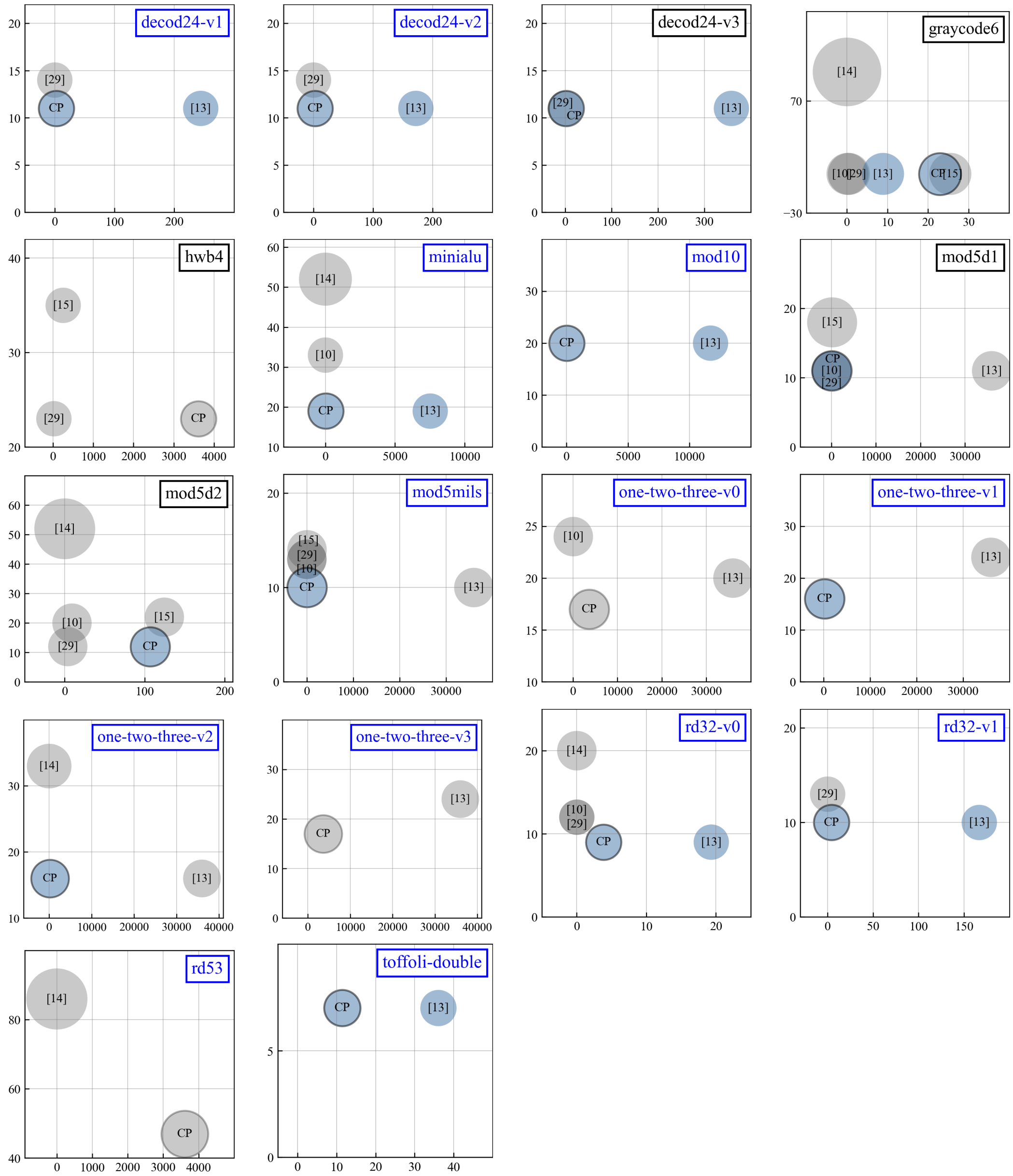


Figure 8 Comparison of Best Results with Previous Studies. *(cont.)*.

In particular, [29] provides high-quality solutions in a short amount of time, but improvements are still possible for some instances. For `decod24-v1` for example, [29] presents a circuit with six gates and a quantum cost of 14. This solution is found by first minimizing the number of gates, and then minimizing the quantum cost when the number of gates is fixed to six. The new optimization model, however, can directly solve the case with seven gates to find a circuit with quantum cost 11. This indicates that even for a small circuit of only four qubits, significant savings may still be obtained: a 21% cost reduction in this case. Overall, six new circuits have been found with new best-known quantum costs. These circuits are presented in Appendix C.

6 Conclusion

This paper introduced a new optimization model and symmetry-breaking constraints for the MCT quantum circuit design problem. The new model simplifies earlier work and drastically reduces the number of binary variables in the formulation. Computational experiments have shown that the new model allows both CP and MIP solvers to significantly improve solving time, with up to two orders of magnitude speedup when the CP solver is used. Experiments with larger instances of up to seven qubits and 15 gates have resulted in six new circuits with a lower cost than the previously best known. It was also shown that the symmetry-breaking constraints are very effective, especially in larger instances. Finally, a detailed comparison demonstrated that optimization models may require more time, but can provide superior circuits with guaranteed optimality.

There are several directions that may be explored in future work. While the new optimization model is effective, technical work remains to be done to scale to instances with more gates and more qubits. One opportunity would be to apply decomposition methods, as the structure of the proposed optimization model is such that the problem decomposes into independent minimum-cost flow problems when the binary variables are fixed. Another potential direction would be to extend the optimization model to different gate libraries or to directly optimize over elementary quantum gates instead of MCT gates.

References

1 Nabila Abdessaied, Mathias Soeken, Robert Wille, and Rolf Drechsler. Exact Template Matching Using Boolean Satisfiability. In *International Symposium on Multiple-Valued Logic*, pages 328–333, 2013. doi:10.1109/ismvl.2013.26.

2 Mustapha Y. Abubakar, Low Tang Jung, Nordin Zakaria, Ahmed Younes, and Abdel-Haleem Abdel-Aty. Reversible circuit synthesis by genetic programming using dynamic gate libraries. *Quantum Information Processing*, 16(6):1–24, 2017. doi:10.1007/s11128-017-1609-8.

3 Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, 1993.

4 Mohammad AlFailakawi, Imtiaz Ahmad, Laila AlTerkawi, and Suha Hamdan. Depth optimization for topological quantum circuits. *Quantum Information Processing*, 14(2):447–463, 2015. doi:10.1007/s11128-014-0867-y.

5 Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457–3467, 1995. doi:10.1103/physreva.52.3457.

6 Kamalika Datta, Gaurav Rathi, Indranil Sengupta, and Hafizur Rahaman. Synthesis of Reversible Circuits Using Heuristic Search Method. In *International Conference on VLSI Design*, pages 328–333, 2012. doi:10.1109/vlsid.2012.92.

7 Kamalika Datta, Indranil Sengupta, and Hafizur Rahaman. Particle Swarm Optimization Based Circuit Synthesis of Reversible Logic. In *International Symposium on Electronic System Design*, pages 226–230, 2012. doi:10.1109/ised.2012.33.

8 Alexandre A. A. de Almeida, Gerhard W. Dueck, and Alexandre C. R. da Silva. Reversible Circuit Optimization Based on Tabu Search. In *International Symposium on Multiple-Valued Logic*, pages 103–108, 2018. doi:10.1109/ismvl.2018.00026.

9 Oleg Golubitsky and Dmitri Maslov. A Study of Optimal 4-Bit Reversible Toffoli Circuits and Their Synthesis. *IEEE Transactions on Computers*, 61(9):1341–1353, 2011. doi:10.1109/tc.2011.144.

10 Daniel Große, Robert Wille, Gerhard W. Dueck, and Rolf Drechsler. Exact Multiple-Control Toffoli Network Synthesis With SAT Techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(5):703–715, 2009. doi:10.1109/tcad.2009.2017215.

11 Pallav Gupta, Abhinav Agrawal, and Niraj K. Jha. An Algorithm for Synthesis of Reversible Logic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(11):2317–2330, 2006. doi:10.1109/tcad.2006.871622.

12 Kazuo Iwama, Yahiko Kambayashi, and Shigeru Yamashita. Transformation rules for designing CNOT-based quantum circuits. In *Design Automation Conference*. ACM Press, 2002. doi:10.1145/513918.514026.

13 Jihye Jung and In-Chan Choi. A multi-commodity network model for optimal quantum reversible circuit synthesis. *PLOS ONE*, 16(6):e0253140, 2021. doi:10.1371/journal.pone.0253140.

14 Mridul Krishna and Anupam Chattopadhyay. Efficient Reversible Logic Synthesis via Isomorphic Subgraph Matching. In *International Symposium on Multiple-Valued Logic*, pages 103–108, 2014. doi:10.1109/ismvl.2014.26.

15 Chia-Chun Lin and Niraj K. Jha. RMDDS: Reed-Muller decision diagram synthesis of reversible logic circuits. *ACM Journal on Emerging Technologies in Computing Systems*, 10(2):1–25, 2014. doi:10.1145/2564923.

16 Dmitri Maslov and Gerhard W. Dueck. Improved quantum cost for n-bit Toffoli gates. *Electronics Letters*, 39(25):1790, 2003. doi:10.1049/el:20031202.

17 Dmitri Maslov, Gerhard W. Dueck, and D. Michael Miller. Toffoli network synthesis with templates. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(6):807–817, 2005. doi:10.1109/tcad.2005.847911.

18 Dmitri Maslov, Gerhard W. Dueck, and D. Michael Miller. Techniques for the synthesis of reversible Toffoli networks. *ACM Transactions on Design Automation of Electronic Systems*, 12(4):42:1–42:28, 2007. doi:10.1145/1278349.1278355.

19 Dmitri Maslov, Gerhard W. Dueck, D. Michael Miller, and Camille Negrevergne. Quantum Circuit Simplification and Level Compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):436–444, 2008. doi:10.1109/tcad.2007.911334.

20 PACE. *Partnership for an Advanced Computing Environment (PACE)*, 2017. URL: http://www.pace.gatech.edu.

21 Laurent Perron and Vincent Furnon. Google OR-Tools v9.8. https://developers.google.com/optimization/, 2023.

22 Aditya K. Prasad, Vivek V. Shende, Igor L. Markov, John P. Hayes, and Ketan N. Patel. Data structures and algorithms for simplifying reversible circuits. *ACM Journal on Emerging Technologies in Computing Systems*, 2(4):277–293, 2006. doi:10.1145/1216396.1216399.

23 Mehdi Saeedi and Igor L. Markov. Synthesis and optimization of reversible circuits—a survey. *ACM Computing Surveys*, 45(2):1–34, 2013. doi:10.1145/2431211.2431220.

24 Mehdi Saeedi, Morteza Saheb Zamani, Mehdi Sedighi, and Zahra Sasanian. Reversible circuit synthesis using a cycle-based approach. *ACM Journal on Emerging Technologies in Computing Systems*, 6(4):1–26, 2010. doi:10.1145/1877745.1877747.

25 Trailokya Nath Sasamal, Ashutosh Kumar Singh, and Anand Mohan. Reversible Logic Circuit Synthesis and Optimization Using Adaptive Genetic Algorithm. *Procedia Computer Science*, 70:407–413, 2015. doi:10.1016/j.procs.2015.10.054.

26 Mathias Soeken, Robert Wille, Gerhard W. Dueck, and Rolf Drechsler. Window optimization of reversible and quantum circuits. In *IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 341–345, 2010. doi:10.1109/ddecs.2010.5491754.

27 Robert Wille and Rolf Drechsler. Effect of BDD Optimization on Synthesis of Reversible and Quantum Logic. *Electronic Notes in Theoretical Computer Science*, 253(6):57–70, 2010. doi:10.1016/j.entcs.2010.02.006.

28 Robert Wille, Daniel Große, Lisa Teuber, Gerhard W. Dueck, and Rolf Drechsler. RevLib: An Online Resource for Reversible Functions and Reversible Circuits. In *International Symposium on Multiple-Valued Logic*, pages 220–225, 2008. doi:10.1109/ismvl.2008.43.

29 Robert Wille, Hoang M. Le, Gerhard W. Dueck, and Daniel Große. Quantified synthesis of reversible logic. In *Conference on Design, automation and test in Europe*, pages 1015–1020, 2008. doi:10.1145/1403375.1403620.

30 H. Paul Williams. *Model Building in Mathematical Programming*. John Wiley & Sons, 2013.

31 Wei Zhu, Zhiqiang Li, Gaoman Zhang, Suhan Pan, and Wei Zhang. A Reversible Logical Circuit Synthesis Algorithm Based on Decomposition of Cycle Representations of Permutations. *International Journal of Theoretical Physics*, 57(8):2466–2474, 2018. doi:10.1007/s10773-018-3768-5.

32 Alwin Zulehner and Robert Wille. Skipping Embedding in the Design of Reversible Circuits. In *International Symposium on Multiple-Valued Logic*, pages 173–178, 2017. doi:10.1109/ismvl.2017.19.

A Nomenclature

Symbol	Definition
Circuit Design: (1b)-(1d), (1j)	
Q	$= \{1, \dots, n\}$ set of qubits.
D	$= \{1, \dots, m\}$ set of gates.
t_q^d	variable with value 1 if qubit $q \in Q$ is the target qubit of gate $d \in D$, and 0 otherwise.
w_q^d	variable with value 1 if qubit $q \in Q$ is a control qubit of gate $d \in D$, and 0 otherwise.
Quantum Cost: (1a), (1e)-(1f), (1j)	
$f(c)$	quantum cost of a single MCT gate with $c \geq 0$ control qubits.
y_j^d	variable with value 1 if gate $d \in D$ consists of a total of $j \in Q$ target and control qubits, zero otherwise.
Quantum States and Flow Commodities: (1g)-(1i), (1k)	
Ω	$= \{0_{(2)}, \dots, (2^n - 1)_{(2)}\}$ set of pure quantum states.
Q_σ^0	$= \{q \in Q : \sigma_q = 0\}$ set of qubits that are zero in state $\sigma \in \Omega$.
K	set of indices of the flow commodities; each commodity represents a set of input quantum states that have the same (possibly incomplete) output specification.
Ω_k^{in}	$\subseteq \Omega$ set of input quantum states that represent commodity $k \in K$; together the sets Ω_k^{in} $\forall k \in K$ provide a partition of Ω.
Ω_k^{out}	$\subseteq \Omega$ set of quantum states that meet the (possibly incomplete) output specification associated with commodity $k \in K$; the sets Ω_k^{out} may overlap, and together cover Ω.
Flow Networks: (1g)-(1i), (1k)	
V	set of vertices in each flow network; consists of source S, sink T, and nodes (σ, d) $\forall \sigma \in \Omega, d \in D \cup \{m+1\}$.
A_k	set of arcs in the flow network of commodity $k \in K$.
A_k^{flip}	$\subset A_k$ set of arcs for commodity $k \in K$ that represent a transition that flips a qubit.
A_k^{keep}	$\subset A_k$ set of arcs for $k \in K$ that represent a transition that keeps the state the same.
x_a^k	variable with value 1 if commodity $k \in K$ uses arc $a \in A_k$, and 0 otherwise.
$\delta_k^+(v)$	$\subseteq A_k$ set of arcs for $k \in K$ coming out of vertex $v \in V$.
$\delta_k^-(v)$	$\subseteq A_k$ set of arcs for $k \in K$ coming into vertex $v \in V$.
$d(a)$	$\in D$ shorthand for the gate associated with arc $a \in A_k^{flip} \cup A_k^{keep}$.
$q(a)$	$\in Q$ shorthand for the qubit that is flipped by arc $a \in A_k^{flip}$.
$\sigma(a)$	$\in \Omega$ shorthand for the state that arc $a \in A_k^{flip} \cup A_k^{keep}$ transitions from.

B Proof Proposition 1

▶ **Proposition 1.** *Any swappable circuit can be turned into an unswappable circuit by repeatedly applying Swap 1-3.*

Proof. The number of permutations of the gates is finite, so it is sufficient to prove that the swaps can be applied in a way that avoids cycling. First repeatedly apply Swap 1, which results in all empty gates moving to the end of the circuit. These empty gates are not affected by Swap 2 and 3, and hence will stay in place and Swap 1 will not be applicable again. Let τ be a vector of length m that contains the target qubit of each gate, or zero otherwise. Every Swap 2 strictly decreases τ in lexicographical order, which avoids cycling. For example, the swap in Example 5a-5b changes τ from $(4, 2)$ to $(2, 4)$. Whenever no Swaps 2 can be made, repeatedly applying Swap 3 has the effect of sorting groups with the same target qubit by the number of control qubits, and does not introduce cycles. Swap 3 does not affect the vector τ, so no cycles are introduced when returning to Swap 2 after Swap 3 is exhausted. Eventually, an unswappable circuit is obtained after finitely many steps. ◀

C New Best-Known Circuits

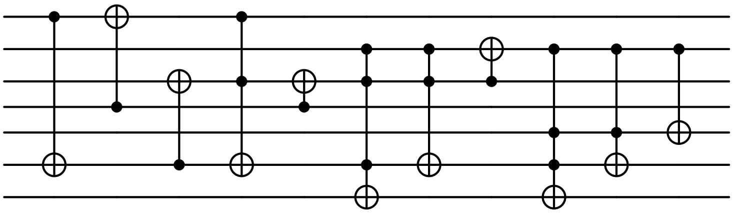


Figure 9 `rd53` for $m = 11$ (quantum cost: 47, optimality proven: no).

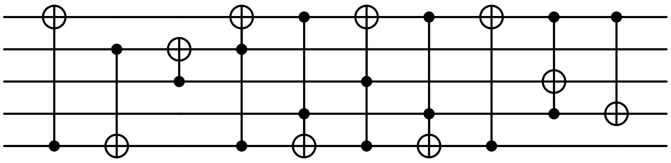


Figure 10 `4mod7-v0` for $m = 10$ (quantum cost: 30, optimality proven: no).

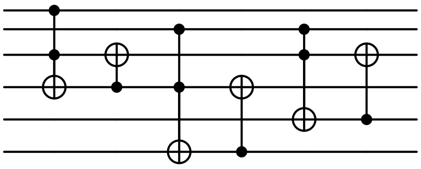


Figure 11 `decod24-enable` for $m = 6$ (quantum cost: 18, optimality proven: yes).

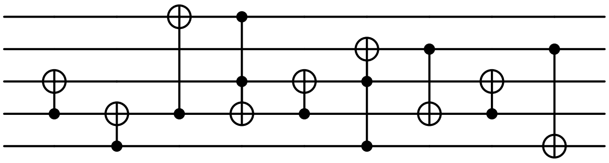


Figure 12 `one-two-three-v0` for $m = 9$ (quantum cost: 17, optimality proven: no).

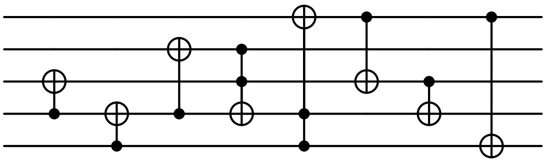


Figure 13 `one-two-three-v1` for $m = 8$ (quantum cost: 16, optimality proven: yes).

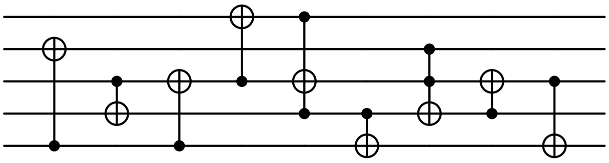


Figure 14 `one-two-three-v3` for $m = 9$ (quantum cost: 17, optimality proven: no).

Exponential Steepest Ascent from Valued Constraint Graphs of Pathwidth Four

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

Melle van Marle
Department of Mathematics, and Department of Information and Computing Sciences,
Utrecht University, The Netherlands

Abstract

We examine the complexity of maximising fitness via local search on valued constraint satisfaction problems (VCSPs). We consider two kinds of local ascents: (1) steepest ascents, where each step changes the domain that produces a maximal increase in fitness; and (2) ≺-ordered ascents, where – of the domains with available fitness increasing changes – each step changes the ≺-minimal domain. We provide a general padding argument to simulate any ordered ascent by a steepest ascent. We construct a VCSP that is a path of binary constraints between alternating 2-state and 3-state domains with exponentially long ordered ascents. We apply our padding argument to this VCSP to obtain a Boolean VCSP that has a constraint (hyper)graph of arity 5 and pathwidth 4 with exponential steepest ascents. This is an improvement on the previous best known construction for long steepest ascents, which had arity 8 and pathwidth 7.

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

Keywords and phrases valued constraint satisfaction problem, steepest ascent, local search, bounded treewidth, intractability

Digital Object Identifier 10.4230/LIPIcs.CP.2024.17

Acknowledgements AK would like to thank Dave Cohen and Peter Jeavons for helpful discussions. AK and MvM would also like to thank Daniel Dadush for helpful feedback and questions.

1 Introduction

Local search is often used in combinatorial optimisation. One of the most common methods for choosing which local modification to make is the steepest ascent algorithm, which at each step selects the highest-value option from the neighbours of the current state. Clearly such an algorithm could get trapped at local optima that might prevent it from reaching a higher optimum. Surprisingly, [9] showed that for problems that are hard for the complexity class of polynomial local search (PLS), even local optima can be intractable to find – regardless of what polynomial time algorithm is used for the search. As such, it is natural to ask: under what conditions could popular local search algorithms like steepest ascent be guaranteed to find even a local optimum in reasonable time? Or stated in term of intractability: for what problems does steepest ascent not find a local optimum quickly, taking instead an exponential number of steps before arriving at any local optimum.

Many combinatorial optimisation problems can be formulated as valued constraint satisfaction problems (VCSPs). Since weighted 2-SAT is PLS-complete [14] and a special case of binary Boolean VCSPs, it is believed to be intractable to find local optima in general VCSPs. It is also possible to create VCSPs where every ascent from some initial assignment is exponentially long. VCSPs of bounded treewidth, however, are tractable – even for finding global optima – by using a non-local-search algorithm [1, 2]. But the existence of efficient non-local algorithms does not mean that local search algorithms will find optima efficiently.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 17; pp. 17:1–17:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

For situations where local search is forced upon us – as is for example the case when we are modelling biological evolution [11, 10, 13] or social systems in business [15, 17] and economics [18]; or energy-minimization in physical systems – it is useful to know whether we can expect the local search algorithm to terminate in reasonable time.

Even in the case of VCSPs of bounded treewidth, a (reasonable) local search algorithm like steepest ascent may take a long time to equilibrate. [3] have provided a Boolean VCSP with arity 8 and pathwidth 7 (and treewidth 7), on which an exponentially long steepest ascent exists. This shows that what is tractable/intractable for steepest ascent is distinct from what is tractable/intractable for non-local algorithms. We currently do not yet have a full characterisation of the intractability class for steepest ascent. In this article, we take a next step towards this full characterisation and lower the threshold for intractability, by constructing a Boolean VCSP of arity 5 and pathwidth 4 that has exponentially long steepest ascents. We do this in four steps:

Section 3: Introduce a general padding argument that allows us to simulate any ascent respecting an ordering of the domains by a steepest ascent (see Theorem 9).

Section 4: Introduce a new VCSP (specifically path made by alternating two different 2-by-3 constraints between variables of domain size alternating between 2 and 3) that produces an exponentially long ordered ascent (see Proposition 12).

Section 5: Apply the padding argument to the 2-by-3 construction. This yields a 3-by-5 construction, implementable by a VCSP with ternary constraints, on which the original ordered ascent is simulated by a steepest ascent.

Section 6: Encode the expanded domains using Boolean variables, and apply some tricks to get the resulting VCSP to arity 5 and pathwidth 4 while preserving the exponentially long steepest ascent.

2 Background

Let D be a set and $R \subset D \times D$ a binary relation on D. We call (D, R) a *domain* and R the *transition relation.* We omit R when it is a complete graph or obvious from context.

We will consider local search problems on *search spaces* of the form $D_1 \times D_2 \times \cdots \times D_n$, where the (D_i, R_i) are domains. We call the elements of $D_1 \times \cdots \times D_n$ *assignments.* We consider two assignments $x, y \in D_1 \times \cdots \times D_n$ to be as adjacent iff x differs from y at exactly one position, say k, and $(x_k, y_k) \in R_k$. We view R_k as being undirected, so that $(x_k, y_k) \in R_k$ implies that both the transition from x_k to y_k and the transition from y_k to x_k are allowed. $N(x)$ is the set of assignments adjacent to x.

▶ **Definition 1.** *Let $D_1 \times \cdots \times D_n$ be a search space of assignments, and $f : D_1 \times \cdots \times D_n \to \mathbb{Z}$ a function. We call f a* fitness function *and the pair $(D_1 \times \cdots \times D_n, f)$ a* fitness landscape. *We say that an assignment x is a* local solution *in the fitness landscape $(D_1 \times \cdots \times D_n, f)$ if $f(x) \geq f(y)$ for all $y \in N(x)$.* [1]

We can represent fitness landscapes using a collection of constraints. A *valued constraint* on $D_1 \times \cdots \times D_n$ with scope $S \subseteq [n]$ is a function $C_S : \prod_{i \in S} D_i \to \mathbb{Z}$. The size $|S|$ of the scope is the *arity* of the constraint.

In general, we can represent a constraint of arity n by an n-dimensional tensor, whose fibers are indexed by the domains in the scope of the constraint. In particular, this means that a binary constraint between two domains D_k and D_l can be represented by a matrix whose rows are indexed by D_k and whose columns are indexed by D_l (or vice versa).

[1] We could have used the more traditional "value" or "reward", but we prefer "fitness" given the connection to biological evolution that we discuss in Section 7.

▶ **Example 2.** Let $C_{\{i,j\}}$ be a binary constraint between domains $D_i = \{u_1, u_2, u_3, u_4\}$ and $D_j = \{v_1, v_2, v_3\}$, then we can represent $C_{\{i,j\}}$ by

$$C_{\{i,j\}} = \begin{array}{cccc} v_1 & v_2 & v_3 & \\ \left(\begin{matrix} C_{\{i,j\}}(u_1, v_1) \\ C_{\{i,j\}}(u_2, v_1) \\ C_{\{i,j\}}(u_3, v_1) \\ C_{\{i,j\}}(u_4, v_1) \end{matrix}\right. & \begin{matrix} C_{\{i,j\}}(u_1, v_2) \\ C_{\{i,j\}}(u_2, v_2) \\ C_{\{i,j\}}(u_3, v_2) \\ C_{\{i,j\}}(u_4, v_2) \end{matrix} & \left.\begin{matrix} C_{\{i,j\}}(u_1, v_3) \\ C_{\{i,j\}}(u_2, v_3) \\ C_{\{i,j\}}(u_3, v_3) \\ C_{\{i,j\}}(u_4, v_3) \end{matrix}\right) & \begin{matrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{matrix} \end{array} \tag{1}$$

▶ **Definition 3** (Valued Constraint Satisfaction Problem (VCSP), based on [8]). *An instance of a* valued constraint satisfaction problem (VCSP) *is given by a a tuple $(D_1 \times \cdots \times D_n, \mathcal{C})$, where $\mathcal{C} = \{C_{S_1}, C_{S_2}, \ldots, C_{S_m}\}$ is a collection of valued constraints on domains $D_1 \times D_2 \times \cdots \times D_n$. The fitness function f of the VCSP instance is given by $f(x) = \sum_{i=1}^{m} C_{S_i}((x_j)_{j \in S_i})$. The goal is to find a local solution in the fitness landscape $(D_1 \times \cdots \times D_n, f)$.*

We will usually denote a VCSP instance $(D_1 \times \cdots \times D_n, \mathcal{C},)$ simply by $\mathcal{C}$, and we will say that the fitness landscape $(D_1 \times \cdots \times D_n, f)$ is *implemented* by $\mathcal{C}$.

To any VCSP $\mathcal{C}$ on n variables, we can associate a *constraint (hyper)graph $G_{\mathcal{C}}$*, whose set of vertices is given by $[n]$. For each constraint C_s, there is a (hyper)edge S, labeled by C_s. We are interested in VCPS with "sparse" constraint graphs. Specifically, constraint graphs of bounded *pathwidth.*

▶ **Definition 4** (adapted from [5]). *Let $G_{\mathcal{C}} = ([n], \{S \mid C_S \in \mathcal{C}\})$ be the constraint graph of a VCSP $\mathcal{C}$. A sequence $(X_i)_{i=1}^{m}$ of subsets $X_i \subseteq [n]$ is called a* path decomposition *of $G_{\mathcal{C}}$ if the following three properties hold*

1. *For all $h \in [n]$, there is some i such that $h \in X_i$.*
2. *For all $S \in E(G_{\mathcal{C}})$, there is some i such that $S \subseteq X_i$.*
3. *For $h \in [n]$ and for all $i \leq j \leq k$, if $h \in X_i$ and $h \in X_k$, then $h \in X_j$.*

The pathwidth *of $G_{\mathcal{C}}$ is the least l such that $l \geq |X_i| - 1$ for all $1 \leq i \leq m$.*

Note that when the value of a single variable changes within a VCSP, the accompanying change in fitness value is determined entirely by those constraints whose scope contains this variable. Because of this, it is often useful to consider the "restricted" fitness function resulting from only considering these domains. Let f be a fitness function associated to some VCSP $\mathcal{C}$, and let $\mathcal{C}[k] \subset \mathcal{C}$ be the set of constraints whose scope contains the k-th variable. We use f_k to denote the function given by $f_k(u; z) = \sum_{C_{S_i} \in \mathcal{C}[k]} C_{S_i}(u, (z_j)_{j \in S_i \setminus \{k\}})$ where u is the value of the k-th variable, and z is a (sub-)assignment consisting of values for all variables who share a constraint with the k-th variable.

▶ **Definition 5.** *Let $\mathbb{F} = (D_1 \times \cdots \times D_n, f)$ be a fitness lanscape, and let $p = (x^t)_{t=0}^{T} \subset (D_1 \times \cdots \times D_n)^{T+1}$ be a sequence of assignments in $D_1 \times \cdots \times D_n$. We call p an* ascent *on $\mathbb{F}$ if for all $t < T$, we have $x^{t+1} \in N(x^t)$, we have $f(x^t) < f(x^{t+1})$, and x^T is a local solution.*

In this article, we are focused on ascents which take steps that most increase fitness.

▶ **Definition 6.** *Let $\mathbb{F} = (D_1 \times \cdots \times D_n, f)$ be a fitness landscape, and let $p = (x^t)_{t=0}^{T}$ be an ascent on $\mathbb{F}$. We call p a* steepest ascent *on $\mathbb{F}$ if for all $t < T$ and for all $y \in N(x^t)$, we have $f(y) \leq f(x^{t+1})$.*

In other words, at any step, all neighbours of an assignment in p have fitness less than or equal to the fitness of the next assignment in p.

We will use the following notation to represent local changes: if y can be generated from x by changing the k-th entry from $x_k = u$ to $y_k = v$, we write $y = x[k : v]$.[2] Furthermore, if $p = (x^t)_{t=0}^T$ is an ascent, we write $u \xrightarrow{k}_{p(t)} v$ to denote that the transition from x^t to x^{t+1} in p consists of replacing symbol u with symbol v at position k. If the ascent is clear from context, we drop the p from the notation.

▶ **Definition 7.** *Let $\mathbb{F} = (D_1 \times \cdots \times D_n, f)$ be a fitness landscape, and let $p = (x^t)_{t=0}^T$ be an ascent on $\mathbb{F}$. Let $\prec$ be an ordering on $[n]$. We call p a* $\prec$-ordered ascent *on $\mathcal{C}$ if the following holds. For any $t < T$, if $x^{t+1} = x^t[k : v]$, then, for all $j \prec k$ and for all $u \in D_j$ with $(x_j^t, u) \in R$, we have $f(x^t[j : u]) \leq f(x^t)$.*

In other words, at any step, the ascent changes an entry in the domain with $\prec$-minimal index where a change can yield a fitness increase.

3 Steepest Ascent Simulation of Ordered Ascents

Given an ordered ascent on some fitness landscape, we show how to construct a new fitness landscape that "simulates" the ordered ascent with a steepest ascent. This will be done by expanding the domain and then encoding the expanded domain using Boolean variables.

3.1 Domain Expansion

Let $\mathbb{F} = (D_1 \times \cdots \times D_n, f)$ be a fitness landscape. Let $p = (x^t)_{t=0}^T$ be a $\prec$-ordered ascent on $\mathbb{F}$. For any $k \in \{1, 2, \ldots, n\}$, we expand D_k by adding *intermediate states* $\sigma_{uv} = \sigma_{vu}$ for all $(u, v) \in R_k$. We call the elements of the original domain D_k *main states*. We denote the resulting expanded domain by $\widehat{D}_k \supset D_k$. We define the new transition relation on the expanded domains to be $\widehat{R}_k = \{(u, \sigma_{uv}) \mid (u, v) \in R_k\} \cup \{(\sigma_{uv}, v) \mid (u, v) \in R_k\}$. This relation ensures that the only possible transition are those from a main state to an intermediate state and vice versa.

We now construct a new fitness function $\widehat{f} : \widehat{D}_1 \times \cdots \times \widehat{D}_n \to \mathbb{Z}$. For any $x \in D_1 \times \cdots \times D_n$ – i.e., any x containing only main states – we set:

$$\widehat{f}(x) := (2n + 1)f(x) \qquad \text{if } x \text{ contains only main states.} \tag{2}$$

For an assignment x containing a single intermediate state σ_{uv} at position k and main states at all other positions, if $f(x[k : u]) \neq f(x[k : v])$ then we set $\widehat{f}(x)$ to be:

$$\widehat{f}(x) := n - k + 1 + (2n + 1) \min_{w \in \{u,v\}} f(x[k : w]) \qquad \text{if } x \text{ containts exactly one intermediate state } \sigma_{uv} \text{ at position } k. \tag{3}$$

If $f(x[k : u]) = f(x[k : v])$ then we set $\widehat{f}(x) = (2n + 1) \min_{w \in \{u,v\}} \{f(x[k : w])$.

Next, let x be an assignment that contains exactly two intermediate symbols $\sigma_{u_j v_j}, \sigma_{u_k v_k}$ at positions j and k respectively. We do not want such an assignments to appear in the steepest ascent. To ensure this, we require that $\widehat{f}$ satisfies the following:

$$\widehat{f}(x) \leq 2n - (j + k) + 2 + (2n + 1) \min_{\substack{w \in \{u_j, v_j\} \\ w' \in \{u_k, v_k\}}} f(x[j, k : w, w']) \qquad \text{if } x \text{ contains exactly two intermediate states } \sigma_{u_j v_j} \text{ and } \sigma_{u_k v_k} \text{ at positions } j \text{ and } k. \tag{4}$$

For remaining assignments with more than two intermediate symbols, $\widehat{f}$ may take any value.

[2] Note that $y = x[k : v]$ is equivalent to $x = y[k : u]$.

3.2 Steepest Ascent Simulation

▶ **Definition 8.** *Given an ascent $p = (x^t)_{t=0}^T$ on a fitness landscape $(D_1 \times \cdots \times D_n, f)$, we define an ascent $\widehat{p} = (\widehat{x}^t)_{t=0}^{2T}$ on the new fitness landscape $(\widehat{D}_1 \times \cdots \times \widehat{D}_n, \widehat{f})$ as:*

$$\widehat{x}^t = \begin{cases} x^s, & t = 2s; \\ x^s[k : \sigma_{uv}], & t = 2s+1,\ x^{s+1} = x^s[k : v] \text{ and } x^s = x^{s+1}[k : u]. \end{cases} \tag{5}$$

We say $\widehat{p}$ a simulation *of p.*

Note that $\widehat{p}$ alternates between main states of p and the relevant intermediate states between them. Given the definition of $\widehat{p}$, we are now ready to state our main result.

▶ **Theorem 9.** *Let $\mathbb{F} = (D_1 \times \cdots \times D_n, f)$ be a fitness landscapes, and let $\prec$ be an ordering on $\{1, 2, \ldots, n\}$. Suppose p is a $\prec$-ordered ascent on $\mathbb{F}$ and that $\widehat{p}$ is a simulation of p. Then, $\widehat{p}$ is a steepest ascent on $(\widehat{D}_1 \times \cdots \times \widehat{D}_n, \widehat{f})$.*

By reindexing the domains, we may assume that p is $<$-ordered without loss of generality (where $<$ is the standard ordering on $[n]$). We prove this theorem through the following two lemmas. The first shows that the transitions from main states into intermediate states are steepest ascent steps. The second shows that the transitions from intermediate states into main states are steepest ascent steps.

▶ **Lemma 10.** *Let $x^t, x^{t+1} \in p$, with the transition between these two states being $u \xrightarrow{k}_{p(t)} v$. Then, the highest fitness neighbour of $\widehat{x}^{2t} \in \widehat{p}$ is $\widehat{x}^{2t}[k : \sigma_{uv}]$. Moreover, $\widehat{x}^{2t}[k : \sigma_{uv}]$ has higher fitness than $\widehat{x}^{2t}$.*

Proof. We begin by noting that $\widehat{x}^{2t}$ contains only main states. Due to the nature of the encoding, the only possible transitions are those flipping a main state to an intermediate state. Note that by equation (3), we have

$$\widehat{f}(\widehat{x}^{2t}[k : \sigma_{uv}]) - \widehat{f}(\widehat{x}^{2t}) = n - k + 1 > 0. \tag{6}$$

Consider any $l < k$. Since we may assume p is a $<$-ordered ascent, we know that for any neighbour $x^t[l : w]$ of x^t, we have $f(x^t[l : w]) \leq f(x^t)$. It follows that

$$\widehat{f}(\widehat{x}^{2t}[l : \sigma]) \leq \widehat{f}(\widehat{x}^{2t}) < \widehat{f}(\widehat{x}^{2t}[k : \sigma_{uv}]$$

for any intermediate state σ in $\widehat{D}_l$.

Next, consider any $l \geq k$. From equation (3), it is clear that the fitness increase from a flip to an intermediate state will be at most $n - l + 1 \leq n - k + 1$. Thus, the transition from $\widehat{x}^{2t}$ to neighbour $\widehat{x}^{2t}[k : \sigma_{uv}]$ with fitness increase of $n - k + 1$, is a steepest step as desired. ◀

▶ **Lemma 11.** *Let $x^t, x^{t+1} \in p$, with the change between these two states being $u \xrightarrow{k}_{p(t)} v$. Then, the highest fitness neighbour of $\widehat{x}^{2t+1} \in \widehat{p}$ is $\widehat{x}^{2t+1}[k : v]$. Moreover, $\widehat{x}^{2t+1}[k : v]$ has higher fitness than $\widehat{x}^{2t+1}$.*

Proof. We begin by noting that by definition, $\widehat{x}^{2t+1}$ consists of the intermediate state σ_{uv} at position k, and main states at all other positions. We have $\widehat{f}(\widehat{x}^{2(t+1)}) - \widehat{f}(\widehat{x}^{2t}) \geq 2n+1$, since $\widehat{x}^{2(t+1)} = x^{t+1}$, $\widehat{x}^{2t} = x^t$ and $f(x^{t+1}) > f(x^t)$. Moreover $\widehat{f}(\widehat{x}^{2t+1}) - \widehat{f}(\widehat{x}^{2t}) = n - k + 1$. Thus $\widehat{f}(\widehat{x}^{2(t+1)}) - \widehat{f}(\widehat{x}^{2t+1}) \geq 2n + 1 - (n - k + 1) \geq n + 1$. Since a transition to an intermediate state at position $j \in [n]$ yields a fitness increase of $n - j + 1$, any transition to an intermediate state can yield a fitness increase of at most n. Thus, we know that such a transition will never be preferred over the transition from intermediate state σ_{uv} to main state v, which yields a fitness increase of at least $n+1$. By definition of the transition relation, the only alternative transition is from σ_{uv} to main state u but this decreases fitness by $n-k+1$. Thus the transition from $\widehat{x}^{2t+1}$ to neighbour $\widehat{x}^{2t+1}[k:v]$ is steepest step as desired. ◀

Together Lemmas 10 and 11 prove Theorem 9, implying that $\widehat{p}$ is a steepest ascent.

3.3 Boolean Encoding of Expanded Domains

Note that both p was an ascent on a fitness landscape of arbitrary domain sizes and $\widehat{p}$ was an ascent on a fitness landscape with further expanded domains. Given that we want to arrive at a Boolean VCSP, we will show how to encode each $(\widehat{D}_k, \widehat{R}_k)$ using $|D_k|$ Boolean variables. Without loss of generality, we may assume that $D_k = \{1, 2, \ldots, |D_k|\}$. We can now encode any main state $u \in D_k$ by the string of length $|D_k|$ that contains a 1 at position u and a 0 in all other positions. Any intermediate state σ_{uv} can likewise be encoded by the string of length $|D_k|$ containing a 1 at positions u and v and a 0 at all other positions.

Using this encoding we ensure that all main states are more than a single bit-flip away from each other. At the same time, the intermediate states are exactly one flip away from the corresponding two main states (and more than one flip away from all other main states). Thus, the transition relation $\widehat{R}_k$ is respected by this encoding.

4 Pair of 2-by-3 Constraints with Long Ordered Ascent

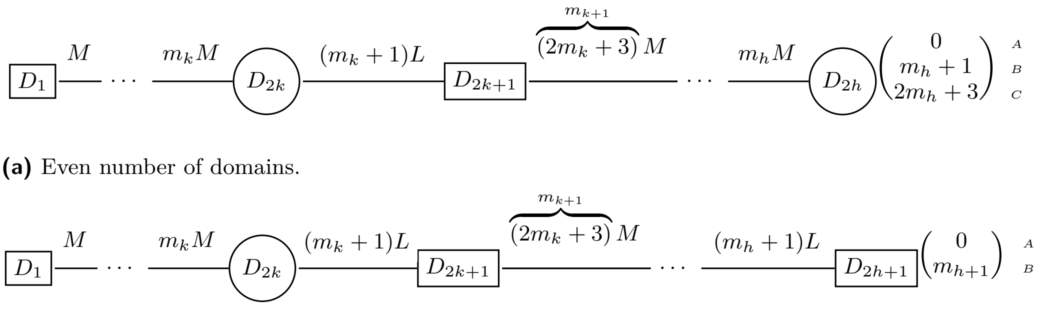
To build an ordered ascent of exponential length, we consider a VCSP on n domains $D_1 \times D_2 \times \cdots \times D_n$, where the odd domains have size 2 (i.e. $D_1, D_{2k+1} = \{A, B\}$), and the even domains have size 3 (i.e. $D_{2k} = \{A, B, C\}$. For the odd domains, the transition relation is simply given by $R_1, R_{2k+1} = \{(A, B)\}$. For the even domains, the transition relation is given by $R_{2k} = \{(A, B), (B, C)\}$. In other words, the values are only allowed to transition between A-and-B, and between B-and-C. Transitions between A-and-C are not allowed.

We arrange the domains into a path, where each pair of consecutive domains have a binary constraint between them. The n-th domain gets a unary constraint inspired by the relevant binary constraint. The binary constraints are different weights of the following:

$$L = \begin{array}{ccc} \scriptstyle A & \scriptstyle B & \\ 0 & 2 & \scriptstyle A \\ 1 & 1 & \scriptstyle B \\ 2 & 0 & \scriptstyle C \end{array} \,, \quad M = \begin{array}{cccc} \scriptstyle A & \scriptstyle B & \scriptstyle C & \\ 0 & 1 & 0 & \scriptstyle A \\ 1 & 0 & 1 & \scriptstyle B \end{array} \tag{7}$$

We recursively define weights for these constraints by setting $m_1 = 1$, and $m_{k+1} = 2m_k + 3$. We can solve this recurrence relation to get $m_k = 2^{k+1} - 3$. To D_1 and D_2 we assign constraint M. Between D_{2k} and D_{2k+1} we set constraint $(m_k + 1)L$. Between D_{2k+1} and $D_{2(k+1)}$ we set constraint $m_{k+1}M$. Finally, if $n = 2h$, we assign unary constraint $(m_h + 1)L(-, A)$ to D_n. If $n = 2h + 1$, we assign unary constraint $m_{h+1}M(-, A)$ to D_n. These cases are shown in Figure 1.

Note that whenever a variable at an even position $2k$ changes value, it can lose at most m_k from the left constraint (i.e., $m_k M$), while any gain from the right constraint (i.e. $(m_k + 1)L$) will be at least $m_k + 1$. Likewise, whenever a variable at an even position $2k + 1$ changes value, it can lose at most $2(m_k + 1)$ from the left constraint (i.e. $(m_k + 1)L$), while any gain


D_1
M
$m_k M$
D_{2k}
$(m_k+1)L$
D_{2k+1}
$\overbrace{(2m_k+3)}^{m_{k+1}} M$
$m_h M$
D_{2h}
$\begin{pmatrix}0\\ m_h+1\\ 2m_h+3\end{pmatrix}$
A
B
C
$(m_h+1)L$
D_{2h+1}
$\begin{pmatrix}0\\ m_{h+1}\end{pmatrix}$

(a) Even number of domains.

(b) Odd number of domains.

Figure 1 Constraint graphs for 2-by-3 VCSP when there are an (a) even number of domains and when there are an (b) odd number of domains. Constraints L and M are given in Equation (7). The final domain is assigned a unary constraint $(m_h+1)L(-,A)$ (in the even case) or $m_{h+1}M(-,A)$ (in the odd case), represented as a column vector in the figures. Boxes represend domains with two values, and circles represent domains with three values.

from the right constraint (i.e. $m_{k+1}M$) will be at least $m_{k+1} = 2(m_k+1)+1$. Thus, any fitness gain from the right constraint will always outweigh the fitness loss from the left. The maximal fitness value that can be attained by an assignment of length n is given by

$$f_{\max}(n) := \begin{cases} 0, & n = 0 \\ \sum_{i=1}^{h}(3m_i+2) = 3\cdot 2^{h+2} - 7h - 12, & n = 2h \text{ for some } h > 0 \\ f_{\max}(2h) + m_{h+1} = 2^{h+4} - 7h - 15, & n = 2h+1 \text{ for some } h \geq 0. \end{cases} \tag{8}$$

This fitness value is attained by the assignment $BABA\ldots ABABC$ for $n = 2h$ and the assignment $BABA\ldots ABAB$ for $n = 2h+1$.

We now prove that there is an ascent that takes on all fitness values from 0 to $f_{\max}(n)$ and is thus exponentially long in the number of variables. To prove this, it will be useful to say that a constraint $C_S \in \mathcal{C}$ is *saturated* by an assignment x if $C_S((x_i)_{i\in S}) \geq C_S(y)$ for all $y \in \prod_{i\in S} D_i$.

▶ **Proposition 12.** *Let $n \in \mathbb{N}$, and let $D_1 \times D_2 \times \cdots \times D_{n-1} \times D_n$ be the search space given by setting $D_{2k} = \{A, B, C\}$ and $D_{2k+1} = \{A, B\}$, with only transitions between A-and-B and B-and-C being allowed. Let $p = (x^t)_{t=0}^{T}$ be the $<$-ordered ascent on the VCSP with constraints from Figure 1, given by starting at $x^0 = A^n$. Then, p has length $f_{\max}(n) \geq 3 \cdot 2^{\lfloor \frac{n}{2} \rfloor} - O(n)$.*

Proof. We will show that for any assignment on n variables whose fitness is below $f_{\max}(n)$, there is a fitness increasing move. Moreover, the fitness increasing move on the least index where such a move is possible yields a fitness increase of 1. Since A^n has fitness 0, it follows from these facts that p has length $f_{\max}(n)$.

Let $x \in D_1 \times \cdots \times D_n$ be an assignment on n variables and suppose $f(x) < f_{\max}(n)$. Then, there must be some position $k \leq n$ such that the constraint between x_k and x_{k+1} is not saturated (or in the case that $k = n$, then the unary constraint on x_n is not saturated). Consider the least k for which is the case.

Firstly, consider the case where $k = 1$. Then, we must have $M(x_1, x_2) = 0$. By considering M, we see that we can change x_1 to obtain a increase of 1.

Next, consider the case where $k = 2l$. We must have that $M(x_{k-1}, x_k) = 1$, since by assumption, this constraint is saturated. Recall that the transition relations of our domains are such that we may only transition between A-and-B, and between B-and-C. By considering the constraints, we see that if $x_k = B$, one of the two options yields a increase of $m_l + 1$ from the right constraint (which is an L-constraint with weight $m_l + 1$), while the other yields a decrease of $m_l + 1$ from right constraint. If $x_k = A$ or $x_k = C$, then due to the nature of the transition relation, the k-th variable can transition only to B. The fact that $M(x_{k-1}, x_k) = 1$ implies that we are in case guaranteed to lose m_l from the left constraint by changing x_k. Furthermore, $L(x_k, x_{k+1})$ not being saturated, ensures that changing x_k to B is guaranteed to yield an increase of $m_l + 1$ from the right constraint. Thus, the fitness-improving change of x_k yields a net fitness increase of 1 by changing x_k.

Finally, consider the case where $k = 2l + 1$. We must have that $L(x_{k-1}, x_k) = 2$, and $M(x_k, x_{k+1}) = 0$. By considering the constraints, we see that the only possible change for x_k yields a fitness increase of $2m_l + 3 = 2(m_l + 1) + 1$, from the right constraint, while it yields a fitness loss of $2(m_l + 1)$ from the left constraint. Thus, this step again yields a net fitness increase of 1. In the cases where $k = n$, the above arguments hold when we fix $x_{k+1} = A$. ◀

5 Steepest Ascent Simulation of 2-by-3 Ordered Ascent

Since the exponentially long ascent described in Section 4 is ordered, we can construct a steepest ascent that simulates it by applying the technique introduced in Section 3. This yields a steepest ascent on an expanded VCSP with alternating 5-state and 3-state domains, and constraints with arity at most 3. The construction has three steps: (i) we define constraints that implement Equation (2), then we define constraints that implement Equation (3) for (ii) odd domains and for (iii) even domains.

Our expanded domains are given by $\widehat{D}_1, \widehat{D}_{2k+1} = \{A, B, \sigma_{AB}\}$ for the odd domains, and $\widehat{D}_{2k} = \{A, B, C, \sigma_{AB}, \sigma_{BC}\}$ for the even domains. In order to obtain the required fitness value for our main states, we add expanded versions $\widehat{L}$ and $\widehat{M}$ from Equation (7). These expanded constraints are still binary, and are given by firstly setting $\widehat{L}_{uv} = L_{uv}$ and $\widehat{M}_{vu} = M_{vu}$ for all $u \in \{A, B, C\}$ and $v \in \{A, B\}$. All other entries in $\widehat{L}$ and $\widehat{M}$, for which at least one of the indices is an intermediate state, are set to 0.

We place these constraints into a path as in Section 4, with new weight $2n + 1$ times the original weight. For any assignment x containing only main states, these expanded constraints yield $\widehat{f}(x) = (2n + 1)f(x)$, which is Equation (2), as desired.

We will separately define constraints that ensure that Equation (3) holds for neighbourhoods of odd domains, and for neighbourhoods of even domains. We begin with the odd domains by introducing a ternary constraint $\widehat{P}$ that we call the minimisation constraint, as well as unary constraint $\widehat{U}$.

Suppose intermediate state σ_{AB} is at odd position $k = 2l + 1$. We assume $u = x_{k-1}$ and $v = x_{k+1}$ are main states. Equation (3) requires our restricted fitness function $\widehat{f}_k$ to satisfy:

$$\begin{aligned}
\widehat{f}_k(\sigma_{AB}; u, v) &= n - k + 1 + (2n + 1) \min_{h \in \{A, B\}} \{f_k(h; u, v)\} && (9)\\
&= n - k + 1 + (2n + 1) \min \begin{cases} (m_l + 1) \cdot L(u, A) + m_{l+1} \cdot M(A, v) \\ (m_l + 1) \cdot L(u, B) + m_{l+1} \cdot M(B, v) \end{cases} && (10)\\
&= n - k + 1 + (2n + 1)(m_l + 1) \cdot K(u, v) && (11)
\end{aligned}$$

where $K = \begin{array}{c@{}c} \begin{array}{ccc} \scriptstyle A & \scriptstyle B & \scriptstyle C \end{array} & \\ \begin{pmatrix} 0 & 2 & 0 \\ 1 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix} & \begin{array}{c} \scriptstyle A \\ \scriptstyle B \\ \scriptstyle C \end{array} \end{array}$. K specifies the non-zero part of the minimisation constraint $\widehat{P}$, by setting $\widehat{P}_{uwv} = K_{uv}$ for $u, v \in \{A, B, C\}$ and $w = \sigma_{AB}$. All other entries in $\widehat{P}$ (i.e. those for w is a main state, or at least one of u and v is an intermediate state) are set to 0. In order to get the $+n - k + 1$ term from Equation (3), we need a unary constraint $\widehat{U}^\top = \begin{array}{c} \begin{array}{ccc} \scriptstyle A & \scriptstyle B & \scriptstyle \sigma_{AB} \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \end{array}$. For odd $k = 2l + 1$, we assign $\widehat{U}$ with weight $n - k + 1$ to $\widehat{D}_k$. We assign ternary constraint $\widehat{P}$ with weight $(2n + 1)(m_l + 1)$ to $\widehat{D}_k$, $\widehat{D}_{k-1}$ and $\widehat{D}_{k+1}$.

Now, suppose that x is an assignment with exactly one intermediate symbol σ_{AB} at odd position $k = 2l + 1$. For any two adjacent main states in x, the binary constraint between them is given by the binary constraint from the original VCSP, multiplied by a factor $2n + 1$. Moreover, $\widehat{P}$ is 0 when there are two adjacent main states among its three indices. Together $\widehat{P}$ and $\widehat{U}$ ensure that for our single intermediate state σ_{AB} at position k, Equation (10) holds. This yields $\widehat{f}(x) = n - k + 1 + (2n + 1) \min_{h \in A,B}\{f(x[k : h])\}$ – the desired Equation (3).

Next, consider intermediate symbol $w \in \{\sigma_{AB}, \sigma_{BC}\}$ at even position $k = 2l$. Assume $u = x_{k-1}$ and $v = x_{k+1}$ are main states. Equation (3) requires:

$$\begin{aligned}
\widehat{f}_k(\sigma_{AB}; u, v) &= n - k + 1 + (2n + 1) \min_{h \in \{A,B\}} \{f_k(x[k : h])\} && (12)\\
&= n - k + 1 + (2n + 1) \min \begin{cases} m_l \cdot M(u, A) + (m_l + 1) \cdot L(A, v) \\ m_l \cdot M(u, B) + (m_l + 1) \cdot L(B, v) \end{cases} && (13)\\
&= n - k + 1 + (2n + 1) \cdot Q^l(u, v) && (14)\\
\widehat{f}_k(\sigma_{BC}; u, v) &= n - k + 1 + (2n + 1) \min_{h \in \{B,C\}} \{f_k(x[k : h])\} && (15)\\
&= n - k + 1 + (2n + 1) \min \begin{cases} m_l \cdot M(u, B) + (m_l + 1) \cdot L(B, v) \\ m_l \cdot M(u, C) + (m_l + 1) \cdot L(C, v) \end{cases} && (16)\\
&= n - k + 1 + (2n + 1) \cdot R^l(u, v) && (17)
\end{aligned}$$

where $Q^l = \begin{array}{c@{}c} \begin{array}{cc} \scriptstyle A & \scriptstyle B \end{array} & \\ \begin{pmatrix} 0 & 2m_l + 1 \\ m_l & m_l + 1 \end{pmatrix} & \begin{array}{c} \scriptstyle A \\ \scriptstyle B \end{array} \end{array}$ and $R^l = \begin{array}{c@{}c} \begin{array}{cc} \scriptstyle A & \scriptstyle B \end{array} & \\ \begin{pmatrix} 2m_l + 1 & 0 \\ m_l + 1 & m_l \end{pmatrix} & \begin{array}{c} \scriptstyle A \\ \scriptstyle B \end{array} \end{array}$. The matrices Q^l and R^l specify the non-zero part of the ternary minimisation constraint $\widehat{S}^l$, by setting $\widehat{S}^l_{uwv} = Q^l_{uv}$ for $w = \sigma_{AB}$, and $\widehat{S}^l_{uwv} = R^l_{uv}$ for $w = \sigma_{BC}$. All other entries in $\widehat{S}^l$ (i.e. those for w is a main state, or at least one of u and v is an intermediate state) are set to 0. Furthermore, in order to get the first summands of Equation (3), we need a unary constraint $\widehat{V}^\top = \begin{array}{c} \begin{array}{ccccc} \scriptstyle A & \scriptstyle B & \scriptstyle C & \scriptstyle \sigma_{AB} & \scriptstyle \sigma_{BC} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$. For any even $k = 2l$, we assign constraint $\widehat{V}$ with weight $n - k + 1$ to $\widehat{D}_k$. We also assign the ternary constraint S^l with weight $2n+1$ to $\widehat{D}_k$ and its neighbouring domains $\widehat{D}_{k-1}$ and $\widehat{D}_{k+1}$.[3]

[3] Note that introducing these new constraints does not impact Equation (3), since $\widehat{V}$ adds fitness value 0 for main states, and S_{uwv} adds fitness value 0 if w is a main state or u or v is an intermediate symbol (which are the only situations occurring in Equation (3)).

Now, suppose that x is an assignment with exactly one intermediate symbol $w \in \{\sigma_{AB}, \sigma_{BC}\}$ at even position $k = 2l$. Note that constraints $\widehat{P}$ and $\widehat{U}$ do not add any fitness value for this assignment. Thus, in the same manner as before, the binary constraints $\widehat{M}$ and $\widehat{L}$, together with $\widehat{S}$ and $\widehat{V}$ yield the desired $\widehat{f}(x) = k + (2n+1)f \min_{h \in A,B}(\{f(x[k:h])\}$.

In order to apply Theorem 9, it only remains to show that the inequality from Equation (4) holds. By inspecting $\widehat{S}^l$ and $\widehat{P}$ however, it is clear that this is the case. Specifically, we have equality in the case of two non-adjacent intermediate states, and strict inequality in the case of adjacent intermediate states.

6 Low-arity Boolean Encoding for 3-by-5 Steepest Ascent

In order to turn the alternating 3-state and 5-state VCSP from Section 5 into a Boolean VCSP, we encode the expanded domains using Boolean variables as in Section 3.3. The 3-state domain is encoded by two boolean variables with $A = 10$, $B = 01$ and $\sigma_{AB} = 11$. The 5-state domain is encoded using three variables with $A = 100$, $B = 010$, $C = 001$, $\sigma_{AB} = 110$ and $\sigma_{BC} = 011$. We write $\hat{\hat{D}}_k$ for an encoded domain and $\hat{\hat{f}}$ for our new fitness function. Note that $\hat{\hat{f}}$ takes the same values as our old fitness function $\widehat{f}$, but has a new domain.

Note that the constraints can be encoded by taking the value of the original constraint for any encoded states and 0 for strings that don't encode any states. In the encoded VCSP, the arity of the constraints defined in Section 5 increases. The $\widehat{S}^l$-constraint, which was a ternary constraint on a 3-state, a 5-state and another 3-state domain, turns into a constraint with arity $2 + 3 + 2 = 7$ in the encoded VCSP. The $\widehat{P}$-constraint, which was a ternary constraint on a 5-state, a 3-state, and another 5-state domain, turns into a constraint with arity $3 + 2 + 3 = 8$. We will use some tricks to reduce this arity to 5 in both cases. The resulting full encoded VCSP is shown in Figure 2.

First, we look at a lower arity implementation of the minimisation constraint $\widehat{P}$ for 3-state domain neighbourhoods, from Section 5. Suppose that we have an intermediate state σ_{AB} at odd position $k = 2l + 1$. We assume that x_{k-1} and x_{k+1} are main states, represented by u and v respectively. Recall that the non-zero part of $\widehat{P}$ is given by $\widehat{P}_{*\sigma_{AB}*} = \begin{array}{cc} \begin{matrix} {\scriptstyle A} & {\scriptstyle B} & {\scriptstyle C} \end{matrix} & \\ \begin{pmatrix} 0 & 2 & 0 \\ 1 & 1 & 1 \\ 2 & 0 & 2 \end{pmatrix} & \begin{matrix} {\scriptstyle A} \\ {\scriptstyle B} \\ {\scriptstyle C} \end{matrix} \end{array}$.[4]

Instead of encoding σ_{AB} with the single string 11, we can let both 00 and 11 perform part of the role of encoding σ_{AB}. Importantly, this encoding still agrees with our transition relation, since 00 and 11 are two bit-flips away from one another, and a single bit flip away from 10 and 01. We now have to set up the fitness values in such a way that we can transition through either 00 or 11. In order for this to work, we require that $\max_{s \in \{00,11\}} \hat{\hat{f}}_k(s; u, v) = \widehat{f}_k(\sigma_{AB}; u, v)$ for all u, v. We may view u and v as vectors, writing them as vectors with $\vec{A} = \begin{array}{c} \begin{matrix} {\scriptstyle A} & {\scriptstyle B} & {\scriptstyle C} \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \end{array}$, $\vec{B} = \begin{array}{c} \begin{matrix} {\scriptstyle A} & {\scriptstyle B} & {\scriptstyle C} \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 \end{pmatrix} \end{array}$ and $\vec{C} = \begin{array}{c} \begin{matrix} {\scriptstyle A} & {\scriptstyle B} & {\scriptstyle C} \end{matrix} \\ \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \end{array}$. We can achieve our desired property with:

[4] One could try to decompose $\widehat{P}$ into two arity-5 constraints between $\hat{\hat{D}}_{k-1}$-and-$\hat{\hat{D}}_k$ and one between $\hat{\hat{D}}_k$-and-$\hat{\hat{D}}_{k+1}$. This is not possible if σ_{AB} is encoded by a single string (see Appendix A).

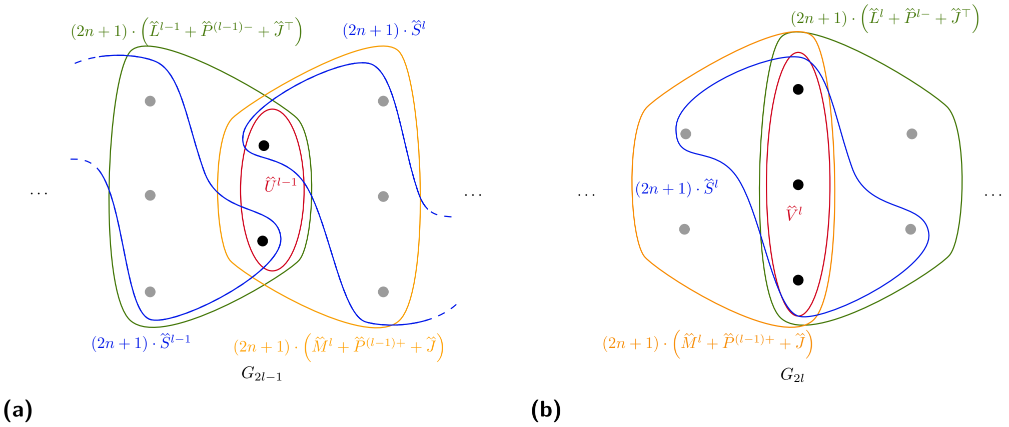

$(2n+1)\cdot\left(\hat{\hat{L}}^{l-1}+\hat{\hat{P}}^{(l-1)-}+\hat{\hat{J}}^{\top}\right)$
$(2n+1)\cdot\hat{\hat{S}}^{l}$
$\hat{\hat{U}}^{l-1}$
$(2n+1)\cdot\hat{\hat{S}}^{l-1}$
$(2n+1)\cdot\left(\hat{\hat{M}}^{l}+\hat{\hat{P}}^{(l-1)+}+\hat{\hat{J}}\right)$
G_{2l-1}
(a)
$(2n+1)\cdot\left(\hat{\hat{L}}^{l}+\hat{\hat{P}}^{l-}+\hat{\hat{J}}^{\top}\right)$
$(2n+1)\cdot\hat{\hat{S}}^{l}$
$\hat{\hat{V}}^{l}$
$(2n+1)\cdot\left(\hat{\hat{M}}^{l}+\hat{\hat{P}}^{(l-1)+}+\hat{\hat{J}}\right)$
G_{2l}
(b)

Figure 2 The final VCSP contains $5n$ domains, divided alternatingly into collections G_1, G_{2l+1} of 2 domains and collections G_{2l} of 3 domains. There are eight different types of constraints $\hat{\hat{M}}^l$, $\hat{\hat{L}}^l$, $\hat{\hat{P}}^{l-}$, $\hat{\hat{P}}^{l+}$, $\hat{\hat{S}}^l$, $\hat{\hat{J}}$, $\hat{\hat{U}}^l$, and $\hat{\hat{V}}^l$. Their values can be found in Equations (23)-(30). These constraints are arranged in a path of repeating chunks. These chunks are shown for (a) odd collections and (b) even collections. As exceptions, we do not include a $\hat{\hat{P}}^+$ constraint between the first two collections G_1 and G_2. Furthermore, for the final collection G_n, we pretend the pattern continues, but that the values for collections beyond G_n are fixed to 100 for even collections and 10 for odd collections. The steepest ascent starting from assignment $10100^{\lfloor \frac{n}{2} \rfloor}$ (with an extra 10 at the end in case n is odd), has length at least $3 \cdot 2^{\lfloor \frac{n}{2} \rfloor + 3} - O(n)$ (see Proposition 12 and Theorem 9). Furthermore, the constraint graph has pathwidth 4. To see this, take for the path decomposition the scopes of the $\hat{\hat{M}}^l$ constraints, the scopes of the $\hat{\hat{S}}^l$ constraints, and the scopes of the $\hat{\hat{L}}^l$ constraints as the bins, and put these bins in the path $\cdots$-$\hat{\hat{M}}^l$-$\hat{\hat{S}}^l$-$\hat{\hat{L}}^l$-$\hat{\hat{M}}^{l+1}$-$\cdots$.

$$\hat{\hat{f}}_k(00; u, v) = n - k + 1 + (2n+1)(m_l+1)\vec{u}\begin{pmatrix} 0 & -2 & 0 \\ 1 & -1 & 1 \\ 2 & 0 & 2 \end{pmatrix}\vec{v} \tag{18}$$

$$= n - k + 1 + (2n+1)(m_l+1)\vec{u}\left(\begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix}\begin{pmatrix} 1 & 1 & 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}\begin{pmatrix} 0 & -2 & 0 \end{pmatrix}\right)\vec{v}, \tag{19}$$

$$\hat{\hat{f}}_k(11; u, v) = n - k + 1 + (2n+1)(m_l+1)\vec{u}\begin{pmatrix} 0 & 2 & 0 \\ -1 & 1 & -1 \\ -2 & 0 & -2 \end{pmatrix}\vec{v} \tag{20}$$

$$= n - k + 1 + (2n+1)(m_l+1)\vec{u}\left(\begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}\begin{pmatrix} 1 & 1 & 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}\begin{pmatrix} -2 & 0 & -2 \end{pmatrix}\right)\vec{v}. \tag{21}$$

Note that $\widehat{P}_{*\sigma_{AB}*}$ is the element-wise maximum of the two 3×3 matrices above. From this, we obtain our desired arity-5 constraints $\hat{\hat{P}}^{l-}$ and $\hat{\hat{P}}^{l+}$ between $\hat{\hat{D}}_{k-1}$ and $\hat{\hat{D}}_k$, and between $\hat{\hat{D}}_k$ and $\hat{\hat{D}}_{k+1}$. The non-zero part of $\hat{\hat{P}}^{l-}$ and $\hat{\hat{P}}^{l+}$ are given by

$$\hat{\hat{P}}^{l-} = \begin{array}{c} \begin{array}{cc} {\scriptstyle 00} & {\scriptstyle 11} \end{array} \\ \begin{pmatrix} 0 & 2(m_l+1) \\ m_l+1 & m_l+1 \\ 2(m_l+1) & 0 \end{pmatrix} \end{array} \begin{array}{c} {\scriptstyle 100} \\ {\scriptstyle 010} \\ {\scriptstyle 001} \end{array} \quad \text{and } \hat{\hat{P}}^{l+} = -2(m_l+1) \cdot \begin{array}{c} \begin{array}{ccc} {\scriptstyle 100} & {\scriptstyle 010} & {\scriptstyle 001} \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{c} {\scriptstyle 00} \\ {\scriptstyle 11} \end{array} . \tag{22}$$

Next, we look at a lower arity implementation of the minimisation constraint $\widehat{S}^l$. Suppose that we have an intermediate state σ_{AB} at even position $k = 2l$. We assume that x_{k-1} and x_{k+1} are main states, represented by u and v respectively. Recall that the non-zero parts of $\widehat{S}^l$ are given by $\widehat{S}^l_{*\sigma_{AB}*} = \begin{array}{cc} \scriptstyle A & \scriptstyle B \\ \left(\begin{array}{cc} 0 & 2m_l+1 \\ m_l & m_l+1 \end{array}\right) & \begin{array}{c} \scriptstyle A \\ \scriptstyle B \end{array} \end{array}$ and $\widehat{S}^l_{*\sigma_{BC}*} = \begin{array}{cc} \scriptstyle A & \scriptstyle B \\ \left(\begin{array}{cc} 2m_l+1 & 0 \\ m_l+1 & m_l \end{array}\right) & \begin{array}{c} \scriptstyle A \\ \scriptstyle B \end{array} \end{array}$.

This time we will use a different trick to lower the arity of the encoded constraint from 7 to 5. Note that only one domain can enter transition. Since we are considering domain D_k entering transition, we can assume that u and w are main states. This implies that we do not need to look at both bits of u's and w's representation to know their value. We can just look at the right bit of u and the left bit of v. This reduces the arity of the constraint to 5. We get a new constraint $\widehat{\widehat{S}}^l$ whose scope consists of right bit of the encoding of x_{k-1}, three bits that encode x_k, and left bit of the encoding of x_{k+1}. Non-zero parts of $\widehat{\widehat{S}}^l$ are

$$\widehat{\widehat{S}}^l_{*110*} = \begin{array}{c} \begin{array}{cc} \scriptstyle 1 & \scriptstyle 0 \end{array} \\ \left(\begin{array}{cc} 0 & 2m_l+1 \\ m_l & m_l+1 \end{array}\right) \begin{array}{c} \scriptstyle 0 \\ \scriptstyle 1 \end{array} \end{array} \text{ and } \widehat{\widehat{S}}^l_{*011*} = \begin{array}{c} \begin{array}{cc} \scriptstyle 1 & \scriptstyle 0 \end{array} \\ \left(\begin{array}{cc} 2m_l+1 & 0 \\ m_l+1 & m_l \end{array}\right) \begin{array}{c} \scriptstyle 0 \\ \scriptstyle 1 \end{array} \end{array}.$$

By restricting our perspective to only a single bit of the representation of x_{k-1} and x_{k+1}, we can no longer distinguish whether x_{k-1} and x_{k+1} are main states, or whether they are intermediate states. Through this, we may inadvertently violate Equation (4). To remedy this, we introduce an arity-5 constraint $\widehat{J}$ that penalises the occurrence of two adjacent intermediate states by setting the non-zero part of $\widehat{J}$ as $-f_{\max}(n) \cdot \begin{array}{c} \begin{array}{cc} \scriptstyle 110 & \scriptstyle 011 \end{array} \\ \left(\begin{array}{cc} 1 & 1 \\ 1 & 1 \end{array}\right) \begin{array}{c} \scriptstyle 00 \\ \scriptstyle 11 \end{array} \end{array}$ where n is the number of domains in the original VCSP and $f_{\max}(n)$ is the maximal fitness from Equation (8) of the original 2-by-3 VCSP. The magnitude of this negative value is always larger than the magnitude of any fitness value assigned by S^l for any l.

We have now constructed a VCSP with Boolean domains. We arrange these domains alternatingly into collections G_{2l} consisting of 3 domains and G_{2l+1} of 2 domains. The VCSP has eight different types of constraints which we list explicitly. We have an arity-5 constraint $\widehat{\widehat{M}}^l$ on collections G_{2l-1} and G_{2l}:

$$\widehat{\widehat{M}}^l = \left(\begin{array}{cccccccc|c} \scriptstyle 100 & \scriptstyle 010 & \scriptstyle 001 & \scriptstyle 110 & \scriptstyle 101 & \scriptstyle 011 & \scriptstyle 000 & \scriptstyle 111 & \\ 0 & m_l & 0 & 0 & 0 & 0 & 0 & 0 & \scriptstyle 10 \\ m_l & 0 & m_l & 0 & 0 & 0 & 0 & 0 & \scriptstyle 01 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \scriptstyle 00 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \scriptstyle 11 \end{array}\right) \tag{23}$$

and an an arity-5 constraint $\widehat{\widehat{L}}^l$ on collections G_{2l} and G_{2l+1}, given by:

$$\widehat{\widehat{L}}^l = \left(\begin{array}{cccc|c} \scriptstyle 10 & \scriptstyle 01 & \scriptstyle 00 & \scriptstyle 11 & \\ 0 & 2(m_l+1) & 0 & 0 & \scriptstyle 100 \\ m_l+1 & m_l+1 & 0 & 0 & \scriptstyle 010 \\ 2(m_l+1) & 0 & 0 & 0 & \scriptstyle 001 \\ 0 & 0 & 0 & 0 & \scriptstyle 110 \\ 0 & 0 & 0 & 0 & \scriptstyle 101 \\ 0 & 0 & 0 & 0 & \scriptstyle 011 \\ 0 & 0 & 0 & 0 & \scriptstyle 000 \\ 0 & 0 & 0 & 0 & \scriptstyle 111 \end{array}\right) \tag{24}$$

Then, we have our (decomposed) minimisation constraints P^{l-} and P^{l+}, on collections G_{2l} and G_{2l+1}, and collections G_{2l+1} and $G_{2(l+1)}$, respectively. These constraints are given by:

$$\widehat{\widehat{P}}^{l-} = \begin{array}{ccccc} \scriptstyle 10 & \scriptstyle 01 & \scriptstyle 00 & \scriptstyle 11 & \\ 0 & 0 & 0 & 2(m_l+1) & \scriptstyle 100 \\ 0 & 0 & m_l+1 & m_l+1 & \scriptstyle 010 \\ 0 & 0 & 2(m_l+1) & 0 & \scriptstyle 001 \\ 0 & 0 & 0 & 0 & \scriptstyle 110 \\ 0 & 0 & 0 & 0 & \scriptstyle 101 \\ 0 & 0 & 0 & 0 & \scriptstyle 011 \\ 0 & 0 & 0 & 0 & \scriptstyle 000 \\ 0 & 0 & 0 & 0 & \scriptstyle 111 \end{array} \tag{25}$$

$$\widehat{\widehat{P}}^{l+} = -2(m_l+1) \cdot \begin{array}{ccccccccc} \scriptstyle 100 & \scriptstyle 010 & \scriptstyle 001 & \scriptstyle 110 & \scriptstyle 101 & \scriptstyle 011 & \scriptstyle 000 & \scriptstyle 111 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \scriptstyle 10 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \scriptstyle 01 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \scriptstyle 00 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \scriptstyle 11 \end{array} \tag{26}$$

The next minimisation constraint is the arity-5 constraint $\widehat{\widehat{S}}^l$ with scope consisting of a single domain in G_{2l-1}, all three domains in G_{2l} and one more domain in G_{2l+1}. The single domains are selected such that the scopes of $\widehat{\widehat{S}}^l$ and $\widehat{\widehat{S}}^{l+1}$ do not overlap. $\widehat{\widehat{S}}^l$ is given by:

$$\widehat{\widehat{S}}^l = \begin{array}{ccccc} \scriptstyle (0,0) & \scriptstyle (1,0) & \scriptstyle (0,1) & \scriptstyle (1,1) & \\ 0 & 0 & 0 & 0 & \scriptstyle 100 \\ 0 & 0 & 0 & 0 & \scriptstyle 010 \\ 0 & 0 & 0 & 0 & \scriptstyle 001 \\ 2m_l+1 & m_l+1 & 0 & m_l & \scriptstyle 110 \\ 0 & 0 & 0 & 0 & \scriptstyle 101 \\ 0 & m_l & 2m_l+1 & m_l+1 & \scriptstyle 011 \\ 0 & 0 & 0 & 0 & \scriptstyle 000 \\ 0 & 0 & 0 & 0 & \scriptstyle 111 \end{array} \tag{27}$$

where the (u, v) for the column indices takes u and v as the values for respective single domains in G_{2l-1} and G_{2l+1}. In order to ensure that this constraint does not inadvertently allow adjacent intermediate states, we introduce an arity-5 constraint $\widehat{\widehat{J}}$ on adjacent collections:

$$\widehat{\widehat{J}} = -f_{\max}(n) \cdot \begin{array}{ccccccccc} \scriptstyle 100 & \scriptstyle 010 & \scriptstyle 001 & \scriptstyle 110 & \scriptstyle 101 & \scriptstyle 011 & \scriptstyle 000 & \scriptstyle 111 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \scriptstyle 10 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \scriptstyle 01 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & \scriptstyle 00 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & \scriptstyle 11 \end{array} \tag{28}$$

where n is the number of collections, and $f_{\max}(n)$ is as defined in Equation (8).

Finally, we have arity-3 constraint V^l on G_{2l} and arity-2 constraint U^l on G_{2l+1}:

$$(\hat{\hat{U}}^l)^\top = \begin{array}{cccc} \scriptstyle 10 & \scriptstyle 01 & \scriptstyle 00 & \scriptstyle 11 \\ \big(0 & 0 & n-2l & n-2l\big) \end{array} \tag{29}$$

$$(\hat{\hat{V}}^l)^\top = \begin{array}{cccccccc} \scriptstyle 100 & \scriptstyle 010 & \scriptstyle 001 & \scriptstyle 110 & \scriptstyle 101 & \scriptstyle 011 & \scriptstyle 000 & \scriptstyle 111 \\ \big(\,0 & 0 & 0 & n-2l+1 & 0 & n-2l+1 & 0 & 0\,\big) \end{array} \tag{30}$$

We arrange these constraints into a path of repeating chunks between the collections, as shown in Figure 2.

7 Summary and Future Directions

In this paper, we presented a sequence of three constructions with each improving on the state-of-the-art for the “simplest” VCSP with exponential (steepest) ascents. In Section 4, we presented a binary VCSP with a path as its constraint graph and domains alternating in size between two-state and three-state. In Proposition 12, we showed that this 2-by-3 VCSP has an exponential ascent. This a simplification over [13]’s simplest example of an exponential ascent from a path-structured VCSP with all domains of size 3. Our example also has the added benefit over prior work of the exponential ascent being an ordered-ascent. The ascent being ordered allows us to apply our general padding technique from Section 3 to create a ternary VCSP with domains alternating between size 3 and 5 in Section 5. It then follows from Theorem 9 that this 3-by-5 ternary VCSP produces a fitness landscapes with exponential steepest ascents. This could be viewed as a simplification over the binary VCSP with domains of size 10 implicit in [3]’s construction of exponential steepest ascents. Finally, in Section 6, we encoded the 3-by-5 VCSP with Boolean domains to construct a Boolean VCSP with a constraint graph of pathwidth 4 that produce a fitness landscape with an exponentially long steepest ascent. This is an improvement over the pathwidth 7 of the best known prior construction [3].

Our final construction means that Boolean VCSPs of pathwidth 4 are intractable for local search by steepest ascent. Since our graph also has treewidth 4 this means that Boolean VCSPs of treewidth 4 are also intractable for steepest ascent. For tractability, [13] have shown that all ascents – and thus in particular the steepest ascent – have at most quadratic length when the constraint graph is a tree, i.e. has treewidth 1. This leaves a gap between treewidth 1 and treewidth 4 for which the status of steepest ascent for finding local maxima in Boolean VCSPs remains unknown. Our current best guess at the exact location of the tractability boundary for steepest ascent is at pathwidth 2:

▶ **Conjecture 13.** *There exists a polynomial $p(n)$ such that for any Boolean VCSP instance $\mathcal{C}$ on n variables if the constraint graph of $\mathcal{C}$ has pathwidth ≤ 2, then any steepest ascent in the associated fitness landscape has length at most $p(n)$.*

Of course, the existence of exponential steepest ascents does not mean that all ascents are long. In our construction, it is relatively easy to find a short ascent that violates the steepest ascent condition. In fact, [12] has shown that polynomially short ascents to some local solution exist from all initial assignment in fitness landscape from VCSPs of bounded treewidth. More generally, there exist efficient (non-local search) algorithms for finding the *global* maximum in VCSPs of bounded treewidth [1, 7, 2]. However, such global algorithms cannot always be run – especially in cases where the algorithm is actually some natural process and thus we have no (or only partial) control to “rewrite” the algorithm.

Biological evolution is an important local search algorithm that is set by nature [20, 16, 11]. The intractability of finding local peaks provides an explanations for important features of evolution like its open-endededness [10, 11]. In this case, we can read ascents as "adaptive paths" [4] and steepest ascent as a strong-selection weak mutation dynamic that is often studied in evolutionary biology [6, 16]. The VCSP's variables correspond to genetic loci, the valued constraints correspond to gene-interactions, and the constraint graphs of the VCSPs encoding fitness landscapes correspond to gene-interaction networks [19, 11].[5] In this case, finding the "simplest" VCSPs that have exponential steepest ascents allows us to reason about the minimal conditions for open-endedness in evolution. Thus, our hope is that further progress on local search for VCSPs increases not only our understanding of combinatorial optimization but also of natural processes like biological evolution.

References

1 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.

2 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.

3 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.

4 K. Crona, D. Greene, and M. Barlow. The peaks and geometry of fitness landscapes. *Journal of Theoretical Biology*, 317:1–10, 2013.

5 Reinhard Diestel. *Graph Theory*. Springer Publishing Company, Incorporated, 5th edition, 2017.

6 John H Gillespie. A simple stochastic gene substitution model. *Theoretical population biology*, 23(2):202–215, 1983.

7 Georg Gottlob, Gianluigi Greco, and Francesco Scarcello. Tractable optimization problems through hypergraph-based structural restrictions. In *International Colloquium on Automata, Languages, and Programming*, pages 16–30. Springer, 2009.

8 P. Jeavons, A. Krokhin, and S. Živný. The complexity of valued constraint satisfaction. *Bulletin of the European Association for Theoretical Computer Science*, 113:21–55, 2014.

9 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.

10 Artem Kaznatcheev. Computational complexity as an ultimate constraint on evolution. *Genetics*, 212(1):245–265, 2019.

11 Artem Kaznatcheev. *Algorithmic Biology of Evolution and Ecology*. PhD thesis, University of Oxford, 2020.

12 Artem Kaznatcheev. Local search for valued constraint satisfaction parameterized by treedepth. *ArXiv*, 2024.

13 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.

14 Mark W. Krentel. On finding and verifying locally optimal solutions. *SIAM Journal on Computing*, 19(4):742–749, 1990. doi:10.1137/0219052.

[5] Similar models where the local search algorithm is set by nature exist for social systems studied in business [15, 17] and economics [18]; and for energy-minimization in physical systems.

15 Daniel A Levinthal. Adaptation on rugged landscapes. *Management Science*, 43(7):934–950, 1997.

16 H. Allen Orr. The genetic theory of adaptation: a brief history. *Nature Reviews. Genetics*, 6:119–127, 2005.

17 Jan W Rivkin and Nicolaj Siggelkow. Patterned interactions in complex systems: Implications for exploration. *Management Science*, 53(7):1068–1085, 2007.

18 T Roughgarden. Computing equilibria: A computational complexity perspective. *Economic Theory*, 42:193–236, 2010.

19 Alexandru Strimbu. Simulating evolution on fitness landscapes represented by valued constraint satisfaction problems. *arXiv:1912.02134*, 2019.

20 S. Wright. The roles of mutation, inbreeding, crossbreeding, and selection in evolution. In *Proc. of the 6th International Congress on Genetics*, pages 355–366, 1932.

A No naive decomposition into arity-5 constraints for P

Let $k > 0$ be an odd integer. One option for reducing the arity of constraint $\widehat{P}$ from Section 5, would be to decompose it into two arity-5 constraints, one between D_{k-1}-and-D_k and one between D_k-and-D_{k+1}. Note that in this case, the value x_k would pick out the column of the D_{k-1}-D_k-constraint, and the row of the D_k-D_{k+1}-constraint. Taken together, these two components would need to result in K from Section 5. Thus, if we want to implement this with two arity-5 constraints between we need to show how to implement K as the sum Q of two rank-1 matrices:

$$Q = \begin{pmatrix} 0 & 1 & 2 \\ 2 & 1 & 0 \\ 0 & 1 & 2 \end{pmatrix} \overset{?}{=} \begin{pmatrix} A_1 \\ B_1 \\ C_1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \begin{pmatrix} A_2 & B_2 & C_2 \end{pmatrix} \tag{31}$$

$$= \begin{pmatrix} A_1 + A_2 & A_1 + B_2 & A_1 + C_2 \\ B_1 + A_2 & B_1 + B_2 & B_1 + C_2 \\ C_1 + A_2 & C_1 + B_2 & C_1 + C_2 \end{pmatrix} = Q \tag{32}$$

We can see that this is impossible to satisfy because on the side of Q we have:

$$Q_{1,1} + Q_{2,2} = A_1 + A_2 + B_1 + B_2 \tag{33}$$

$$= A_1 + B_2 + B_1 + A_2 = Q_{1,2} + Q_{2,1} \tag{34}$$

but on the side of K we have:

$$K_{1,1} + K_{2,2} = 0 + 1 \tag{35}$$

$$\neq 1 + 2 = K_{1,2} + K_{2,1}. \tag{36}$$

Thus, if the even domains contain only a single intermediate state, we cannot decompose the minimisation constraint $\widehat{P}$ into two arity-5 constraints between D_{k-1}-and-D_k and D_k-and-D_{k+1}.

Learning Effect and Compound Activities in High Multiplicity RCPSP: Application to Satellite Production

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

Anouck Chan
DTIS, ONERA, Université de Toulouse, France

Abstract

This paper addresses the High Multiplicity Resource-Constrained Project Scheduling Problem (HM-RCPSP), in which multiple projects are performed iteratively while sharing limited resources. We extend this problem by integrating the learning effect, which makes the duration of some activities decrease when they are repeated. Learning effect can be represented by any decreasing function, allowing us to get flexibility in modeling various scenarios. Additionally, we take composition of activities into consideration for reasoning about precedence and resources in a more abstract way. A Constraint Programming model is proposed for this richer problem, including a symmetry-breaking technique applied to some activities. We also present a heuristic-based search strategy. The effectiveness of these solving approaches is evaluated through an experimentation conducted on data concerning real-world satellite assembly lines, as well as on some adapted literature benchmarks. Obtained results demonstrate that our methods serve as robust baselines for addressing this novel problem (denoted by HM-RCPSP/L-C).

2012 ACM Subject Classification Applied computing → Decision analysis

Keywords and phrases High-multiplicity Project Scheduling, Learning Effect, Compound Activities, Satellite Assembly Line, Constraint Programming, Symmetry Breaking

Digital Object Identifier 10.4230/LIPIcs.CP.2024.18

Supplementary Material *Dataset (PSP-based dataset)*: `https://doi.org/10.57745/ASGLBH` [18]

Funding *Stéphanie Roussel, Anouck Chan*: This work has been partly realised with the support of the French government, in the context of the BPI PSPC project "LiChIE" of the "Programme d'Investissements d'Avenir".
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

In numerous industries, especially in manufacturing, a frequent challenge is to schedule several projects, each demanding multiple executions, within an environment of limited resources. This scheduling problem is known as the High Multiplicity Resource-Constrained Project Scheduling Problem (HM-RCPSP) [8].

[1] Corresponding author

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 18; pp. 18:1–18:25
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This problem arises, for example, when planning the assembly of complex products such as satellites. Traditional satellite manufacturing tends to be handcrafted, and only a few units are produced over a long time horizon. Recently, there has been a growing interest in satellite constellations and large-scale satellite production is now essential to meet demand as highlighted in [7, 9]. We have been working on the production of Earth observation satellites, where most of the activities require human operators. In this case, satellite manufacturers have to adapt their industrial system to cope with very different numbers of units to be produced (from one to dozens) and different deadlines for these satellites. For example, it may be necessary to produce an Earth observation satellite every two months to keep up with constellation development, but a highly specialised sensor satellite can be produced in a dozen months. While each type of satellite has its own assembly process, the resources and machinery used to perform tasks are sometimes identical for satellites of different types. For instance, an Earth observation satellite and a highly dedicated sensor satellite both require aseismic areas in order to install instruments without any interference from the other tasks. As most of these resources and machines are generally very expensive, they are only available in limited quantities in the factory and are shared between the production of all satellite types.

Two features of the satellite assembly line scheduling problem cannot be modelled within the HM-RCPSP frame. Firstly, in addition to the classical precedence relationship over activities, one has also to deal with a composition relationship, which makes possible to express that an activity encompasses other ones (therefore starting and ending with the earliest and latest encompassed activity, respectively). Such an activity, called *compound*, can for instance be used for representing the “installation of a mirror on the satellite”, which is decomposed into several atomic activities such as “fixing the mirror”, “verifying the mirror alignment”, etc. The interest for composition is twofold. It allows us to express the precedence relationship in a more compact way, and to indicate that a resource is consumed globally (the resource becoming unavailable for the whole duration of the compound activity). For instance, a bench test may be required for the overall mirror installation, expressing not only that each activity encompassed in this activity may use the bench test but also that the latter cannot be used by any other activity until the mirror installation is finished.

Secondly, in some manufacturing environments, the time required to perform an operation usually fluctuates over time. As a production system operates, workers acquire expertise or improve the manufacturing process, become proficient in required actions, learn tool utilization, and enhance their interaction with the supply chain, among other factors. Consequently, the overall system becomes more efficient. For example, the time needed to assemble the 10th iteration of a product may be only half of the time taken for the initial one. This phenomenon is recognized as the *learning effect*, and the variation in duration based on the repetition of the same operation is called a *learning curve* [25]. In the case of complex products such as satellites or aircrafts, such a phenomenon can have a significant impact on the medium and long-term production planning of the factory. Interestingly, taking learning effect into account permits a more realistic estimation of the dates at which products will be ready. It can also be useful to predict potential additional investment in terms of resources to meet delivery goals. However, incorporating learning effect makes HM-RCPSP more intricated because projects are inter-dependent in terms of activities duration.

The objective of this paper is to address the High Multiplicity RCPSP with Learning effect and Compound activities, denoted HM-RCPSP/L-C. More precisely, the contributions of this paper are:

- HM-RCPSP/L-C is formally defined, notably by introducing a generic learning effect component;
- a symmetry-breaking method is proposed, while proving its correctness;
- a Constraint Programming (CP) model and a heuristic-based search strategy are introduced;
- new academic benchmarks are proposed (and made publicly available);
- an experimentation is conducted on these benchmarks as well as on industrial satellite assembly line benchmarks, showing the potential of the CP approach (with and without symmetric-breaking constraints) which is compared to some other approaches.

The paper is structured as follows. After presenting some related works in Section 2, HM-RCPSP/L-C is formally introduced in Section 3. Then, symmetry breaking is addressed in Section 4. Two main solving approaches for HM-RCPSP/L-C are introduced in Section 5, and the results of the experiments we have performed are presented in Section 6. Finally, we conclude and discuss some perspectives in Section 7.

2 Related Works

The RCPSP problem is a classical problem in combinatorial optimization, for which plenty of solving approaches have been proposed in the literature [2]. As described in [12], many extensions have been proposed over the years. In this section, we first focus on extensions that address either the presence of several projects or their repetition in a context of shared resources. Then, we present some works in which scheduling and learning effects are simultaneously handled.

Multi Project and Repetition in Scheduling. In a recent survey [22], the authors present several extensions of the Resource-Constrained Multi-Project Scheduling Problem (RCMPSP). Such extensions can address activities features (e.g. preemption or uncertainty), activities relationships (e.g. concurrency or time lags), projects and resource features (e.g. renewability or availability). Because of the learning effect, the projects we consider in this study are inter-dependent, which prevents the HM-RCPSP/L-C to be seen as a special case of RCMPSP.

In [6], the authors address the cyclic RCPSP (a single project is repeated infinitely) through a Discrete Time Constraint Programming (DT-CP) approach. The objective is to find a pattern that interleaves several repetitions of the project in order to use resources optimally. In the context of construction projects [10], identical activities are repeated a given number of times. In [8], the authors address the HM-RCPSP/max in which several projects are repeated and share common resources. Projects can be linked through a generalized precedence relationship which specifies maximum time-lags between activities. They propose a symmetry breaking method for identical projects of the same class.

Learning Effect and Scheduling. The learning effect phenomenon was initially reported in [23]. The author defines a log-linear learning model that we use in our experiments. Various learning curves have been proposed since then [25].

There are two different approaches to learning in scheduling environments: (i) *position-based approach*, meaning that learning is effected by the pure number of times an activity has been completed; and (ii) *sum-of-processing-time approach* which takes into account the processing time of all same activities processed so far [5]. Some works have studied under which conditions the problem becomes polynomial [4, 3, 26]. In [1], a position-based scheduling problem with repetitive projects is studied using a two-stage approximation approach. It consists of identical projects that can be executed in parallel while respecting the resource capacity, and each activity requires exactly one unit of one specific resource type.

Staff allocation has been studied in [24] and [21], respectively with a position based and a sum-of-processing based learning effect modelling. Learning effect has also been studied in the case of Discrete Time/Resource Trade-off Problem (DTRTP), as presented in [20, 19]. DTRTP is a sub-problem of the multi-mode RCPSP, where the duration of each activity depends on the amount of workers or resources assigned to it. Finally, in [13], the authors introduce four DT-CP formulations for the RCPSP with position-based learning effect, and provide an empirical comparison of their scheduling and lower bounding performance. In their study, the effect of learning is modeled by having two duration values for each activity: a nominal and a reduced one, and there is no repetition factor.

To conclude this section, to the best of our knowledge, there is no work in the literature addressing simultaneously multi-projects sharing resources, with compound activities and whose activity duration depends on learning effect.

3 HM-RCPSP/L-C

The components (data) required to define an instance of the high-multiplicity resource-constrained project scheduling problem with compound activities and learning effect (HM-RCPSP/L-C) are formally described in this section.

3.1 Problem Inputs

An instance of HM-RCPSP/L-C is a tuple composed of the following elements:

- $\mathcal{C}$ is the set of project classes (or categories);
- for each class $c \in \mathcal{C}$, $\mathcal{I}_c$ is the set of projects (or instances) of c that have to be realized. For each project $p \in \mathcal{I}_c$, due_p represents its due date, *i.e.* the date at which it should be finished;
- for each class $c \in \mathcal{C}$, $\mathcal{A}_c$ represents the set of non-preemptive activities that have to be realized for each project of c. Activities $\mathcal{A}_c$ can be partitioned into two sets $\mathcal{A}_c^A$ and $\mathcal{A}_c^C$ that respectively represent atomic activities and compound activities. Intuitively, a compound activity can be seen as a group of activities. Such an activity spans over all activities in the group;
- for each class $c \in \mathcal{C}$, $\mathcal{H}_c \subseteq \mathcal{A}_c^C \times \mathcal{A}_c$ is the composition relation of c. If a is a compound activity and b an activity, then $(a, b) \in \mathcal{H}_c$ expresses that b is encompassed by a (or a child activity of a). Note that it is possible to have (a, b) if b is a compound activity;
- for each class $c \in \mathcal{C}$ and for each atomic activity $a \in \mathcal{A}_c^A$, we assume the existence of a monotonically decreasing duration function, denoted as $dur_a : \mathbb{N}^+ \to \mathbb{N}^+$, which returns the duration of the activity based on the number of times it has been completely executed before;
- for each class $c \in \mathcal{C}$, $\mathcal{P}_c$ is a precedence relationship between activities in $\mathcal{A}_c$. If $(a, b) \in \mathcal{P}_c$, then activity a must be finished before the start of activity b;
- $\mathcal{R}$ is the set of cumulative and renewable resources. For each resource $r \in \mathcal{R}$, $capa_r$ denotes its capacity.
- for each resource $r \in \mathcal{R}$, for each class $c \in \mathcal{C}$, for each activity $a \in \mathcal{A}_c$, $cons_{r,a}$ denotes the amount of resource r consumed by a;
- $H \in \mathbb{N}^+$ is the maximum time horizon.

We denote a^p the instance of activity $a \in \mathcal{A}_c$ that has to be realized for project $p \in \mathcal{I}_c$ and $c \in \mathcal{C}$.

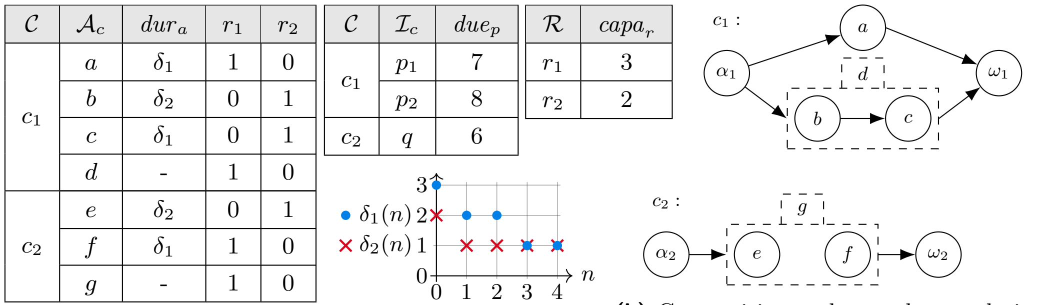
Assumptions. An instance of HM-RCPSP/L-C is said to be well-formed if and only if the following assumptions hold. Note that unless said otherwise, all instances considered in the paper are supposed to be well-formed.

1. Projects and activities are associated with exactly one class.
2. The duration function associated with each activity is monotonically decreasing. Formally, $\forall c \in \mathcal{C}$, $\forall a \in \mathcal{A}_c^A$, $\forall n \geq 0$, $dur_a(n+1) \leq dur_a(n)$.
3. For each class $c \in \mathcal{C}$, we suppose that the set of atomic activities $\mathcal{A}_c^A$ contains two virtual activities α_c and ω_c that respectively represent the start and end activities for a project. Such activities do not consume any resource, they are not encompassed by any compound activity and their duration is null. Moreover, α_c precedes (resp. ω_c follows) all other activities in $\mathcal{A}_c$.
4. For each class $c \in \mathcal{C}$, the graph induced by the compound/atomic activities relationship is a forest, *i.e.* each node has at most one parent and there is no cycle. Formally, such a graph $\mathcal{G}_c^{\mathcal{H}}$ is composed of a set of nodes $N_c^{\mathcal{H}}$ and a set of arcs $Arcs_c^{\mathcal{H}}$ that are defined as follows. $N_c^{\mathcal{H}}$ is composed of one node n_a for each activity a in $\mathcal{A}_c$. $Arcs_c^{\mathcal{H}}$ is composed of one arc (n_a, n_b) for each pair of activities $(a,b) \in \mathcal{H}_c$.
5. For each class $c \in \mathcal{C}$, for each tree in the forest $\mathcal{G}_c^{\mathcal{H}}$, and for each path from the root to an arbitrary leaf, the resource capacities are sufficient to execute all activities on this path simultaneously.
6. For each class $c \in \mathcal{C}$, the graph $\mathcal{G}_c^{\mathcal{P}}$ induced by the precedence relationship is acyclic. To build such a graph, we first define an atomic version of the precedence relationship, denoted $\mathcal{P}_c^A$, in which all precedences of $\mathcal{P}_c$ are transposed to atomic activities. Formally, for all a, b in $\mathcal{A}_c^A$, (a,b) belongs to $\mathcal{P}_c^A$ if and only if there exist two activities d and e in $\mathcal{A}_c$ such that: 1. $(d,e) \in \mathcal{P}_c$, 2. there exists a path in $\mathcal{G}_c^{\mathcal{H}}$ from n_d to n_a and 3. a path from n_e to n_b. Then, the set of nodes for $\mathcal{G}_c^{\mathcal{P}}$ is $N_c^{\mathcal{P}} = \{v_a \mid a \in \mathcal{A}_c^A\}$, that contains one node for each atomic activity. The set of arcs is $Arcs_c^{\mathcal{P}} = \{(v_a, v_b) \mid (a,b) \in \mathcal{P}_c^A\}$ and contains one arc for each pair in the atomic precedence relationship.

▶ **Example 1.** Figure 1 illustrates a toy example instance of HM-RCPSP/L-C. There are two classes, c_1 and c_2. The activities of each class are described on the left table of Figure 1a. For each project of class c_1, 4 activities (+2 dummy activities α_1 and ω_1) must be performed. Activities a, b, c, α_1 and ω_1 are atomic ($\mathcal{A}_{c_1}^A = \{\alpha_1, \omega_1, a, b, c\}$). Activity a's duration is given by function δ_1 described in the lower right corner: if no instance of activity a has been completed yet (i.e. $n = 0$), then its duration is 3, if one execution has been fully completed before starting a then its duration is equal to 2, and so on. Activity d is compound ($\mathcal{A}_{c_1}^C = \{d\}$) and its children are b and c, as represented on the upper graph of Figure 1b. This graph also represents the precedence relationship $\mathcal{P}_{c_1}$. For instance, (α_1, d) and (b, c) both belong to $\mathcal{P}_{c_1}$. r_1 and r_2 are the two resources and respectively have a capacity equal to 3 and 2. Both atomic and compound activities can consume these resources. For instance, $cons_{r_1,d} = 1$ and $cons_{r_2,b} = 1$. Two projects, p_1 and p_2, of class c_1 and one project, q, for class c_2 must be realized. Due dates for these projects are respectively equal to 7, 8 and 6.

3.2 Schedule, Solution and Optimality

Schedule. A schedule σ for an HM-RCPSP/L-C instance is defined through the assignment of a start date to each atomic activity of each project of each class. Formally, if a is an atomic activity, the start date of a^p in σ is denoted $start^\sigma(a^p)$.



$\mathcal{C}$	$\mathcal{A}_c$	dur_a	r_1	r_2
c_1	a	δ_1	1	0
	b	δ_2	0	1
	c	δ_1	0	1
	d	-	1	0
c_2	e	δ_2	0	1
	f	δ_1	1	0
	g	-	1	0

$\mathcal{C}$	$\mathcal{I}_c$	due_p
c_1	p_1	7
	p_2	8
c_2	q	6

$\mathcal{R}$	$capa_r$
r_1	3
r_2	2

(a) Example of classes, projects, activities and resources.

(b) Composition and precedence relationships.

Figure 1 Toy example of a HM-RCPSP/L-C instance.

The end date of a^p, where a is an atomic activity and p a project, is denoted $end^\sigma(a^p)$. Because of the learning effect, a^p's duration depends on the number of times activity a has already been completed for other projects in the same class at the start of a^p. The function nc^σ, applied to activity a and time step t, returns the number of times that a has been completed at t. Formally,

- $end^\sigma(a^p) = start^\sigma(a^p) + dur_a(nc^\sigma(a, start^\sigma(a^p)))$.
- $nc^\sigma(a,t) = \sum_{q \in \mathcal{I}_c \setminus \{p\}} (end^\sigma(a^q) \leq t)$

In practice, the value of these functions can be determined for a given schedule σ by ordering the activities executions using their start dates. The start dates of two virtual activities α_c and ω_c can be then defined as follows:

- $start^\sigma(\alpha_c^p) = \min_{a \in \mathcal{A}_c^A} start^\sigma(a^p)$
- $start^\sigma(\omega_c^p) = \max_{a \in \mathcal{A}_c^A} end^\sigma(a^p)$

Start and end dates of compound activities can be deduced from those of atomic ones: the start date of a compound activity a (resp. the end date) is equal to the minimum start date (resp. maximum end date) of activities encompassed by a. For readability, we also use functions $start^\sigma(a^p)$ and $end^\sigma(a^p)$ for denoting start and end dates of compound activities.

Solution. A schedule σ is a solution for an HM-RCPSP/L-C instance if and only if the precedence constraints (Equation 1) are satisfied and the resources capacity is respected through the entire time horizon (Equation 2).

$$\forall c \in \mathcal{C}, \forall (a,b) \in \mathcal{P}_c, \forall p \in \mathcal{I}_c, \qquad end^\sigma(a^p) \leq start^\sigma(b^p) \tag{1}$$

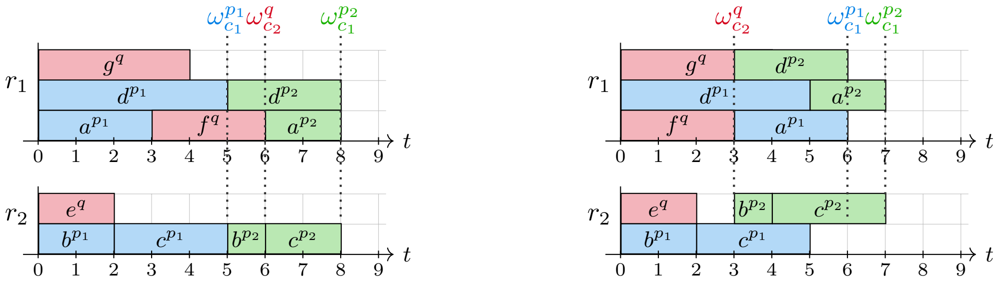
$$\forall r \in \mathcal{R}, \forall t \in [\![0, H]\!], \qquad \left(\sum_{\substack{c \in \mathcal{C}, a \in \mathcal{A}_c, p \in \mathcal{I}_c \\ start^\sigma(a^p) \leq t < end^\sigma(a^p)}} cons_{r,a} \right) \leq capa_r \tag{2}$$

Optimal Solution. A solution σ is said to be optimal if it minimizes (in lexicographic order) the two following optimization criteria: (1) the sum of tardiness for projects (Equation 3), and (2) the makespan (Equation 4).

$$\sum_{c \in \mathcal{C}, p \in \mathcal{I}_c} \max\{0, end^\sigma(\omega_c^p) - due_p\} \tag{3}$$

$$\max_{c \in \mathcal{C}, p \in \mathcal{I}_c} end^\sigma(\omega_c^p) \tag{4}$$

▶ **Example 2.** Figures 2a and 2b respectively illustrate two solutions σ_1 and σ_2. We have $start^{\sigma}(c^{p_1}) = 2$, i.e. the execution of c in project p_1, starts at time 2 in solution σ_1. As $\delta_1(0) = 3$, c^{p_1} lasts 3 time units and $end^{\sigma}(c^{p_1}) = 5$. Still in σ_1, at the start of c^{p_2} (time step 6), one execution of c is completed so duration of c^{p_2} is equal to 2. Note that in σ_2, when starting activity c^{p_2}, the execution of c^{p_1} is not completed yet, so the duration of c^{p_2} is equal to 3.
As the compound activity g^q spans activities e^q and f^q, in solution σ_1, this makes g^q start at time 0 and end at time 6. However, in σ_2, g^q ends at time 3. In both cases, the resource r_1 is consumed all over g^q's duration.
Both solutions satisfy projects due dates. For instance, $end^{\sigma_1}(\omega^{p_1}_{c_1})$ is equal to 5, which is less than 7. However, the makespan of σ_2 is lower than the makespan of σ_1, which makes σ_2 a better solution.



(a) Solution σ_1 with a makespan equal to 8.

(b) Solution σ_2 with a makespan equal to 7.

Figure 2 Illustration of Example 2 – Due dates are satisfied in both solutions.

4 Symmetry Breaking

In this section, we establish the existence of symmetries between identical activities from different projects within the same class.

First, because of the learning effect, an activity a that starts in project q later than in project p can finish earlier in q. This is illustrated on the left part of Figure 3.

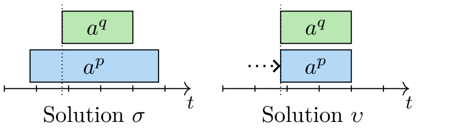


Figure 3 Illustration of Lemma 4's proof: σ is not start-end-consistent but υ is.

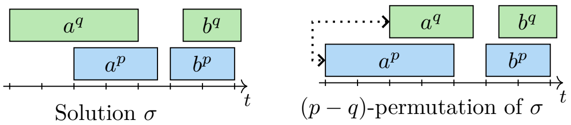


Figure 4 Illustration of a $(p-q)$-*permutation*: a^p and a^q are swapped.

We address that particular case through Definition 3 and Lemma 4.

▶ **Definition 3** (Start-end-consistency). *A solution σ is* start-end-consistent *if for any class $c \in \mathcal{C}$, any atomic activity $a \in \mathcal{A}^A_c$, and any two projects $p, q \in \mathcal{I}_c$, we have:*

$$start^{\sigma}(a^p) < start^{\sigma}(a^q) \iff end^{\sigma}(a^p) < end^{\sigma}(a^q)$$

▶ **Lemma 4.** *If a solution σ is not start-end-consistent then there exists a start-end-consistent solution υ that is equivalent or better than σ with respect to the two optimization criteria.*

Sketch of proof. Let a be an atomic activity and p and q be two projects for which a solution σ is not start-end-consistent as in Figure 3. We can build a schedule υ that is equal to σ except that the start date of a^p is delayed until the start date of a^q. All precedence and resource consumption constraints will be naturally satisfied in υ (they are relaxed). ◀

Note that the transformation for obtaining a start-end consistent solution (proof of Lemma 4) can delay some activities. However, as any activity delayed by the transformation also ends earlier, criteria can only be improved. From this point onward, we only consider start-end-consistent solutions. In the following, we define specific moves on schedules, that we call permutations, and show that they preserve the satisfaction of precedence and resource constraints.

▶ **Definition 5** ((p,q)-permutation). *For any solution σ, any class $c \in \mathcal{C}$ and any two projects $p, q \in \mathcal{I}_c$, a schedule π is a* (p,q)-permutation *of σ if for all atomic activities $a \in \mathcal{A}_c^A \setminus \{\alpha_c, \omega_c\}$, we have:*

- $\forall o \in \mathcal{I}_c \setminus \{p,q\}$, *$start^\pi(a^o) = start^\sigma(a^o)$,*
- *if $start^\sigma(a^p) > start^\sigma(a^q)$, then $start^\pi(a^q) = start^\sigma(a^p)$ and $start^\pi(a^p) = start^\sigma(a^q)$,*
- *else $start^\pi(a^p) = start^\sigma(a^p)$ and $start^\pi(a^q) = start^\sigma(a^q)$.*

The operation that consists in swapping start dates of a specific atomic activity, excluding the source and the sink, between two projects is called a swap.

As illustrated in Figure 4, a (p,q)-permutation consists in swapping all atomic activities of p that start after their counterpart activities in q. Because the start date of compound activities and source and sink depend only on the start dates of atomic activities, a corollary of Definition 5 is that all activities in p start before their counterpart activities in q. Note also that swapping the start dates of atomic activities a^p and a^q also swaps their duration and their end dates.

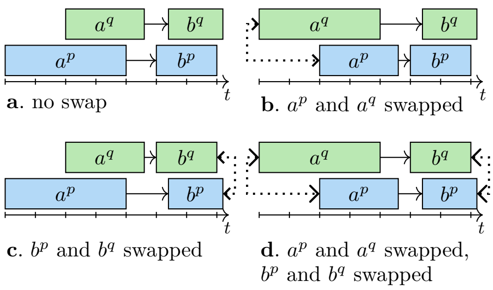
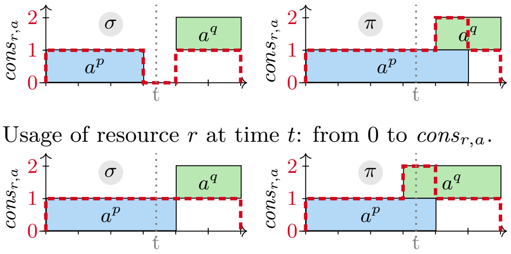


Figure 5 Four precedence cases considered in the proof of Lemma 6.



(a) Usage of resource r at time t: from 0 to $cons_{r,a}$.

(b) Usage of resource r at time t: from $cons_{r,a}$ to $2cons_{r,a}$.

Figure 6 Two impossible cases considered in proof of Lemma 7 when $a \in \mathcal{A}_c^C$.

▶ **Lemma 6.** *Given a start-end-consistent solution σ, for any class $c \in \mathcal{C}$ and any two projects $p, q \in \mathcal{I}_c$, the (p,q)-permutation π of σ is precedence-feasible, i.e. it satisfies Equation 1.*

Sketch of proof. As inferred by Assumption 5, we consider only precedence on atomic activities. If π is not precedence-feasible, there exist two atomic activities a and b such that a precedes b and such that $end^\pi(a^p) > start^\pi(b^p)$ or $end^\pi(a^q) > start^\pi(b^q)$ hold. As σ is precedence-feasible, then $end^\sigma(a^p) \le start^\sigma(b^p)$ and $end^\sigma(a^q) \le start^\sigma(b^q)$ both hold. We then consider all possible cases in terms of exchange, as illustrated in Figure 5, where the dashed lines correspond to an exchange of activities in π. In all cases, if σ is precedence-feasible, then so is π. ◀

▶ **Lemma 7.** *Given a start-end-consistent solution σ, for any class $c \in \mathcal{C}$ and any two projects $p, q \in \mathcal{I}_c$, the (p, q)-permutation π of σ is resource-feasible, i.e. it satisfies Equation 2.*

Sketch of proof. The resource feasibility is easy to prove in the case of atomic activities. In fact, when swapping activities, the consumption of resources does not change. However, in the case of compound activities, it is possible that some children activities are swapped and some other not, meaning that duration of compound activities can change when performing a (p, q)-permutation. In the following, we consider all the cases and show that even with compound activities, the permutation stays resource-feasible.

We define a function $\gamma_{r,a,t}(p, \sigma)$ that represents how much the activity a executed for project p consumes a resource r at a given time step t in σ, i.e. $\gamma_{r,a,t}(p, \sigma) = cons_{r,a}$ if $t \in T^{\sigma}_{p,a}$, 0 otherwise. The consumption of a resource r at each time step t in a schedule σ is given by the formula $\sum_{c\in\mathcal{C},p\in\mathcal{I}_c,a\in\mathcal{A}_c} \gamma_{r,a,t}(p, \sigma)$. The proof is divided into several steps.

1. We show that if π is not-resource feasible, there exists a time-step t and a resource r such that $\sum_{c\in\mathcal{C},p\in\mathcal{I}_c,a\in\mathcal{A}_c} \gamma_{r,a,t}(p, \pi) - \gamma_{r,a,t}(p, \sigma) > 0$. We consider this specific resource r and this time step t.
2. We prove that the resource capacity violation in π comes only from projects p and q.
3. Let c be the class associated with p and q and a an activity in $\mathcal{A}_c$, we show that the consumption of r induced by both activities a^p and a^q is equivalent in π and σ. Formally, $\gamma_{r,a,t}(p, \pi) - \gamma_{r,a,t}(p, \sigma) + \gamma_{r,a,t}(q, \pi) - \gamma_{r,a,t}(q, \sigma) = 0$.
   a. If a is an activity atomic, the usage of r unchanged.
   b. If a is a compound activity then we consider two cases (Figure 6): (i) if activities a^p and a^q do not use r at time t in σ, but at least one of them uses it in π, then we have $\gamma_{r,a,t}(p, \sigma) = \gamma_{r,a,t}(q, \sigma) = 0$ and $\gamma_{r,a,t}(p, \pi) + \gamma_{r,a,t}(q, \pi) \geq cons_{r,a}$ (Figure 5a) ; (ii) if exactly one activity (a^p or a^q) uses r at time t in σ and they both use it in π, then $\gamma_{r,a,t}(p, \sigma) + \gamma_{r,a,t}(q, \sigma) = cons_{r,a}$ and $\gamma_{r,a,t}(p, \pi) + \gamma_{r,a,t}(q, \pi) = 2cons_{r,a}$(Figure 5b). We show that both cases lead to a contradiction.

All cases imply a contradiction, meaning that π is resource-feasible. ◀

▶ **Lemma 8.** *Given a start-end-consistent schedule σ, for any class $c \in \mathcal{C}$ and any two projects $p, q \in \mathcal{I}_c$, the makespan of the (p, q)-permutation π of σ is equivalent to that of σ.*

Complete proof. Let $\epsilon^{\sigma}_{p,q} = \max_{a\in\mathcal{A}_c}(end^{\sigma}(a^p), end^{\sigma}(a^q))$ be the date at which both projects p and q are finished in σ. In π, even if there is a swap between activities, the date when p and q are both finished does not change, *i.e.* $\epsilon^{\pi}_{p,q} = \epsilon^{\sigma}_{p,q}$. As the timing of all other projects in π remains the same as in σ, the makespan of π must be equivalent to that of σ. ◀

For the tardiness, only some (p, q)-permutation such that $due_p < due_q$ allows to get a lower or equal tardiness. Therefore, we define a preorder on projects consistent with due dates and show that it preserves or improves tardiness. Note that we do not define a unique preorder $\prec$ based on uniquely considering $p \prec_c q$ when $due_p < due_q$, as we allow the user to set arbitrarily $p \prec_c q$ when $due_p = due_q$, provided that the relation remains well-formed. This is somewhat distantly related to the arbitrary choices made when posting the global constraint Precedence [17] (for example, between identical projects, you can choose one to precede the other, as done in [8]).

▶ **Definition 9** (Due-date consistent strict preorder)**.** *For any class $c \in \mathcal{C}$, a strict order $\prec_c$ is due-date consistent for c if $\forall p, q \in \mathcal{I}_c$ such that $due_p < due_q$, then $p \prec_c q$.*
A relation $\prec$ is a due-date consistent strict preorder if it is the union of due-date consistent strict preorders $\prec_c$ for each class $c \in \mathcal{C}$.

▶ **Example 10.** Let us consider an instance with two classes, c_1 and c_2, and for which projects p_1, p_2 and p_3, and q_1 and q_2 are respectively associated, and such that $due_{p_1} = 7$, $due_{p_2} = due_{p_3} = 10$, $due_{q_1} = 8$ and $due_{q_2} = 6$. Then $\prec_{c_1} = \{(p_1, p_2), (p_1, p_3), (p_3, p_2)\}$ is a due-date consistent strict preorder for c_1. Considering (p_2, p_3) instead of (p_3, p_2) would also result in a due-date consistent strict preorder. $\prec_{c_2} = \{(q_2, q_1)\}$ is the only due-date consistent strict preorder for c_2. $\prec = \prec_{c_1} \cup \prec_{c_2}$ is a due-date consistent strict preorder for this instance.

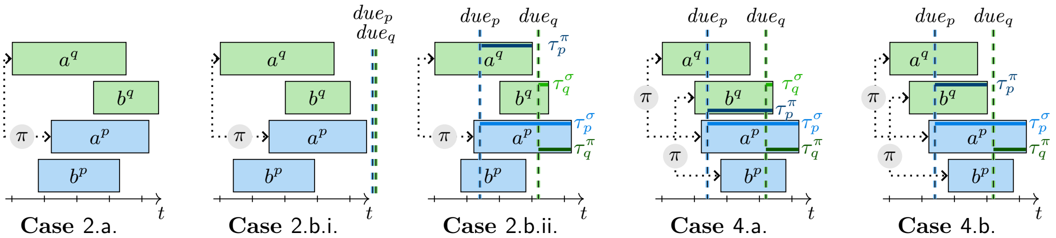
In the following, for a class $c \in \mathcal{C}$, a project $p \in \mathcal{I}_c$ and a schedule σ, let τ_p^σ denote the tardiness of project p. Formally, $\tau_p^\sigma = \max\{0, end^\sigma(\omega_c^p) - due_p\}$. τ^σ denotes the sum of tardiness of all projects, as expressed in Equation 3.

▶ **Lemma 11.** *Given a start-end-consistent solution σ, a due-date consistent strict preorder $\prec$, a class $c \in \mathcal{C}$ and two projects $p, q \in \mathcal{I}_c$. If $p \prec q$ and π is the (p, q)-permutation of σ, then $\tau^\pi \leq \tau^\sigma$, i.e. the tardiness of π is less than or equal to the tardiness of σ.*

Sketch of proof. Except from projects p and q, all other projects activities remain the same in σ and π. Thus, the difference between tardiness of both schedules comes from the difference tardiness of projects p and q in σ and in π. Intuitively, p always ends earlier (its tardiness is reduced) but q might end later in π than in σ. We therefore have to show that $\tau_p^\pi + \tau_q^\pi \leq \tau_p^\sigma + \tau_q^\sigma$ or $\tau_q^\pi \leq \tau_q^\sigma$. We suppose that a^p and b^q are respectively the latest activities for p and q in σ. Cases considered in the following are illustrated in Figure 7.

1. If both activities a and b are not swapped between projects p and q in π, so b^q still concludes q in π, which implies $\tau_q^\pi = \tau_q^\sigma$.
2. If a^p and a^q are swapped, but b^p and b^q are not then $\tau_p^\pi \leq \tau_p^\sigma$.
   a. If $end^\sigma(a^p) \leq end^\sigma(b^q)$ then we can show that b^q still concludes q in π, so $\tau_q^\pi = \tau_q^\sigma$.
   b. Else, a^q concludes q in π and we have to consider several following situations.
      i. If $end^\sigma(a^p) \leq due_q$, then $\tau_q^\pi = \tau_q^\sigma = 0$.
      ii. If $end^\sigma(a^p) > due_q$, then $\tau_q^\pi = end^\sigma(a^p) - due_q$ and $\tau_p^\sigma = end^\sigma(a^p) - due_p$. Regardless of the relative positions of $end^\sigma(b^q)$, due_p and due_q, we show that $\tau_p^\pi + \tau_q^\pi \leq \tau_p^\sigma + \tau_q^\sigma$.
3. It is impossible to have b^p and b^q being swapped but not a^p and a^q, because a^p and b^q are respectively the latest activities for p and q in σ.
4. If both a^p and a^q are swapped, and so are b^p and b^q, then $end^\sigma(a^q) \leq end^\sigma(b^q) < end^\sigma(b^p) \leq end^\sigma(a^p)$ then b^p and a^q are respectively the latest activities for p and q in π. We prove that $\tau_p^\pi + \tau_q^\pi \leq \tau_p^\sigma + \tau_q^\sigma$ in several cases: a. both p and q are late in σ, b. p is late and q is early in σ and c. both p and q are early in σ.

In those all cases, $\tau_p^\pi + \tau_q^\pi \leq \tau_p^\sigma + \tau_q^\sigma$, which proves that $\tau^\pi \leq \tau^\sigma$. ◀



Case 2.a. $\tau_p^\pi \leq \tau_p^\sigma$, $\tau_q^\pi = \tau_q^\sigma$

Case 2.b.i. $\tau_p^\pi \leq \tau_p^\sigma$, $\tau_q^\pi = \tau_q^\sigma$

Case 2.b.ii. $\tau_p^\pi + \tau_q^\pi \leq \tau_p^\sigma + \tau_q^\sigma$

Case 4.a. $\tau_p^\pi + \tau_q^\pi \leq \tau_p^\sigma + \tau_q^\sigma$

Case 4.b. $\tau_p^\pi + \tau_q^\pi \leq \tau_p^\sigma$

Figure 7 Illustration of cases considered in Lemma 11's proof.

We now define a $\prec$-permutation of σ in which (p,q)-permutations are done for all p,q such that $p \prec q$ and show that if σ is a solution then the $\prec$-permutation is also a solution better or equivalent to σ.

▶ **Definition 12** ($\prec$-permutation). *Given a solution σ and a due-date consistent strict preorder $\prec$, a schedule π is a $\prec$-permutation of σ if, for any class $c \in \mathcal{C}$, and any two projects p, q in $\mathcal{I}_c$ such that $p \prec q$, π is a (p,q)-permutation of σ.*

▶ **Lemma 13.** *Given a start-end-consistent solution σ and a due-date consistent strict preorder $\prec$, the $\prec$-permutation π of σ is a solution with a makespan equivalent to that of σ and a tardiness as least as good as that of σ.*

Complete proof. Let π be a $\prec$-permutation of a schedule σ. π can be built from σ through several schedules $\upsilon_0, \ldots, \upsilon_n$ where $\upsilon_0 = \sigma$, $\upsilon_n = \pi$ and $\forall i \in [\![1,n]\!]$, υ_i is a (p,q)-permutation of υ_{i-1} with p and q two projects such that $p \prec q$. According to Lemmas 6, 7, 8 and 11, each of these permutations is a solution with an equivalent makespan and equivalent or lesser tardiness. ◀

This result means that for finding a solution σ to an HM-RCPSP/L-C instance, one may consider a due date consistent preorder $\prec$ and look for solutions in which for all p and q such that $p \prec q$, for all activity $a \in \mathcal{A}_c$, $start^{\sigma}(a^p) \leq start^{\sigma}(a^q)$.

5 Solving HM-RCPSP/L-C

In this section we present the two main approaches we have developed for solving HM-RCPSP/L-C, namely a CP approach and a heuristic-based search approach. They can be combined, but the exact description of these combinations is left to Section 6.

5.1 Building a CP Model

Regarding the CP approach, in this paper, we use the interval based variable formulation of Optimization Programming Language (OPL) associated with CP Optimizer [16]. An interval variable allows us to represent an activity along with its variable starting date, its variable duration, its mandatory or optional presence in the computed schedule, its earliest start and latest end dates. The start date, end date and duration of an interval variable are respectively accessible through functions startOf, endOf and lengthOf. For each class $c \in \mathcal{C}$, for each project $p \in \mathcal{I}_c$, for each activity $a \in \mathcal{A}_c$, we consider the following variables:

- $\mathbf{itv}_{a,p} \in [\![0, H]\!]$ is an interval variable for the execution of activity a^p;
- $\mathbf{n}_{a,p} \in [\![0, |\mathcal{I}_c|]\!]$ is an integer variable representing the number of times activity a of class c was completed before the start of this instance of a, corresponding to $nc^{\sigma}(a, start^{\sigma}(a^p))$ defined in Section 3.

Constraints and criteria are encoded as follows:

$$minimize \left(\sum_{\substack{c\in\mathcal{C},p\in\mathcal{I}_c\\ a\in\mathcal{A}_c}} \max(0, \mathsf{endOf}(\mathbf{itv}_{a,p}) - due_p), \max_{\substack{c\in\mathcal{C},p\in\mathcal{I}_c\\ a\in\mathcal{A}_c}} \mathsf{endOf}(\mathbf{itv}_{a,p}) \right) \quad (5)$$

such that:

$$\forall r \in \mathcal{R}, \quad \textstyle\sum_{\substack{c\in\mathcal{C},p\in\mathcal{I}_c,\\ a\in\mathcal{A}_c}} \mathsf{pulse}(\mathbf{itv}_{a,p}, cons_{r,a}) \leq capa_r \quad (6)$$

$$\forall c \in \mathcal{C}, \forall p \in \mathcal{I}_c, \forall (a,b) \in \mathcal{P}_c, \quad \mathsf{endBeforeStart}(\mathbf{itv}_{a,p}, \mathbf{itv}_{b,p}) \quad (7)$$

$$\forall c \in \mathcal{C}, \forall p \in \mathcal{I}_c, \forall a \in \mathcal{A}_c^C, \quad \mathsf{span}(\mathbf{itv}_{a,p}, \{\mathbf{itv}_{b,p} | (a,b) \in \mathcal{H}_c\}) \quad (8)$$

$$\forall c \in \mathcal{C}, \forall p \in \mathcal{I}_c, \forall a \in \mathcal{A}_c^A, \quad \mathbf{n}_{a,p} = \textstyle\sum_{q\in\mathcal{I}_c} (\mathsf{endOf}(\mathbf{itv}_{a,q}) \leq \mathsf{startOf}(\mathbf{itv}_{a,p})) \quad (9)$$

$$\forall c \in \mathcal{C}, \forall p \in \mathcal{I}_c, \forall a \in \mathcal{A}_c^A, \quad \mathsf{lengthOf}(\mathbf{itv}_{a,p}) = dur_a(\mathbf{n}_{a,p}) \quad (10)$$

The lexicographic order of criteria is formalized in Equation (5). Constraints (6) are cumulative global constraints with respect to resource capacities, expressed through the OPL pulse function. Constraints (7) guarantee that the precedence relationships are satisfied (OPL function endBeforeStart). Constraints (8) enforce compound activities to span over their children. Constraints (9) encode the computation of $nc^{\sigma}(a,)$ at the time the atomic activity a^p starts. The duration with learning effect are pre-computed and atomic activities are assigned their duration through *Element* Constraints (10). In addition to the original constraints mentioned above, symmetry breaking constraints are defined as follows:

$$\forall c \in \mathcal{C}, \forall p, q \in \mathcal{I}_c \ s.t. \ p \prec_c q, \forall a \in \mathcal{A}_c^A, \mathsf{startBeforeStart}(\mathbf{itv}_{a,p}, \mathbf{itv}_{a,q}) \tag{11}$$

5.2 Using Squeaky Wheel Optimization

The second main approach we use to address HM-RCPSP/L-C is called Squeaky Wheel Optimisation (SWO) [14]. This technique consists of constructing an initial solution using a greedy algorithm. It is followed by an analysis to identify areas for improvement, i.e. parts of the solution that might improve the objective function score if they are modified. From this analysis, new priorities are generated, which modifies the sequence in which the greedy algorithm constructs subsequent solutions. This iterative process is repeated until a predefined limit is reached.

More precisely, the greedy approach for constructing a solution of HM-RCPSP/L-C consists in a Parallel Schedule Generation Scheme. Starting with an empty schedule, it iterates chronologically over the time horizon and fills the schedule with eligible activities. For a given time step t, an atomic activity a is eligible if Equations (1) and (2) are satisfied when a starts at time t (taking the compound activity of a into account). Using a choice heuristic, an eligible activity is inserted at time t and the set of eligible activities is updated. When this set becomes empty, the process is repeated for the next time step $t + 1$. The schedule is complete when all activities have been inserted. Given a set of eligible activities $\mathcal{E}$, the choice heuristic follows several steps.

1. Because of the symmetry proof, we first remove from $\mathcal{E}$ all the activities a^q for which there exists a project p of the same class as q such that p precedes q in the due-date consistent strict preorder ($p \prec q$) and a^p has not been inserted yet ($a^p \in \mathcal{E}$). The resulting set of eligible activities is denoted $\mathcal{E}_{SB}$.
2. There are two possible cases: a) if there is no late project at the moment, all available projects with activities in $\mathcal{E}_{SB}$ will be considered as candidates; b) otherwise, only the late projects will be considered as candidates. For each candidate project p of a class c, the *remaining-graph* of p is defined as follows: i) for each $a \in \mathcal{A}_c^A$ which is already inserted, we remove from $\mathcal{P}_c^A$ the vertex a and all the arcs related to a; ii) the weight of an arc (a, b) in the remaining-graph is the duration of activity a. Each candidate project p is assigned a probability based on the ratio ${CPL'_p}/{(due_p - t)}$ if p is early and ${1}/{CPL'_p}$ otherwise, where CPL'_p is the critical path length of the remaining-graph of p. One project is then selected according to the resulting probability distribution. The set of eligible activities is reduced to $\mathcal{E}^p_{SB}$ by considering only activities from the selected project p.
3. An activity is randomly selected in $\mathcal{E}^p_{SB}$ using a uniform probability distribution.

As the heuristic choice is stochastic, the greedy algorithm is run several times until a given number of solutions have been constructed without improvement. At each step, the tardiness of the projects in the best-constructed solution is analysed to modify the probability of projects for the next cycle. To achieve that, we increase (resp. decrease) the due date of the most advanced (resp. the tardiest) project: this corresponds to a random portion of the smallest amount between the earliness of the most advanced project and the due date of the tardiest project (in our datasets, we do not have the case in which all projects are late). Note that these modified due dates are only used to calculate the project selection probability, not the tardiness objective. This iterative cycle continues until a given timeout condition is reached, and the best-found solution is returned.

6 Experimentation

In this section, we present the results of some experiments conducted on real-world and synthetic benchmarks. Before presenting the experimental results, we describe the benchmarks and the solving approaches tested, including the main ones and their combinations or derivations.

6.1 Benchmarks

Learning Curves. In this research, we use the log-linear learning curve with a steady learning state [23, 25, 11][2]. Formally, for each class $c \in \mathcal{C}$, each atomic activity $a \in \mathcal{A}_c^A$ and each integer n, we consider the following elements: (i) the duration for the first execution, denoted dur_a^0; (ii) the steady state of learning, denoted dur_a^∞, that represents the ultimate duration of a; (iii) the learning effect $l_a \in]0, 1]$, that impacts the duration curve slope. Then, for all $n \geq 0$, $dur_a(n) = dur_a^\infty + \lceil (dur_a^0 - dur_a^\infty) \cdot (n+1)^{\log_2 l_a} \rceil$. In our experiments, we consider that $dur_a^\infty = \frac{1}{2} dur_a^0$.

Table 1 Features of instances per dataset: number of classes, atomic activities, compound activities, projects, resources, capacity of resources, and learning rate.

Instances		$\lvert\mathcal{C}\rvert$	$\lvert\mathcal{A}_c\rvert$	$\lvert\mathcal{A}_c^C\rvert$	$\lvert\mathcal{I}_c\rvert$	$\lvert\mathcal{R}\rvert$	$capa_r$	l_a
PSP-based	small	$[\![2, 4]\!]$	$\{30, 60\}$	0	$[\![5, 10]\!]$	4	$\sum_{i_{LIB}} capa_r^{i_{LIB}}$	$[0.45, 0.95]$
	large	$[\![5, 7]\!]$	$\{60, 90, 120\}$					
Satellite	original	3	≤ 30	≤ 3	≤ 5	40	≤ 16	$[0.05, 0.95]$
	extended	6			≤ 14			0.85

Datasets. We tested our approaches on two datasets presented in Table 1. First, we created the PSP-based dataset (50 instances), where each class is an RCPSP instance from PSPLIB [15]. The due dates are computed as follows: a project has a due date equal to the maximum due date d in the original PSPLIB instance, then for each other project's due date, we iteratively add $k \times d$, where k is a random value in $[0.3, 0.8]$. The time horizon is equal to 40 (resp. 70) times the largest due dates among the involved PSPLIB instances in small (resp. large) instances. The capacity of each resource is the sum of the capacities of that resource from the involved PSPLIB instances. This dataset does not include any compound activity. This dataset is publicly available ([18]).

The satellite dataset contains 24 instances updated from a satellite manufacturer. The original set of instances, described in Table 1, covers a time horizon of more than one year with activities lasting from a few hours to a few days. We only vary the learning rate in these instances. We have created larger instances by increasing the number of classes, projects,

[2] This curve was not provided by our industrial partner but is classically used in the aerospace domain.

and the horizon from the original data while maintaining the resource capacities. The due dates for the additional projects have been randomly selected with good diversity: the due dates are distributed so that there are at most two satellites in every thousand time unit.

6.2 Solving Approaches

We have compared various approaches, based on the main techniques presented in Section 5.

CP-based Approaches. Two CP (variants of) models of HM-RCPSP/L-C are defined as follows: i) LC, Learning Curve, corresponding to Constraints (5) to (10), and ii) LCSB, Learning Curve and Symmetry Breaking, including Constraints (11) as well.

SWO Approach. In our implementation, the greedy algorithm is run until the solution has not been improved 150 times in a row.

Hybrid Approach. SWO is able to produce solutions in a very short time, whereas CP-based approaches can take longer to find a first "good" solution. Therefore, we have tested an hybrid approach, denoted HYB, which consists of computing a solution with SWO in a short time (5 seconds in the experiments) and then using that solution as a starting point for LCSB (that tries to improve it for the rest of the time).

Constant then Learning Curve Approach. The Constant then Learning Curve approach, denoted CLC, aims to adress the complexity of dealing with varying durations (due to learning curves). Such an approach consists of the following steps:

1. we consider the Constant Duration variant of LCSB, denoted CD, in which Constraints (10) are replaced by constraints imposing a constant duration for all activities (actually solving HM-RCPSP/C);
2. we extract the resource-accessing order of activities from the CD solution, i.e. the execution order of all activities;
3. using a Parallel Generation Scheme, we chronologically compute the start date of each activity using the fixed execution order of step 2., considering that the durations of activities follow the specified learning curves.

We have tested three variations of this approach, denoted CLC^0, CLC^{mid} and CLC^{∞}, depending on the value chosen as constant duration in CD: this value is respectively dur_a^0, $1/2(dur_a^0 + dur_a^{\infty})$ and dur_a^{∞}. For each dataset, only the results of the variation with the best performance will be presented in the main paper result tables (complete results for the PSPLib-based dataset are available online [18]).

6.3 Experimental Results

The tests have been launched using IBM CP Optimizer 22.1.1 through the DOcplex API for the CP-based approaches and Julia v1.10.2 for the SWO-based approaches, on Intel® Xeon® CPU E5-2660v3 2.60-3.30 GHz with 62 GB of RAM. All approaches have been launched twice, to compare them over a short timeout (15 seconds) and a long timeout (2 hours).

Satellite Dataset. For the original satellite instances, all approaches achieve zero tardiness within 15 seconds, except CLC^{mid} and CLC^{∞}. The final solutions obtained by CP-based approaches after 2 hours are the same as those obtained after 15 seconds. Interestingly, LCSB and HYB have successfully proved the optimality of 2 instances after 2 hours (whereas

SWO has the worst makespan values). For the five extended satellite instances, the tardiness values are much higher as showed in Table 2. In the 2-hour test, LCSB clearly outperforms the others in 4 instances out of 5. LC falls behind and struggles to order projects in a way that minimizes tardiness. LCSB also provides the best makespan values in all instances but one (not visible in Table 2). Note that the optimality was not proved by any approach. In the 15-second test, HYB achieves the best tardiness in more than half of the instances, but CLC^{mid} surprisingly performs best overall.

Table 2 Tardiness values obtained by all solving approaches on the extended satellite dataset.

$\sum_{c,p} \lvert\mathcal{A}_c\rvert$	2 hours					15 seconds				
	LC	LCSB	SWO	HYB	CLC^{mid}	LC	LCSB	SWO	HYB	CLC^{mid}
637	1,341	1,318	4,566	**1,243**	1,668	5,516	3,286	6,979	**2,021**	2,033
913	1,267	**1,210**	5,963	**1,210**	2,443	9,906	9,981	6,734	**2,544**	2,582
1,217	2,004	**1,786**	13,174	2,626	4,372	48,099	22,121	21,854	**12,510**	13,893
1,526	10,993	**2,135**	23,530	2,149	5,132	126,012	41,409	33,228	41,204	**17,430**
1,832	13,372	**3,040**	39,440	4,403	8,268	210,673	30,827	54,347	54,266	**26,979**

PSP-based Dataset. In Table 3, we present for each criterion and each approach the number of times the best value is computed (obtained) compared to others, and also the average difference from the best value found by all approaches, denoted ADBV, which can be calculated by: $\text{ADBV}(obj, \lambda) = \sum_{i \in \mathcal{I}}(obj_i^{\lambda} - obj_i^{best}) / \sum_{i \in \mathcal{I}}(obj_i^{\lambda} \neq obj_i^{best})$, where $\mathcal{I}$ is the set of instances, obj_i^{λ} is the final value of the objective obj (either tardiness or makespan) found by approach λ in instance i, and obj_i^{best} is the best-found value of the objective obj for instance i by all approaches.

Table 3 Number of times the best value (#Best) is obtained and ADBV values for all solving approaches on the PSP-based dataset.

Timeout	Dataset	Objective	#Best					ADBV				
			LC	LCSB	SWO	HYB	CLC^0	LC	LCSB	SWO	HYB	CLC^0
2h	small	Tardiness	**21**	**21**	0	**21**	15	**13**	27	144	28	113
		Makespan	18	**19**	0	14	0	3	**1**	14	2	31
	large	Tardiness	**17**	16	0	16	7	**66**	109	629	72	451
		Makespan	14	15	4	**16**	0	7	**6**	10	7	62
15s	small	Tardiness	**22**	16	0	19	15	48	40	126	**38**	115
		Makespan	**16**	15	2	10	0	4	**3**	12	5	33
	large	Tardiness	**15**	4	6	3	5	5,494	–	386	**307**	2,011
		Makespan	8	2	6	**12**	0	23	–	**7**	10	96

In the 2-hour test, for the PSP-based instances, LC, LCSB, and HYB performed best, successfully meeting all due dates in 31 to 33 out of 50 instances (not visible in Table 3). Nevertheless, LCSB has a slight advantage when considering the second criterion (makespan). CLC^0 performs satisfactorily but it has a poor ADBV value of tardiness. Clearly, SWO has the worst performance since it never reaches the best value. Note that the optimality was not proved by any solving approach regardless the dataset (small or large).

In the 15-second test, the three approaches LC, LCSB, and HYB keep taking the lead for the small PSP-based dataset, with a slight advantage for LC. However, for the large dataset, the pure CP-based approach LC sometimes struggles to find a good solution in time, resulting in a large ADBV value, even though it achieves the best tardiness in 15 out of 24 instances. SWO showcases its strengths when dealing with big-size instances in a low-timeout condition, while LCSB cannot find a solution for half of the instances. The hybrid approach HYB shows the best overall performance in the 15-second test, with the lowest ADBV value across both small and large PSP-based datasets.

7 Conclusion

In this paper, we have i) formally defined the High Multiplicity RCPSP with Compound activities and Learning effect, ii) proposed a CP model for this problem, iii) proved the existence of symmetric projects within each class, and iv) adapted a heuristic-based search (SWO) using the proof of symmetric projects. Additionally, we compared the performance of various solving approaches using an industrial satellite dataset and a PSP-based one.

On satellite assembly instances, the approaches presented in this paper can generate a schedule within a reasonable amount of time. Interestingly enough, we have observed that the performance of the CP-based approaches was boosted by breaking symmetries on these industrial instances. Importantly, built schedules allow satellite engineers to effectively scale resources, particularly human means, and anticipate potential delays. Besides, industrial partners will be able to fine-tune the schedules according to their environment learning effect features, as our approach is completely generic on that point.

For future research, it would be beneficial to explore cases where learning effect is shared among similar activities across different classes, which will render projects from different classes interdependent. Additionally, considering the uncertainty in learning efficiency could be explored to enhance the robustness of the schedule.

References

1 J.P. Amor. Scheduling programs with repetitive projects using composite learning curve approximations. *Project Management Journal*, 33(3):16–29, 2002.

2 C. Artigues, S. Demassey, and E. Neron. *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. ISTE/Wiley, 2008. URL: https://hal.science/hal-00482946.

3 A. Bachman and A. Janiak. Scheduling jobs with position-dependent processing times. *Journal of the Operational Research Society*, 55(3):257–264, 2004.

4 D. Biskup. Single-machine scheduling with learning considerations. *European Journal of Operational Research*, 115(1):173–178, 1999.

5 D. Biskup. A state-of-the-art review on scheduling with learning effects. *European Journal of Operational Research*, 188(2):315–329, 2008.

6 A. Bonfietti, M. Lombardi, L. Benini, and M. Milano. A constraint based approach to cyclic RCPSP. In *Proceedings of CP'11*, pages 130–144. Springer, 2011.

7 C. Daehnick, I. Klinghoffer, B. Maritz, and B. Wiseman. Large LEO satellite constellations: Will it be different this time? *McKinsey & Company*, 4, 2020.

8 S.J. Edwards, D. Baatar, K. Smith-Miles, and A.T. Ernst. Symmetry breaking of identical projects in the high-multiplicity rcpsp/max. *Journal of the Operational Research Society*, 72(8):1822–1843, 2021.

9 M. Eugeni, T. Quercia, M. Bernabei, A. Boschetto, F. Costantino, L. Lampani, A. Marchetti Spaccamela, A. Lombardo, M. Mecella, L. Querzoni, R. Usinger, M. Aliprandi, A. Stancu, M.M. Ivagnes, G. Morabito, A. Simoni, A. Brandão, and P. Gaudenzi. An industry 4.0 approach to large scale production of satellite constellations. the case study of composite sandwich panel manufacturing. *Acta Astronautica*, 192:276–290, March 2022. doi:10.1016/j.actaastro.2021.12.039.

10 J.D. García-Nieves, J.L. Ponz-Tienda, A. Ospina-Alvarado, and M. Bonilla-Palacios. Multipurpose linear programming optimization model for repetitive activities scheduling in construction projects. *Automation in Construction*, 105:102799, 2019. doi:10.1016/j.autcon.2019.03.020.

11 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.

12 S. Hartmann and D. Briskorn. An updated survey of variants and extensions of the resource-constrained project scheduling problem. *Eur. J. Oper. Res.*, 297(1):1–14, 2022. doi:10.1016/J.EJOR.2021.05.004.

13 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 *Proceedings of CPAIOR'21*, pages 26–44. Springer, 2021.

14 D.E. Joslin and D.P. Clements. Squeaky wheel optimization. *Journal of Artificial Intelligence Research*, 10:353–373, 1999.

15 R. Kolisch and A. Sprecher. Psplib - a project scheduling problem library: Or software - orsep operations research software exchange program. *European Journal of Operational Research*, 96(1):205–216, 1997. doi:10.1016/S0377-2217(96)00170-1.

16 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.

17 Y.C. Law and J. Lee. Global constraints for integer and set value precedence. In *Proceedings of CP'04*, pages 362–376, 2004.

18 D.A. Le and S. Roussel. Dataset for the High Multiplicity RCPSP with Learning Effect and Compound Activities, 2024. Dataset, version 3.2. (visited on 2024-08-19). doi:10.57745/ASGLBH.

19 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.

20 V. Van Peteghem and M. Vanhoucke. Influence of learning in resource-constrained project scheduling. *Computers & Industrial Engineering*, 87:569–579, 2015.

21 S. Qin, S. Liu, H. Kuang, et al. Piecewise linear model for multiskilled workforce scheduling problems considering learning effect and project quality. *Mathematical problems in Engineering*, 2016, 2016.

22 M. Gómez Sánchez, E. Lalla-Ruiz, A. Fernández Gil, C. Castro, and S. Voß. Resource-constrained multi-project scheduling problem: A survey. *European Journal of Operational Research*, 309(3):958–976, 2023. doi:10.1016/j.ejor.2022.09.033.

23 T.P. Wright. Factors affecting the cost of airplanes. *Journal of the aeronautical sciences*, 3(4):122–128, 1936.

24 M.C. Wu and S.H. Sun. A project scheduling and staff assignment model considering learning effect. *The International Journal of Advanced Manufacturing Technology*, 28:1190–1195, 2006.

25 L.E. Yelle. The learning curve: Historical review and comprehensive survey. *Decision sciences*, 10(2):302–328, 1979.

26 Y. Yin, D. Xu, K. Sun, and H. Li. Some scheduling problems with general position-dependent and time-dependent learning effects. *Information Sciences*, 179(14):2416–2425, 2009.

Appendix

In this appendix, we provide the complete proofs of lemmas in Section 4 for breaking the symmetry. Note that we believe that sketches of proof provided in the paper are sufficient for understanding and reproducing the complete ones.

Note that the complete result tables of the tests in Section 6 are available at the datasets URL.

A Lemma 4

Complete proof. Let σ be a solution that is not start-end-consistent. There exists a class $c \in \mathcal{C}$, an atomic activity $a \in \mathcal{A}_c^A$, two projects p and q in $\mathcal{I}_c$ such that $start^{\sigma}(a^p) < start^{\sigma}(a^q)$ and $end^{\sigma}(a^p) \geq end^{\sigma}(a^q)$. We create a new solution $\upsilon_{a,p,q}$ such that for all activities b in $\mathcal{A}_c^A$, for all projects $u \in \mathcal{I}_c$ such that $b^u \neq a^p$, $start^{\sigma}(b^u) = start^{\upsilon_{a,p,q}}(b^u)$. Then, for activity a and project p, we delay the start of activity a^p in order to make it start at the same time as a^q. Formally, we have $start^{\upsilon_{a,p,q}}(a^p) = start^{\upsilon_{a,p,q}}(a^q) = start^{\sigma}(a^q)$. We also have $end^{\upsilon_{a,p,q}}(a^p) = end^{\upsilon_{a,p,q}}(a^q) = end^{\sigma}(a^q)$.

As a^p starts later and finishes earlier in $\upsilon_{a,p,q}$, all precedence that are satisfied in σ are also satisfied in $\upsilon_{a,p,q}$. The same holds for the resource consumption. If a^p is the activity which concludes project p in σ (*i.e.* $end^{\sigma}(\omega_c^p) = end^{\sigma}(a^p)$), then we have $end^{\upsilon_{a,p,q}}(a^p) \leq end^{\upsilon_{a,p,q}}(\omega_c^p) \leq end^{\sigma}(\omega_c^p)$. The end date of project p in $\upsilon_{a,p,q}$ either becomes earlier or remains the same as in σ. As the timing for all other projects remains the same as in σ, the values of both criteria for $\upsilon_{a,p,q}$ are either better than or equivalent to σ.

We perform the solution modification iteratively for each tuple (a, p, q) that is not start-end-consistent. The resulting solution υ is start-end-consistent and at least as good as σ for both criteria. ◀

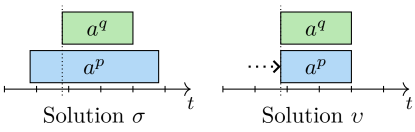


Figure 8 Illustration of Lemma 4's proof: σ is not start-end-consistent but υ is.

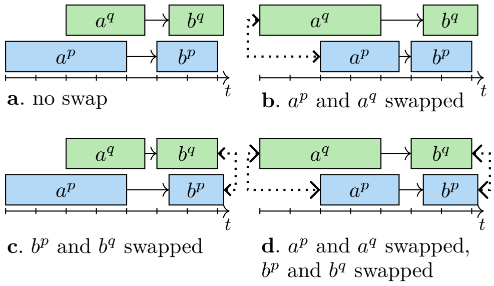

a^q
b^q
a^p
b^p
t
a. no swap
a^q
b^q
a^p
b^p
t
b. a^p and a^q swapped
a^q
b^q
a^p
b^p
t
c. b^p and b^q swapped
a^q
b^q
a^p
b^p
t
d. a^p and a^q swapped, b^p and b^q swapped

Figure 9 Four precedence cases considered in the proof of Lemma 6.

B Lemma 6

Complete proof. We assume that π is not precedence-feasible. Since all precedence relations in $\mathcal{P}_c$ can be transposed into $\mathcal{P}_c^A$ (see Assumption 5), which is concerned only with atomic activities, this implies the existence of two atomic activities $a, b \in \mathcal{A}_c^A$ such that $(a, b) \in \mathcal{P}_c^A$ and such that at least one of the following equations hold:

$$end^{\pi}(a^p) > start^{\pi}(b^p) \tag{12}$$

$$end^{\pi}(a^q) > start^{\pi}(b^q) \tag{13}$$

Due to the definition of the start dates for the two virtual activities α_c and ω_c (see Schedule), these activities never contribute to a precedence violation and they are not considered in the following.

Case 1. $start^\sigma(a^p) \leq start^\sigma(a^q)$ and $start^\sigma(b^p) \leq start^\sigma(b^q)$ (Figure 9a). This means that there is no swap of activities, and the start dates of a^p, a^q, b^p, b^q in π remain the same as in σ. In this case, end dates are not modified and we have:

- $end^\pi(a^p) = end^\sigma(a^p)$, $start^\pi(b^p) = start^\sigma(b^p)$
- $end^\pi(a^q) = end^\sigma(a^q)$, $start^\pi(b^q) = start^\sigma(b^q)$

If any of the conditions 12 and 13 is met, then it means that σ is not be precedence-feasible. This leads to a contradiction.

Case 2. $start^\sigma(a^p) > start^\sigma(a^q)$ and $start^\sigma(b^p) \leq start^\sigma(b^q)$ (Figure 9b). This means that in the schedule π, the start dates of a^p and a^q are swapped, but b^p and b^q remain the same as in σ. In this case, we have:

- $end^\pi(a^p) = end^\sigma(a^q)$; $start^\pi(b^p) = start^\sigma(b^p)$
- $end^\pi(a^q) = end^\sigma(a^p)$; $start^\pi(b^q) = start^\sigma(b^q)$

If Cond. 12 is true, then we have $end^\sigma(a^q) > start^\sigma(b^p)$. Since $start^\sigma(a^p) > start^\sigma(a^q)$ from hypothesis, then $end^\sigma(a^p) > end^\sigma(a^q)$ by the Definition 3. Therefore, we have $end^\sigma(a^p) > end^\sigma(a^q) > start^\sigma(b^p)$. As a result, σ is not precedence-feasible, leading to a contradiction.

If Cond. 13 is true, then we have $end^\sigma(a^p) > start^\sigma(b^q)$. Since $start^\sigma(b^q) \geq start^\sigma(b^p)$ from hypothesis, then we have $end^\sigma(a^p) > start^\sigma(b^p)$. As a result, σ is not precedence-feasible, leading to a contradiction.

Case 3. $start^\sigma(a^p) \leq start^\sigma(a^q)$ and $start^\sigma(b^p) > start^\sigma(b^q)$ (Figure 9c). This means that in the schedule π, the positions of a^p and a^q remain the same as in σ, but b^p and b^q are swapped. In this case, we have:

- $end^\pi(a^p) = end^\sigma(a^p)$; $start^\pi(b^p) = start^\sigma(b^q)$
- $end^\pi(a^q) = end^\sigma(a^q)$; $start^\pi(b^q) = start^\sigma(b^p)$

If Cond. 12 is true, then we have $end^\sigma(a^p) > start^\sigma(b^q)$. Since $start^\sigma(a^q) \geq start^\sigma(a^p)$ from hypothesis, then $end^\sigma(a^q) \geq end^\sigma(a^p)$ by the Definition 3. Therefore, we have $end^\sigma(a^q) \geq end^\sigma(a^p) > start^\sigma(b^q)$. As a result, σ is not precedence-feasible, leading to a contradiction.

If Cond. 13 is true, then we have $end^\sigma(a^q) > start^\sigma(b^p)$. Since we have $start^\sigma(b^p) > start^\sigma(b^q)$, then $end^\sigma(a^q) > start^\sigma(b^p) > start^\sigma(b^q)$. As a result, σ is not precedence-feasible, leading to a contradiction.

Case 4. $start^\sigma(a^p) > start^\sigma(a^q)$ and $start^\sigma(b^p) > start^\sigma(b^q)$ (Figure 9d). This means that in the schedule π, the positions between a^p and a^q, and between b^p and b^q are swapped. In this case, we have:

- $end^\pi(a^p) = end^\sigma(a^q)$; $start^\pi(b^p) = start^\sigma(b^q)$
- $end^\pi(a^q) = end^\sigma(a^p)$; $start^\pi(b^q) = start^\sigma(b^p)$

If any of the conditions 12 and 13 is met, σ will not be precedence-feasible, leading to a contradiction.

Every possible cases are leading to a contradiction, so π is precedence-feasible and lemma 6 is proved. ◀

C Lemma 7

Complete proof. We first define several elements for the proof.

Let $T^{\sigma}_{p,a}$ denote the time interval during which the activity $a \in \mathcal{A}_c$ of project p of a class c is active in a schedule σ, *i.e.* $T^{\sigma}_{p,a} = \{t \in [\![0, H]\!] | \mathit{start}^{\sigma}(a^p) \leq t < \mathit{end}^{\sigma}(a^p)\}$.

We then define a function representing the resource consumption of an activity within a project at a given time step in a schedule.

For any resource $r \in \mathcal{R}$, any time step $t \in [\![0, H]\!]$, any class $c \in \mathcal{C}$ and any activity $a \in \mathcal{A}_c$, we define a function γ that takes in parameter a schedule σ and a project $p \in \mathcal{I}_c$ and that has the following value:

$$\gamma_{r,a,t}(p, \sigma) = \begin{cases} \mathit{cons}_{r,a} & \text{if } t \in T^{\sigma}_{p,a} \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

The consumption of a resource r at each time step t in a schedule σ is given by the formula $\sum_{c \in \mathcal{C}, p \in \mathcal{I}_c, a \in \mathcal{A}_c} \gamma_{r,a,t}(p, \sigma)$.

Then, we define a function $\Delta_{r,a,t}(p, \sigma, \pi)$ representing the difference in resource consumption of resource r by activity a in project p between two schedules σ and π, at time step t. Formally,

$$\Delta_{r,a,t}(p, \sigma, \pi) = \gamma_{r,a,t}(p, \pi) - \gamma_{r,a,t}(p, \sigma)$$

We now consider a solution σ, a class $c \in \mathcal{C}$ and two projects $p, q \in \mathcal{I}_c$. We suppose the (p, q)-permutation π of σ is not resource-feasible. This means there exists a time moment $t \in [\![0, H]\!]$ and a resource $r \in \mathcal{R}$ such that $\sum_{d \in \mathcal{C}, u \in \mathcal{I}_d, a \in \mathcal{A}_d} \gamma_{r,a,t}(u, \pi) > \mathit{capa}_r$. Since σ is resource-feasible, we have $\sum_{d \in \mathcal{C}, u \in \mathcal{I}_d, a \in \mathcal{A}_d} \gamma_{r,a,t}(u, \sigma) \leq \mathit{capa}_r$. This implies:

$$\sum_{d \in \mathcal{C}, u \in \mathcal{I}_d, a \in \mathcal{A}_d} \big(\gamma_{r,a,t}(u, \pi) - \gamma_{r,a,t}(u, \sigma)\big) > 0$$

and then:

$$\sum_{d \in \mathcal{C}, u \in \mathcal{I}_d, a \in \mathcal{A}_d} \Delta_{r,a,t}(u, \sigma, \pi) > 0 \tag{15}$$

Following the definition of π (Definition 5), any class $d \in \mathcal{C}$, any project $u \in \mathcal{I}_d$ such that $u \neq p$ and $u \neq q$, and any activity $a \in \mathcal{A}_d$, $\mathit{start}^{\pi}(a^u) = \mathit{start}^{\sigma}(a^u)$ and $\mathit{end}^{\pi}(a^u) = \mathit{end}^{\sigma}(a^u)$. This means that for any time moment $t \in [\![0, H]\!]$, $\gamma_{r,a,t}(u, \sigma) = \gamma_{r,a,t}(u, \pi)$ and then $\Delta_{r,a,t}(u, \sigma, \pi) = 0$, i.e. the consumption of resource r by activity a^u does not change. Equation 15 can be simplified by only considering activities of projects p and q as:

$$\sum_{a \in \mathcal{A}_c} \big(\Delta_{r,a,t}(p, \sigma, \pi) + \Delta_{r,a,t}(q, \sigma, \pi)\big) > 0 \tag{16}$$

Case 1. Consider an arbitrary atomic activity $a \in \mathcal{A}^A_c$. The two virtual activities α_c and ω_c are not considered in the following as they do not consume resources (see Assumption 3). The two activities a^p and a^q are positioned in π according to the condition in Definition 5, that is:

$$\begin{aligned} \mathit{start}^{\sigma}(a^p) \leq \mathit{start}^{\sigma}(a^q) &\implies \mathit{start}^{\pi}(a^p) = \mathit{start}^{\sigma}(a^p) \textit{ and } \mathit{start}^{\pi}(a^q) = \mathit{start}^{\sigma}(a^q) \\ \mathit{start}^{\sigma}(a^p) > \mathit{start}^{\sigma}(a^q) &\implies \mathit{start}^{\pi}(a^p) = \mathit{start}^{\sigma}(a^q) \textit{ and } \mathit{start}^{\pi}(a^q) = \mathit{start}^{\sigma}(a^p) \end{aligned}$$

In this case, we have $T^{\pi}_{p,a} \cup T^{\pi}_{q,a} = T^{\sigma}_{p,a} \cup T^{\sigma}_{q,a}$ and $T^{\pi}_{p,a} \cap T^{\pi}_{q,a} = T^{\sigma}_{p,a} \cap T^{\sigma}_{q,a}$ because:

- If $\mathit{start}^{\sigma}(a^p) \leq \mathit{start}^{\sigma}(a^q)$ then $T^{\pi}_{p,a} = T^{\sigma}_{p,a}$ and $T^{\pi}_{q,a} = T^{\sigma}_{q,a}$
- If $\mathit{start}^{\sigma}(a^p) > \mathit{start}^{\sigma}(a^q)$ then $T^{\pi}_{p,a} = T^{\sigma}_{q,a}$ and $T^{\pi}_{q,a} = T^{\sigma}_{p,a}$

Case 1.1. If there is no swap between a^p and a^q, then $T^{\pi}_{p,a} = T^{\sigma}_{p,a}$ and $T^{\pi}_{q,a} = T^{\sigma}_{q,a}$. Consumption of resource r is not modified at time t and we have $\Delta_{r,a,t}(p,\sigma,\pi) = \Delta_{r,a,t}(q,\sigma,\pi) = 0$.

Case 1.2. If there is a swap between a^p and a^q, then $T^{\pi}_{p,a} = T^{\sigma}_{q,a}$ and $T^{\pi}_{q,a} = T^{\sigma}_{p,a}$.

1. If $t \notin T^{\sigma}_{p,a} \cup T^{\sigma}_{q,a}$, then $t \notin T^{\pi}_{p,a} \cup T^{\pi}_{q,a}$. Both a^p and a^q are not being executed at time t in π, so

$$\gamma_{r,a,t}(p,\sigma) = \gamma_{r,a,t}(q,\sigma) = \gamma_{r,a,t}(p,\pi) = \gamma_{r,a,t}(q,\pi) = 0$$

   This implies $\Delta_{r,a,t}(p,\sigma,\pi) = \Delta_{r,a,t}(q,\sigma,\pi) = 0$.

2. If $t \in (T^{\sigma}_{p,a} \cup T^{\sigma}_{q,a}) \setminus (T^{\sigma}_{p,a} \cap T^{\sigma}_{q,a})$, then $t \in (T^{\pi}_{p,a} \cup T^{\pi}_{q,a}) \setminus (T^{\pi}_{p,a} \cap T^{\pi}_{q,a})$. In both schedules σ and π, either a^p or a^q is being executed at time t.

   a. If $t \in T^{\sigma}_{p,a}$, $t \notin T^{\sigma}_{q,a}$, then we have $t \in T^{\pi}_{q,a}$ and $t \notin T^{\pi}_{p,a}$. This implies:

$$\gamma_{r,a,t}(p,\sigma) = \gamma_{r,a,t}(q,\pi) = cons_{r,a} \ ; \ \gamma_{r,a,t}(q,\sigma) = \gamma_{r,a,t}(p,\pi) = 0$$

   and then:

$$\Delta_{r,a,t}(p,\sigma,\pi) + \Delta_{r,a,t}(q,\sigma,\pi) = -cons_{r,a} + cons_{r,a} = 0$$

   b. Similarly, if $t \notin T^{\sigma}_{p,a}$, $t \in T^{\sigma}_{q,a}$ then:

$$\gamma_{r,a,t}(p,\sigma) = \gamma_{r,a,t}(q,\pi) = 0 \ ; \ \gamma_{r,a,t}(q,\sigma) = \gamma_{r,a,t}(p,\pi) = cons_{r,a}$$

   This implies:

$$\Delta_{r,a,t}(p,\sigma,\pi) + \Delta_{r,a,t}(q,\sigma,\pi) = cons_{r,a} - cons_{r,a} = 0$$

3. If $t \in T^{\sigma}_{p,a} \cap T^{\sigma}_{q,a}$, then $t \in T^{\pi}_{p,a} \cap T^{\pi}_{q,a}$. Both a^p and a^q are being executed at time t in π, so

$$\gamma_{r,a,t}(p,\sigma) = \gamma_{r,a,t}(q,\sigma) = \gamma_{r,a,t}(p,\pi) = \gamma_{r,a,t}(q,\pi) = cons_{r,a}$$

   This implies $\Delta_{r,a,t}(p,\sigma,\pi) = \Delta_{r,a,t}(q,\sigma,\pi) = 0$.

From the above, we can conclude that the swap between two atomic activities never violates the resource constraint as $\Delta_{r,a,t}(p,\sigma,\pi) + \Delta_{r,a,t}(q,\sigma,\pi) = 0$. Therefore, Equation 16 can be rewritten as:

$$\sum_{a \in \mathcal{A}^C_c} \Delta_{r,a,t}(p,\sigma,\pi) + \Delta_{r,a,t}(q,\sigma,\pi) > 0 \tag{17}$$

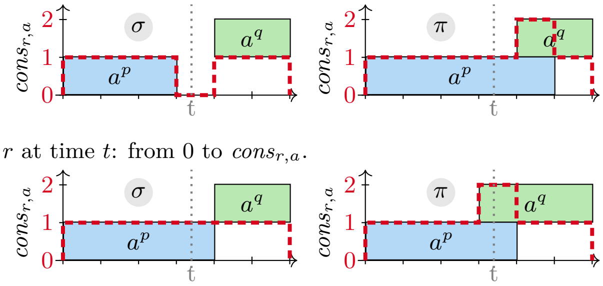
Case 2. Consider an arbitrary compound activity $a \in \mathcal{A}^C_c$. There are two ways for making $\Delta_{r,a,t}(p,\sigma,\pi) + \Delta_{r,a,t}(q,\sigma,\pi)$ strictly positive.

1. If activities a^p and a^q do not use r at time t in σ, but at least one of them uses it in π, then we have $\gamma_{r,a,t}(p,\sigma) = \gamma_{r,a,t}(q,\sigma) = 0$ and $\gamma_{r,a,t}(p,\pi) + \gamma_{r,a,t}(q,\pi) \geq cons_{r,a}$. This is expressed through the following condition.

$$(t \notin T^{\sigma}_{p,a} \cup T^{\sigma}_{q,a}) \wedge (t \in T^{\pi}_{p,a} \cup T^{\pi}_{q,a}) \tag{18}$$

2. if exactly one activity (a^p or a^q) uses r at time t in σ and they both use it in π, then $\gamma_{r,a,t}(p,\sigma) + \gamma_{r,a,t}(q,\sigma) = cons_{r,a}$ and $\gamma_{r,a,t}(p,\pi) + \gamma_{r,a,t}(q,\pi) = 2cons_{r,a}$. This is expressed by

$$(t \in (T^{\sigma}_{p,a} \cup T^{\sigma}_{q,a}) \setminus (T^{\sigma}_{p,a} \cap T^{\sigma}_{q,a})) \wedge (t \in T^{\pi}_{p,a} \cap T^{\pi}_{q,a}) \tag{19}$$



(a) Usage of resource r at time t: from 0 to $cons_{r,a}$.

(b) Usage of resource r at time t: from $cons_{r,a}$ to $2cons_{r,a}$.

Figure 10 Two impossible cases considered in proof of Lemma 7 when $a \in \mathcal{A}_c^C$.

Case 2.1. Let the condition 18 be true.

1. If $t < start^\sigma(a^p)$ and $t < start^\sigma(a^q)$ then for all $b \in \mathcal{A}_c$ with $(a, b) \in \mathcal{H}_c$, we have $t < start^\sigma(b^p)$ and $t < start^\sigma(b^q)$. We then have $t < start^\pi(b^p)$ and $t < start^\pi(b^q)$ and this implies $t \notin T^\pi_{p,a} \cup T^\pi_{q,a}$, leading to a contradiction.
2. If $t \geq end^\sigma(a^p)$ and $t \geq end^\sigma(a^q)$ then for all $b \in \mathcal{A}_c$ such that $(a, b) \in \mathcal{H}_c$, we have $t \geq end^\sigma(b^p)$ and $t \geq end^\sigma(b^q)$. We then have $t \geq end^\pi(b^p)$ and $t \geq end^\pi(b^q)$ and this implies $t \notin T^\pi_{p,a} \cup T^\pi_{q,a}$, leading to a contradiction.
3. If $end^\sigma(a^p) \leq t < start^\sigma(a^q)$ then for all $b \in \mathcal{A}_c$ such that $(a, b) \in \mathcal{H}_c$, we have $start^\sigma(b^p) \leq end^\sigma(b^p) \leq t < start^\sigma(b^q)$. This implies that zero swap operations were performed. Thus, $t \notin T^\pi_{p,a} \cup T^\pi_{q,a}$, leading to a contradiction.
4. If $end^\sigma(a^q) \leq t < start^\sigma(a^p)$ then for all $b \in \mathcal{A}_c$ such that $(a, b) \in \mathcal{H}_c$, we have $start^\sigma(b^q) \leq end^\sigma(b^q) \leq t < start^\sigma(b^p)$. We then have $start^\pi(b^p) \leq end^\pi(b^p) \leq t < start^\pi(b^q)$ and this implies $t \notin T^\pi_{p,a} \cup T^\pi_{q,a}$, leading to a contradiction.

All possible situations lead to a contradiction, so condition 18 is never satisfied.

Case 2.2. Let the condition 19 be true.

1. If $T^\sigma_{p,a} \cap T^\sigma_{q,a} = \emptyset$, then for all $t' \in T^\sigma_{p,a}$, either $t' < start^\sigma(a^q)$ or $t' \geq end^\sigma(a^q)$. In the first case, none activities from p and q are swapped, meaning that $T^\pi_{p,a} = T^\sigma_{p,a}$ and $T^\pi_{q,a} = T^\sigma_{q,a}$. In the second case, all activities from p and q are swapped, meaning that $T^\pi_{p,a} = T^\sigma_{q,a}$ and $T^\pi_{q,a} = T^\sigma_{p,a}$. In both cases, $T^\pi_{p,a} \cap T^\pi_{q,a} = T^\sigma_{p,a} \cap T^\sigma_{q,a} = \emptyset$, leading to a contradiction.
2. If $T^\sigma_{p,a} \cap T^\sigma_{q,a} \neq \emptyset$, there are four possible situations (see Figure 11):
   a. We suppose that $start^\sigma(a^p) \leq t < start^\sigma(a^q)$. Because π is a (p, q)-permutation of σ, for all activity $b \in \mathcal{A}_c$ such that $(a, b) \in \mathcal{H}_c$, we have $start^\sigma(b^q) \leq start^\pi(b^q)$. In particular, for each b such that $(a, b) \in \mathcal{H}_c$, we have $t < start^\sigma(b^q) \leq start^\pi(b^q)$. As $start^\pi(a^q)$ is the minimum value of start dates of b^q, we have $t < start^\pi(a^q)$. This implies that $t \notin T^\pi_{q,a}$, which contradicts Condition 19.
   b. We suppose that $start^\sigma(a^q) \leq t < start^\sigma(a^p)$. Let b be a child activity of a. Because $start^\sigma(a^p) \leq start^\sigma(b^p)$, we have $t < start^\sigma(b^p)$.
      - Suppose that $start^\sigma(b^q) < start^\sigma(a^p)$. As $start^\sigma(a^p) \leq start^\sigma(b^p)$, we have $start^\sigma(b^q) < start^\sigma(b^p)$. Therefore, b^q and b^p are swapped in π, which means that $start^\pi(b^q) = start^\sigma(b^p)$. We therefore have $t < start^\pi(b^q)$.
      - Suppose that $start^\sigma(b^q) \geq start^\sigma(a^p)$. Because start dates of activities of q in π are always greater or equal to those in σ, we have $start^\pi(b^q) \geq start^\sigma(b^q) \geq start^\sigma(a^p)$. This means that $t < start^\pi(b^q)$.

      We have $t < start^\pi(b^q)$ for all $b \in \mathcal{A}_c$ such that $(a, b) \in \mathcal{H}_c$, so $t < start^\pi(a^q)$ and then $t \notin T^\pi_{q,a}$, leading to a contradiction.

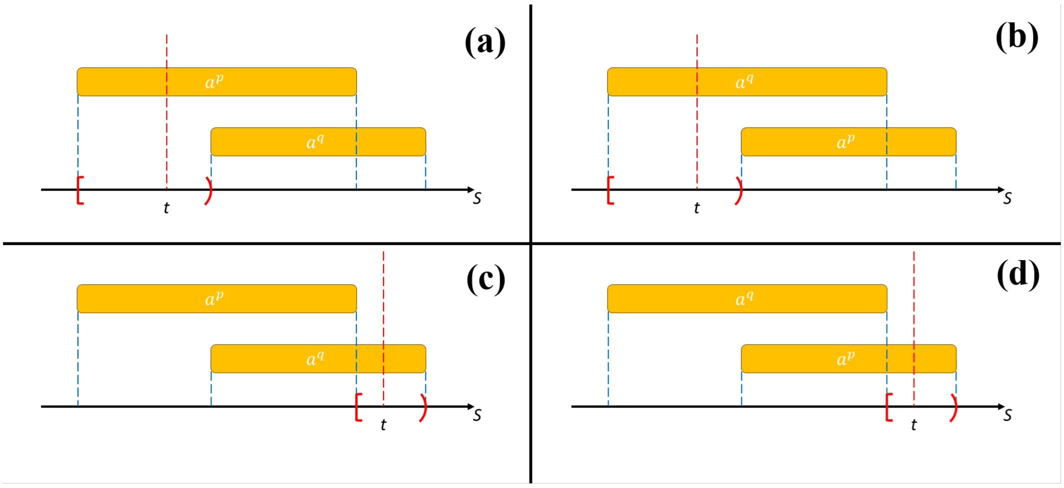


Figure 11 Four possible situation if $T^\sigma_{p,a} \cap T^\sigma_{q,a} \neq \emptyset$ (Case 2.2.2 – Lemma 7).

c. Let suppose that $end^\sigma(a^p) \leq t < end^\sigma(a^q)$. Let b be a child activity of a. Because $end^\sigma(b^p) \leq end^\sigma(a^p)$, we have $end^\sigma(b^p) \leq t$. π is a (p,q)-permutation so $end^\pi(b^p) \leq end^\sigma(b^p)$. So, $end^\pi(b^p) \leq t$.
This means that $end^\pi(a^p) \leq t$, which implies that $t \notin T^\pi_{p,a}$, leading to a contradiction.

d. Let suppose that $end^\sigma(a^q) \leq t < end^\sigma(a^p)$. Let b be a child activity of a. Because $end^\sigma(b^q) \leq end^\sigma(a^q)$, we have $end^\sigma(b^q) \leq t$.

- Let suppose that $end^\sigma(a^q) < end^\sigma(b^p)$. Then, $end^\sigma(b^q) < end^\sigma(b^p)$. With definition 3, this means that $start^\sigma(b^q) < start^\sigma(b^p)$. This condition implies that b^q and b^p are swapped in π. Therefore, $end^\pi(b^p) = end^\sigma(b^q)$. So $end^\pi(b^p) \leq t$.
- Now suppose that $end^\sigma(a^q) \geq end^\sigma(b^p)$. We have $end^\sigma(b^p) \geq end^\pi(b^p)$ (end dates of activities of p in σ are greater or equal to those in π). Therefore, $t \geq end^\pi(b^p)$.

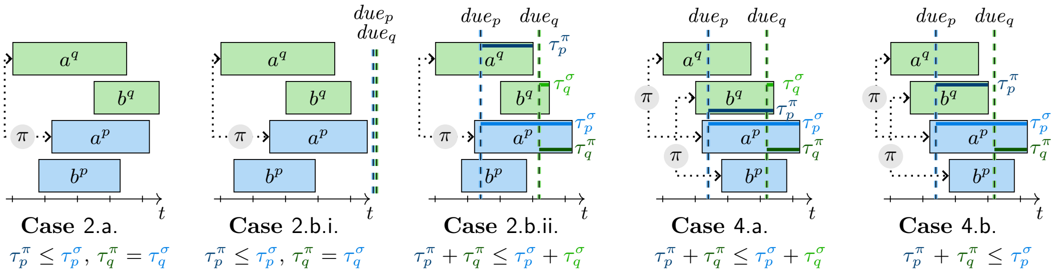
We have $t \geq end^\pi(b^p)$ for all child b of a so $t \geq end^\pi(a^p)$. This means that $t \notin T^\pi_{p,a}$, leading to a contradiction.

All possible situations lead to a contradiction, so condition 19 is never satisfied. Both conditions 18 and 19 are never satisfied, so the sum $\Delta_{r,a,t}(p,\sigma,\pi) + \Delta_{r,a,t}(q,\sigma,\pi)$ is never positive.

In all cases, we have $\sum_{a \in \mathcal{A}^C_c} \Delta_{r,a,t}(p,\sigma,\pi) + \Delta_{r,a,t}(q,\sigma,\pi) \leq 0$ so the (p,q)-permutation π of σ is resource-feasible. ◀

D Lemma 11

Complete Proof. As all compound activities start and conclude with at least one atomic activity among their children, the end date of a project only depends on the end date of its atomic activities. Since σ is start-end-consistent, π is also start-end-consistent because the swap operation does not change the duration of atomic activities. Thus, we have $start^\pi(a^p) \leq start^\pi(a^q)$ and $end^\pi(a^p) \leq end^\pi(a^q), \forall a \in \mathcal{A}^A_c$. Because $p \prec q$, then for each activity $a \in \mathcal{A}^A_c$ that is swapped, we have $end^\pi(a^p) = end^\sigma(a^q) < end^\sigma(a^p)$. This implies that the statement $\tau^\pi_p \leq \tau^\sigma_p$ is always true.
Let $a, b \in \mathcal{A}^A_c$ such that a^p concludes project p and b^q concludes project q in the schedule σ. There are four possible situations:



Case 2.a. $\tau_p^\pi \le \tau_p^\sigma$, $\tau_q^\pi = \tau_q^\sigma$

Case 2.b.i. $\tau_p^\pi \le \tau_p^\sigma$, $\tau_q^\pi = \tau_q^\sigma$

Case 2.b.ii. $\tau_p^\pi + \tau_q^\pi \le \tau_p^\sigma + \tau_q^\sigma$

Case 4.a. $\tau_p^\pi + \tau_q^\pi \le \tau_p^\sigma + \tau_q^\sigma$

Case 4.b. $\tau_p^\pi + \tau_q^\pi \le \tau_p^\sigma$

Figure 12 Illustration of cases considered in Lemma 11's proof.

1. Both a and b are not swapped in π then for each activity $i \in \mathcal{A}_c^A$, we have $end^\sigma(i^p) \le end^\sigma(a^p) \le end^\sigma(a^q) \le end^\sigma(b^q)$ and $end^\sigma(i^q) \le end^\sigma(b^q)$. Whether activity i is swapped or not, we always have $end^\pi(i^q) \le end^\sigma(b^q) = end^\pi(b^q)$, so b^q is still the activity that concludes q in π and $\tau_q^\pi = \tau_q^\sigma$. Since $\tau_p^\pi \le \tau_p^\sigma$ is always true, π is at least as good as σ in this criterion.
2. a is swapped but b is not swapped in π.
   a. If $end^\sigma(a^p) \le end^\sigma(b^q)$ then for each activity $i \in \mathcal{A}_c^A$, we have $end^\sigma(i^p) \le end^\sigma(a^p) \le end^\sigma(b^q)$ and $end^\sigma(i^q) \le end^\sigma(b^q)$. Whether activity i is swapped or not, we always have $end^\pi(i^q) \le end^\sigma(b^q) = end^\pi(b^q)$, so b^q is still the activity that concludes q in π and $\tau_q^\pi = \tau_q^\sigma$. Since $\tau_p^\pi \le \tau_p^\sigma$ is always true, π is at least as good as σ in this criterion.
   b. If $end^\sigma(a^p) > end^\sigma(b^q)$ then for each activity $i \in \mathcal{A}_c^A$, we have $end^\sigma(i^q) \le end^\sigma(b^q) < end^\sigma(a^p)$ and $end^\sigma(i^p) \le end^\sigma(a^p)$. Whether activity i is swapped or not, we always have $end^\pi(i^q) \le end^\sigma(a^p) = end^\pi(a^q)$, so a^q is the activity that concludes q in π.
      i. If $end^\sigma(a^p) \le due_q$ then $end^\pi(a^q) \le due_q$. This implies $\tau_q^\pi = 0$ and since $\tau_p^\pi \le \tau_p^\sigma$ is always true, π is at least as good as σ in this criterion.
      ii. If $end^\sigma(a^p) > due_q \ge due_p$, then $\tau_q^\pi = end^\sigma(a^p) - due_q$ and $\tau_p^\sigma = end^\sigma(a^p) - due_p$. For each activity $i \in \mathcal{A}_c^A$, there exists i^q such that $end^\sigma(i^q) \le end^\sigma(b^q)$. Whether i is swapped or not in π, we always have $end^\pi(i^p) \le end^\sigma(b^q)$. Thus we have $\tau_p^\pi \le \max(0, end^\sigma(b^q) - due_p)$.

         We also have $\tau_q^\sigma = \max(0, end^\sigma(b^q) - due_q)$. We next consider three cases for the relative values for $end^\sigma(b^q)$, due_p and due_q.
         A. If $end^\sigma(b^q) \ge due_q$, then $end^\sigma(b^q) \ge due_p$. So $\tau_p^\pi \le end^\sigma(b^q) - due_p$. We also have $\tau_q^\sigma = end^\sigma(b^q) - due_q$.

            Therefore, $\tau_p^\pi + \tau_q^\pi \le end^\sigma(b^q) - due_p + end^\sigma(a^p) - due_q$. The right member of that inequality is exactly equal to $\tau_p^\sigma + \tau_q^\sigma$ so $\tau_p^\pi + \tau_q^\pi \le \tau_p^\sigma + \tau_q^\sigma$.
         B. If $due_p \le end^\sigma(b^q) \le due_q$, then $\tau_q^\sigma = 0$ and $\tau_p^\pi \le end^\sigma(b^q) - due_p$. Then, we have: $\tau_p^\pi + \tau_q^\pi \le end^\sigma(b^q) - due_p + end^\sigma(a^p) - due_q$. The right part is equal to $end^\sigma(b^q) - due_q + \tau_p^\sigma$. Because $end^\sigma(b^q) \le due_q$, we have $\tau_p^\pi + \tau_q^\pi \le \tau_p^\sigma$. As $\tau_q^\sigma = 0$, we have $\tau_p^\pi + \tau_q^\pi \le \tau_p^\sigma + \tau_q^\sigma$.
         C. If $end^\sigma(b^q) \le due_p$, then $end^\sigma(b^q) \le due_q$. This means that $\tau_p^\pi = \tau_q^\sigma = 0$. Therefore, $\tau_p^\pi + \tau_q^\pi = end^\sigma(a^p) - due_q$. Because $due_p \le due_q$, $\tau_p^\pi + \tau_q^\pi \le end^\sigma(a^p) - due_p$, where the left part is equal to τ_p^σ. As $\tau_q^\sigma = 0$, then $\tau_p^\pi + \tau_q^\pi \le \tau_p^\sigma + \tau_q^\sigma$.

   We have shown that in this case, $\tau_p^\pi + \tau_q^\pi \le \tau_p^\sigma + \tau_q^\sigma$ so π is at least as good as σ for the tardiness criterion.
3. a is not swapped but b is swapped in π. This means $end^\sigma(a^p) \le end^\sigma(a^q)$ and $end^\sigma(b^q) < end^\sigma(b^p)$. As a^p concludes p in σ, we have $end^\sigma(b^q) < end^\sigma(b^p) \le end^\sigma(a^p) \le end^\sigma(a^q)$. This violates the condition that b^q concludes q in σ, so this situation never happen.

4. Both a and b are swapped in π. We have $end^{\sigma}(a^q) < end^{\sigma}(a^p)$ and $end^{\sigma}(b^q) < end^{\sigma}(b^p)$. Because a^p concludes p and b^q concludes q in σ, then:
$end^{\sigma}(a^q) \leq end^{\sigma}(b^q) < end^{\sigma}(b^p) \leq end^{\sigma}(a^p)$
For each activity $i \in \mathcal{A}_c^A$, i^q and i^p are such that $end^{\sigma}(i^q) \leq end^{\sigma}(b^q)$ and $end^{\sigma}(i^p) \leq end^{\sigma}(a^p)$. Whether i is swapped or not in π, we always have $end^{\pi}(i^p) \leq end^{\sigma}(b^q) = end^{\pi}(b^p)$ and $end^{\pi}(i^q) \leq end^{\sigma}(a^p) = end^{\pi}(a^q)$. Thus, we have b^p concludes p and a^q concludes q in π. If:

a. q is late in σ then p is also late in σ because $due_p \leq due_q$ and $end^{\sigma}(b^q) < end^{\sigma}(a^p)$. We have:
$due_p \leq due_q < end^{\sigma}(b^q) = end^{\pi}(b^p) < end^{\sigma}(a^p) = end^{\pi}(a^q)$
So both p and q are late in π. We have:
$\tau_p^{\pi} + \tau_q^{\pi} = end^{\pi}(b^p) - due_p + end^{\pi}(a^q) - due_q$
$= end^{\sigma}(b^q) - due_p + end^{\sigma}(a^p) - due_q$
$= \tau_p^{\sigma} + \tau_q^{\sigma}$
So π and σ are equivalent in this criterion.

b. q is early and p is late in σ then $\tau_p^{\sigma} = end^{\sigma}(a^p) - due_p$ and $\tau_q^{\sigma} = 0$. Since a^p is the last finished activity for both projects in σ and $due_p \leq due_q$, the value $end^{\sigma}(a^p) - due_p$ is the maximal possible tardiness of both projects in both schedules, *i.e.*, we have $\tau_p^{\pi} < end^{\sigma}(a^p) - due_p$ and $\tau_q^{\pi} < end^{\sigma}(a^p) - due_p$. If:

i. At least one between p and q is early in π, then at lease one between τ_p^{π} and τ_q^{π} is equal to zero. Thus, $\tau_p^{\pi} + \tau_q^{\pi} < end^{\sigma}(a^p) - due_p = \tau_p^{\sigma} + \tau_q^{\sigma}$, so π is better than σ in this criterion.

ii. Both p and q are late in π, then we have $\tau_p^{\pi} = end^{\pi}(b^p) - due_p = end^{\sigma}(b^q) - due_p$ and $\tau_q^{\pi} = end^{\pi}(a^q) - due_q = end^{\sigma}(a^p) - due_q$. We have:
$\tau_p^{\pi} + \tau_q^{\pi} = end^{\sigma}(b^q) - due_p + end^{\sigma}(a^p) - due_q$
$= \tau_p^{\sigma} + end^{\sigma}(b^q) - due_q$
As q is early in σ then $end^{\sigma}(b^q) - due_q < 0$. This implies $\tau_p^{\pi} + \tau_q^{\pi} < \tau_p^{\sigma} = \tau_p^{\sigma} + \tau_q^{\sigma}$, so π is better than σ in this criterion.

c. Both p and q are early in σ, then we have:
$end^{\sigma}(b^q) < end^{\sigma}(a^p) \leq due_p \leq due_q$
and then:
$end^{\pi}(b^p) < end^{\pi}(a^q) \leq due_p \leq due_q$
This implies that both p and q are early in π, so π and σ are equivalent in this criterion.

From the above, we can conclude that π is at least as good as σ in this criterion. ◀

An Efficient Local Search Solver for Mixed Integer Programming

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

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

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

Abstract

Mixed integer programming (MIP) is a fundamental model in operations research. Local search is a powerful method for solving hard problems, but the development of local search solvers for MIP still needs to be explored. This work develops an efficient local search solver for solving MIP, called *Local-MIP*. We propose two new operators for MIP to adaptively modify variables for optimizing the objective function and satisfying constraints, respectively. Furthermore, we design a new weighting scheme to dynamically balance the priority between the objective function and each constraint, and propose a two-level scoring function structure to hierarchically guide the search for high-quality feasible solutions. Experiments are conducted on seven public benchmarks to compare *Local-MIP* with state-of-the-art MIP solvers, which demonstrate that *Local-MIP* significantly outperforms *CPLEX*, *HiGHS*, *SCIP* and *Feasibility Jump*, and is competitive with the most powerful commercial solver *Gurobi*. Moreover, *Local-MIP* establishes 4 new records for MIPLIB open instances.

2012 ACM Subject Classification Mathematics of computing → Integer programming; Theory of computation → Randomized local search; Applied computing → Operations research

Keywords and phrases Mixed Integer Programming, Local Search, Operator, Scoring Function

Digital Object Identifier 10.4230/LIPIcs.CP.2024.19

Supplementary Material *Software (Source Code)*: `https://github.com/shaowei-cai-group/Local-MIP` [31], archived at `swh:1:dir:883191ffb9b4503105cce3e9d3da6d50421956f3`

Funding This work is supported by National Key R&D Program of China (2023YFA1009500).

1 Introduction

Mixed integer programming (MIP) is a fundamental mathematical model in operations research [3], which represents the problem of optimizing a linear objective function under linear constraints, where some variables are restricted to taking integer values. Due to its powerful expressive ability, MIP is widely used both in academic areas and in industrial sectors to model various problems [46, 41]. In previous research, many combinatorial optimization problems can be described by MIP formulation, such as the Boolean satisfiability

[1] Corresponding author

(SAT) and maximum satisfiability (MaxSAT) [13], the knapsack problem [32], the traveling salesman problem (TSP) [37], the job-shop scheduling problem (JSP) [26], and many graph problems [34]. In the industry domain, MIP could model lots of optimization problems in applications, including production planning [38], crew scheduling [16], resource allocation [17], and so on.

Solving MIP is a challenging task as the problem is NP-hard [25, 24]. The solving methods can be divided into two classes: complete methods and incomplete methods. While complete methods aim at computing the exact optimal solution and proving its optimality, incomplete methods aim to obtain high-quality solutions within a reasonable time. In real-world applications, due to the large scale of instances, problems are usually difficult to handle by complete methods. On the other hand, high-quality solutions usually show good usability in practical applications; thus, incomplete methods are of great importance in MIP solving.

Existing MIP solvers, however, rely primarily on complete methods. The most commonly admitted complete method is the branch-and-bound algorithm [27, 28], which iteratively divides the feasible region of solution space and prunes nodes by the bounds of the objective function. Additionally, many techniques were developed, including the cutting plane method [20] and the domain propagation [40], which are often incorporated into the branch-and-bound process in modern MIP solvers [2]. Almost all state-of-the-art MIP solvers are based on the branch-and-bound framework, including the commercial solvers *Gurobi* [21] and *CPLEX* [36], and the academic solvers *SCIP* [5] and *HiGHS* [23].

Local search is a powerful incomplete method for solving challenging problems across various fields in computer science and operations research [22]. It aims to find good solutions quickly and demonstrates significant effectiveness in solving SAT and MaxSAT [6]. Some local search solvers for special cases of MIP have been proposed, such as pseudo-Boolean optimization [4, 11] and integer linear programming [39, 30]. However, the development of local search solvers for MIP is still in its infancy. As far as we know, there is in lack of efficient free local search solvers for MIP that is available to public communities. *Feasibility Jump* [33] is the most related work to us, which proposed a local search algorithm to solve MIP, and won 1st place in the MIP 2022 Competition.[2] However, it only focuses on finding feasible solutions and ignores the objective function, which does not treat MIP's original optimization purpose and does not show a strong capacity for finding high-quality solutions.

We intend to propose an efficient local search solver for MIP. There are two main challenges to achieving this goal: 1. Enhancing adaptability: to handle MIP, local search must accommodate general constraints and variables rather than specific forms. However, solving such general-form problems is more difficult than specific combinatorial problems since less information about problems could be leveraged. Therefore, it is important to enhance the adaptability of the solver to search states. 2. Balance the optimization and the satisfaction: unlike decision problems such as SAT, MIP considers both the optimization of the objective function and the satisfaction of all constraints. But these two factors are sometimes conflicting, thus finding the balance between them is also an important concern.

In this work, we design a novel local search solver for solving general MIP, synthesizing new operators and scoring functions to handle the above challenges. We propose a mixed tight move operator to satisfy general constraints, and a breakthrough operator to surpass the best-found solution by considering the objective value of the dynamically updated best-found solution. Moreover, we propose a two-level scoring function structure to measure the benefit of candidate solutions, which is also related to the dynamically updated best-found objective

[2] https://www.mixedinteger.org/2022/competition/

value. Within each level of the structure, we design the scoring function that synthesizes the score for optimizing the objective function and the score for satisfying constraints. Experimental results demonstrate our solver's excellent performance for solving MIP.

Contributions

We develop an efficient local search solver to find high-quality feasible solutions for MIP.

Firstly, we propose two new operators to adaptively modify variables to optimize the objective function and satisfy constraints, respectively. The breakthrough move operator is proposed to break through the objective value of the dynamically updated best-found solution, and the mixed tight move operator is proposed to satisfy and tighten constraints.

Then, to efficiently guide the search, we first design a new weighting scheme to balance the priority between the objective function and each constraint during the search process. Based on the weighting scheme, we propose a two-level scoring function structure to measure the benefit of each candidate solution. The first level is the progress score, which measures operations by comparing them with the current solution, aiming to make local progress for the current solution. The second level is the bonus score, containing the breakthrough bonus for improving the objective function and the robustness bonus for satisfying constraints stably, being complementary to the progress score. The breakthrough bonus, working together with the breakthrough move operator, enables our solver to select and modify variables according to the dynamically updated best-found solution, improving its adaptability to the global search state. At each level, the scoring function consider both the optimization of the objective function and the satisfaction of constraints.

By putting these together, we develop a new local search solver for MIP called *Local-MIP*. Experiments conducted on seven public benchmarks show the efficiency of *Local-MIP* in finding high-quality feasible solutions for MIP. We compare *Local-MIP* with the state-of-the-art MIP solvers, including the commercial solvers *Gurobi* [21] and *CPLEX* [36], the academic solvers *SCIP* [5] and *HiGHS* [23], and the local search algorithm *Feasibility Jump* [33]. Experimental results show the excellent performance of *Local-MIP*, which significantly outperforms *CPLEX*, *HiGHS*, *SCIP* and *Feasibility Jump*, and is competitive with the most powerful commercial solver *Gurobi*, indicating a significant improvement in the field of local search solver for MIP. Moreover, *Local-MIP* establishes 4 new records for MIPLIB open instances by finding the new best solutions. Additionally, we analyze the effectiveness of proposed strategies and the stability of *Local-MIP* with different random seeds.

2 Preliminaries and Notation

2.1 Formulations of MIP

▶ **Definition 1.** *Mixed Integer Programming (MIP): Given a matrix* $\boldsymbol{A} \in \mathbb{R}^{m\times n}$*, vectors* $\boldsymbol{b} \in \mathbb{R}^m, \boldsymbol{c}, \boldsymbol{l}, \boldsymbol{u} \in \mathbb{R}^n$*, and a subset* $I \subseteq N = \{1, ..., n\}$*. Let* $\boldsymbol{x} = \{x_1, x_2, ..., x_n\}$ *be a set of variables. The mixed integer programming is to solve*

$$min\{\boldsymbol{c}^\top \boldsymbol{x} |\ \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b},\ \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u},\ \boldsymbol{x} \in \mathbb{R}^n,\ x_j \in \mathbb{Z}\ for\ all\ j \in I\} \tag{1}$$

In the above definition, we call $\boldsymbol{c}^\top \boldsymbol{x}$ the objective function, $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$ the general linear constraints, $\boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}$ the global bounds, and $x_j \in \mathbb{Z}\ for\ all\ j \in I$ the integrality constraints. A general linear constraint $\boldsymbol{A_i}\boldsymbol{x} \leq b_i$ is denoted as con_i, and it contains x_j if $A_{ij} \neq 0$. MIP aims to minimize the objective function while satisfying all constraints. A maximization problem and other types of linear constraints can be easily converted into this formulation.

A solution $\boldsymbol{s}$ of a MIP instance is a vector of values assigned for each variable, where s_j denotes the value of x_j. A solution is a **feasible solution** if and only if it satisfies all constraints, including general linear constraints, global bounds, and integrality constraints. For feasible solutions, the lower objective value indicates higher quality.

To facilitate clarity, we establish certain symbols here. $\boldsymbol{s^*}$ denotes the best-found solution in the local search process, and $\boldsymbol{s^{cur}}$ denotes the current solution in each step of local search. $\boldsymbol{s^*}$ is initialized to an empty set and updated whenever a new best feasible solution is found. The objective value of a solution $\boldsymbol{s}$ is denoted as $obj(\boldsymbol{s})$, i.e., $obj(\boldsymbol{s}) = \boldsymbol{c}^\top \boldsymbol{s}$.

2.2 Local Search Algorithm

When solving combinatorial optimization problems, the local search typically starts from an initial solution and iteratively modifies the current solution by changing the value of a variable, in order to search for feasible solutions with high-quality objective values.

In local search, an **operator** defines how to modify variables to generate candidate solutions. When an operator is instantiated by a specifying variable to operate, an **operation** is obtained. For example, for Boolean variables, the standard operator is $flip$, which turns the value of a variable to its opposite, and flip(x_1) is an operation to flip the specifying variable x_1. For the current solution, performing an operation could generate a new candidate solution.

During each step of the local search process, scoring functions are used to evaluate different candidate operations for picking one to execute to update the current solution. Given an operation *op*, a scoring function $score(op)$ measures how good *op* is. An operation *op* is said **positive** if $score(op) > 0$, which indicates that performing *op* could improve the quality of the current solution.

In the following two sections, we propose tailored operators and scoring functions to guide the local search process. Section 5 provides a detailed description of our local search solver.

3 New Operators Tailored for MIP

In this section, we propose two novel local search operators for MIP: the breakthrough move for optimizing the objective function, and the mixed tight move for satisfying constraints.

3.1 Breakthrough Move

To find solutions with high-quality objective values in the local search process, for the first time, we propose the new idea of the breakthrough move operator, which modifies the current solution to improve the quality of the objective function, aiming to break through the objective value of the dynamically updated best-found solution.

▶ **Definition 2.** *Given a variable x_j that appears in the objective function (i.e., $c_j \neq 0$), and a solution $\boldsymbol{s}$ that $obj(\boldsymbol{s}) \geq obj(\boldsymbol{s^*})$, the **breakthrough move operator**, denoted as $bm(x_j, \boldsymbol{s})$, assigns a variable x_j to the threshold value making the objective value better than $obj(\boldsymbol{s^*})$ as possible and keeping x_j's bounds satisfied. Precisely, let ϵ be a very small positive number (e.g., 10^{-6}) for making the objective value strictly better, $\Delta_j = (obj(\boldsymbol{s^*}) - obj(\boldsymbol{s}) - \epsilon)/c_j$, a breakthrough move operation $bm(x_j, \boldsymbol{s})$ assigns x_j to a new value s_j^{new},*

$$s_j^{new} = \begin{cases} min(s_j + \Delta_j, u_j), & if\ c_j < 0, \\ max(\ s_j + \Delta_j, l_j), & else. \end{cases} \tag{2}$$

If x_j is an integer variable, s_j^{new} is rounded to the integer within its global bounds to optimize the objective function, specifically:

$$s_j^{new} = \begin{cases} min(\lceil s_j + \Delta_j \rceil, \lfloor u_j \rfloor), & if\ c_j < 0, \\ max(\lfloor s_j + \Delta_j \rfloor, \lceil l_j \rceil), & else. \end{cases} \tag{3}$$

Given the above definition, the breakthrough move operator adaptively modifies the variable to make the objective value strictly better than the best-found solution while keeping the variable's global bound satisfied. To the best of our knowledge, this is the first time that the idea of an operator to break through the dynamically updated best-found objective value is proposed in the local search for solving combinatorial optimization problems.

▶ **Example 3.** Given a MIP instance whose objective function is $obj = 3x_1 - 2x_2$, where x_1 is an integer variable with global bounds $1 \leq x_1 \leq 5$; x_2 is a real variable with $1 \leq x_2 \leq 3$. Suppose the best-found solution is $\boldsymbol{s^*} = \{s_1 = 3, s_2 = 2\}$, thus $obj(\boldsymbol{s^*}) = 5$; the current solution is $\boldsymbol{s^{cur}} = \{s_1 = 4, s_2 = 1.5\}$, thus $obj(\boldsymbol{s^{cur}}) = 9$. As shown in Figure 1, the operation $bm(x_1, \boldsymbol{s^{cur}})$ refers to assigning x_1 to the threshold value 2 to make objective value better than $obj(\boldsymbol{s^*})$, and the operation $bm(x_2, \boldsymbol{s^{cur}})$ assigns x_2 to its upper bound 3 to optimize the objective function as much as possible while satisfying the variable's bound.

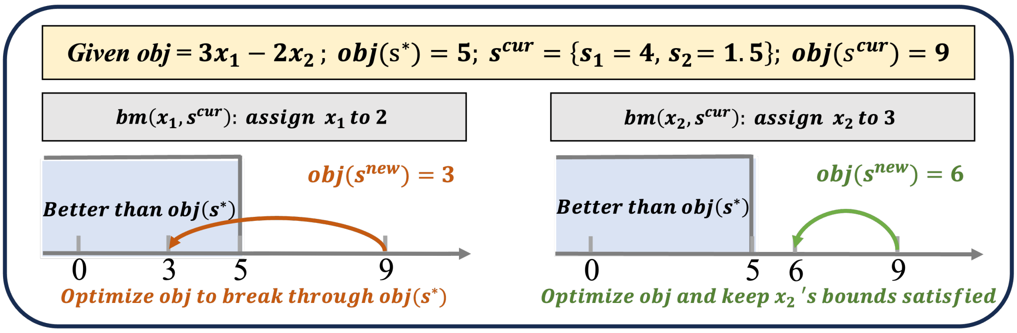

Given $obj = 3x_1 - 2x_2$; $obj(s^*) = 5$; $s^{cur} = \{s_1 = 4, s_2 = 1.5\}$; $obj(s^{cur}) = 9$
$bm(x_1, s^{cur})$: assign x_1 to 2
$bm(x_2, s^{cur})$: assign x_2 to 3
$obj(s^{new}) = 3$
Better than $obj(s^*)$
0
3
5
9
Optimize obj to break through $obj(s^*)$
$obj(s^{new}) = 6$
Better than $obj(s^*)$
0
5
6
9
Optimize obj and keep x_2's bounds satisfied

Figure 1 A graphical explanation of the breakthrough move operator.

3.2 Mixed Tight Move

How to adaptively move variables to satisfy constraints is the key technology when applying local search to solve linear systems.

The Simplex algorithm [12] solves linear programming by tightening constraints to locate solutions in extreme points of the polyhedron. Also, tightening constraints show benefits for finding feasible solutions for integer programming [30]. Both linear programming and integer programming are the subclasses of MIP. Here, based on the insight for tightening constraints, we go one step further and propose an operator suitable for MIP, which can handle both real and integer variables, dubbed as the mixed tight move.

▶ **Definition 4.** *Given a variable x_j, a constraint con_i containing x_j (i.e., $A_{ij} \neq 0$), and a solution $\boldsymbol{s}$, the **mixed tight move operator**, denoted as $\boldsymbol{mtm}(x_j, con_i, \boldsymbol{s})$, assigns x_j to the threshold value making the constraint con_i satisfied and tight while keeping x_j's bounds satisfied. Precisely, let $\Delta_{ij} = (b_i - \boldsymbol{A_i} \cdot \boldsymbol{s})/A_{ij}$, a mixed tight move operation $mtm(x_j, con_i, \boldsymbol{s})$ assigns x_j to a new value s_j^{new},*

$$s_j^{new} = \begin{cases} min(s_j + \Delta_{ij}, u_j), & if\ \Delta_{ij} > 0, \\ max(s_j + \Delta_{ij}, l_j), & if\ \Delta_{ij} < 0, \\ s_j, & else. \end{cases} \tag{4}$$

If x_j is an integer variable, s_j^{new} is rounded to the feasible integer to satisfy and tighten the corresponding constraint, specifically:

$$s_j^{new} = \begin{cases} min(\lceil s_j + \Delta_{ij} \rceil, \lfloor u_j \rfloor), & if\ b_i - \boldsymbol{A_i} \cdot \boldsymbol{s} < 0\ and\ A_{ij} < 0, \\ max(\lfloor s_j + \Delta_{ij} \rfloor, \lceil l_j \rceil), & if\ b_i - \boldsymbol{A_i} \cdot \boldsymbol{s} < 0\ and\ A_{ij} > 0. \\ max(\lceil s_j + \Delta_{ij} \rceil, \lceil l_j \rceil), & if\ b_i - \boldsymbol{A_i} \cdot \boldsymbol{s} > 0\ and\ A_{ij} < 0, \\ min(\lfloor s_j + \Delta_{ij} \rfloor, \lfloor u_j \rfloor), & if\ b_i - \boldsymbol{A_i} \cdot \boldsymbol{s} > 0\ and\ A_{ij} > 0, \\ s_j, & else. \end{cases} \tag{5}$$

According to the above definition, both violated and satisfied constraints can be handled by the mixed tight move operator. The mixed tight move ensures the global bound of related variables is satisfied. Assuming that there is no global bound for each variable, a mixed tight operation of a violated constraint will satisfy the corresponding constraint by choosing the minimal possible change to a variable, which will have the least impact on other constraints that contain the corresponding variable. For a satisfied constraint, a mixed tight operation assigns a variable to its extreme value while ensuring that the corresponding constraint remains satisfied. This allows the exploration for escaping local optimum, as it takes the maximal change of the corresponding variable to the objective function and other constraints.

4 Weighting Scheme and Scoring Functions

As core techniques to guide the local search process, scoring functions measure the benefits of candidate operations for selecting one with the highest score to execute in each step. When solving combinatorial optimization problems, dynamic weighting techniques are commonly used in scoring functions to guide and diversify the search process [29, 11]. In this section, we first design a new weighting scheme for MIP, and then we propose a two-level weighting-based scoring function structure to hierarchically guide search for high-quality feasible solutions.

4.1 Weighting Scheme for MIP

Weighting schemes are usually used to adjust the priority of each constraint by diversified weights in the local search process [45, 43]. Weighting schemes typically increase the weights of constraints that are often violated, hence guiding the search process toward satisfying these constraints. When designing a weighting scheme for combinatorial optimization problems, the main challenge is how to balance the weights between the objective function and each constraint in the search process [45]. For example, in solving MaxSAT, excessive weights of soft clauses would make the local search difficult to satisfy all hard clauses, thereby hindering the capability of finding feasible solutions [10].

Here, we design a new weighting scheme for MIP based on the probabilistic version of PAWS scheme [44, 9, 7, 30], which updates weights according to a smoothing probability sp, and we set $sp = 0.0003$ as mentioned in [7, 30]. Our weighting scheme aims to dynamically balance the weights of the objective function and each constraint by preventing excessive weight while maintaining the distinction between them. It works as follows:

(a) The objective function and each constraint have the attribute of an integral weight, which are denoted as $w(obj)$ and $w(con_i)$, respectively.

(b) Initalization: $w(obj) = 1$ and $w(con_i) = 1$.

(c) When the search process is trapped in a local optimum (i.e., there is no positive operation to select), the weighting scheme is activated, and weights are updated in one of the ways as follows:

- With probability $1 - sp$, if the current solution is feasible, $w(obj) = w(obj) + 1$; otherwise, for each violated constraint con_i, $w(con_i) = w(con_i) + 1$.
- With probability sp, if $obj(\boldsymbol{s^{cur}}) < obj(\boldsymbol{s^*})$ and $w(obj) > 0$, $w(obj) = w(obj) - 1$; for each satisfied constraint con_i whose $w(con_i) > 0$, $w(con_i) = w(con_i) - 1$.

The weighting scheme makes the weights of constraints diverse, reflecting different priorities for considering constraints in search directions. By focusing on constraints that are often violated in local optima, the weighting scheme helps the local search process find feasible solutions. Accordingly, if the search process frequently visits feasible solutions in local optima, then the objective function should be prioritized and will be set with a higher weight, which helps to find higher-quality feasible solutions. Besides, the weight of the objective function decreases if the visited solutions are often infeasible but have a better objective value than the best-found solution, which means the satisfaction of constraints should be more addressed.

4.2 Two-level Scoring Function Structure

Based on the weighting scheme, we propose a two-level scoring function structure, which contains a base scoring function for the first level and a bonus scoring function for the second level. The base scoring function incorporates basic metrics on improving the quality of the objective function and the number of satisfied constraints. The bonus score evaluates the selected operations from a finer view: it both rewards the breakthrough of the objective function, and the robust satisfaction of constraints.

In evaluating an operation, the base scoring function is applied first to select the best operations under it, and then the bonus scoring function is used to evaluate the operations with the best base scores. The composition of the two-level scoring functions takes into consideration both the basic improvement and the adaptive adjusting with the solving process. Moreover, at each level, the scoring functions both measure the objective value and satisfaction of constraints, which balance the aim of optimization and satisfaction.

4.2.1 First Level: Progress Score as the Base Scoring Function

In the first level, we propose the progress score as the base scoring function. The progress score takes a local perspective for improving the quality of the current solution, including the value of the objective function and the satisfaction of each constraint.

MIP aims to minimize the objective. Therefore, the progress score rewards the situation if the objective value is decreased from the current solution, and punishes it if increased.

▶ **Definition 5.** *Given an operation op, and the current solution $\boldsymbol{s}^{cur}$. Let $\boldsymbol{s}^{op}$ be the new candidate solution generated by performing op on $\boldsymbol{s}^{cur}$. The progress score of op for improving the quality of the objective value, denoted as $score^{obj}_{progress}(op)$,*

$$score^{obj}_{progress}(op) = \begin{cases} w(obj), & if\ obj(\boldsymbol{s}^{op}) < obj(\boldsymbol{s}^{cur}), \\ -w(obj), & if\ obj(\boldsymbol{s}^{op}) > obj(\boldsymbol{s}^{cur}), \\ 0, & else. \end{cases} \tag{6}$$

For constraints, the primary goal of MIP is to make them satisfied, thus the progress score rewards the changes from violated to satisfied, and punishes reverse changes. Additionally, for unsatisfied constraints, it is preferred to make them closer to being satisfied. Therefore, the progress score also rewards the proximity to be satisfied and punishes aggravated violations.

▶ **Definition 6.** *Given an operation op, a constraint con_i, and the current solution $\boldsymbol{s}^{cur}$. Let $\boldsymbol{s}^{op}$ be the new candidate solution generated by performing op on $\boldsymbol{s}^{cur}$. The progress score of op for improving the satisfaction of the constraint con_i, denoted as $score^{con_i}_{progress}(op)$,*

$$score^{con_i}_{progress}(op) = \begin{cases} w(con_i), & if\ \boldsymbol{A_i} \cdot \boldsymbol{s}^{op} \leq b_i < \boldsymbol{A_i} \cdot \boldsymbol{s}^{cur}, \\ -w(con_i), & if\ \boldsymbol{A_i} \cdot \boldsymbol{s}^{cur} \leq b_i < \boldsymbol{A_i} \cdot \boldsymbol{s}^{op}, \\ w(con_i)/2, & if\ b_i < \boldsymbol{A_i} \cdot \boldsymbol{s}^{op} < \boldsymbol{A_i} \cdot \boldsymbol{s}^{cur}, \\ -w(con_i)/2, & if\ b_i < \boldsymbol{A_i} \cdot \boldsymbol{s}^{cur} < \boldsymbol{A_i} \cdot \boldsymbol{s}^{op}, \\ 0, & else. \end{cases} \tag{7}$$

Considering the objective function and all constraints, the progress score is defined as below.

▶ **Definition 7.** *Given an operation op, the progress score of op, denoted as $score_{progress}(op)$,*

$$score_{progress}(op) = score^{obj}_{progress}(op) + \sum_{i=1}^{m} score^{con_i}_{progress}(op) \tag{8}$$

According to the above definitions, the progress score measures benefits including improving objective value and satisfying constraints. Performing the operations with positive progress scores indicates overall progress made based on the current solution.

4.2.2 Second Level: Bonus Scoring Function

The progress score introduced above is employed as the elementary scoring function for selecting the operations with the highest score to be applied. However, the progress score has limited capacity to distinguish better operations, according to our preliminary experiments which execute 10000 steps local search on all testing instances, there are on average %47.5 steps where multiple operations with the same greatest progress score are presented. It is important to further evaluate these operations by designing finer scores, to do tie-breaking and award better operations in some metrics. For a similar purpose, previous local search works often use the *age* information of variables as the secondary scoring function [8], where *age* is defined as the number of steps since the last time it is modified, and the operation of oldest age variable is selected to break tie.

However, the *age* scoring function does not consider the characterizations of MIP and cannot effectively guide the search process for MIP according to the experiment. According to Definition 7, the progress score takes the local perspective to make progress for the current solution. Therefore, to further choose an operation among operations with the same best progress score, we consider global properties for the objective function and each constraint, instead of comparing the candidate solution with the current solution.

We design the bonus scoring function as the second level, denoted as $score_{bonus}$, which contains the breakthrough bonus for the objective function and the robustness bonus for constraints, to make distinctions in operations with the best progress score.

Breakthrough Bonus

For the objective function, the bonus scoring function aims to move a variable to break through the global best-found solution, working together with the proposed breakthrough move operator. Thus, we propose the breakthrough bonus to reward the situation that the new candidate solution is better than the dynamically updated best-found solution for the objective value.

▶ **Definition 8.** *Given an operation op, and the best-found solution $\boldsymbol{s}^*$. Let $\boldsymbol{s}^{op}$ be the new candidate solution generated by performing op on the current solution. The breakthrough bonus of op for breaking through the objective value of $\boldsymbol{s}^*$, denoted as $bonus_{break}(op)$,*

$$bonus_{break}(op) = \begin{cases} w(obj), & if\ obj(\boldsymbol{s}^{op}) < obj(\boldsymbol{s}^*), \\ 0, & otherwise. \end{cases} \tag{9}$$

Robustness Bonus

For a constraint $\boldsymbol{A_i} \cdot \boldsymbol{x} \leq b_i$, we take a further step by distinguishing satisfied constraints in terms of the equality between left-hand side $\boldsymbol{A_i} \cdot \boldsymbol{x}$ and right-hand side b_i. For the current solution $\boldsymbol{s^{cur}}$, a special type of satisfied constraint is those $\boldsymbol{A_i} \cdot \boldsymbol{s^{cur}} = b_i$. These constraints, although being satisfied, would become violated more easily than other satisfied constraints because they are fragile and sensitive to the operations of variables contained in the constraint. Therefore, we propose the robustness bonus to reward the operations that keep the strict inequality, i.e. the left-hand side is strictly less than the right-hand side

▶ **Definition 9.** *Given an operation op, a constraint con_i. Let $\boldsymbol{s}^{op}$ be the new candidate solution generated by performing op on the current solution. The robustness bonus of op of the constraint con_i, denoted as $bonus^{con_i}_{robust}(op)$,*

$$bonus^{con_i}_{robust}(op) = \begin{cases} w(con_i), & if\ \boldsymbol{A_i} \cdot \boldsymbol{s^{op}} < b_i, \\ 0, & otherwise. \end{cases} \tag{10}$$

To our knowledge, this is the first time that the distinctions in satisfied constraints are utilized for scoring in local search.

Synthesize the breakthrough bonus for breaking through the best-found solution and the robustness bonus for robust satisfaction, the bonus scoring function that serves in the second level is defined below.

▶ **Definition 10.** *Given an operation op, the bonus score of op, denoted as $score_{bonus}(op)$,*

$$score_{bonus}(op) = bonus_{break}(op) + \sum_{i=1}^{m} bonus^{con_i}_{robust}(op) \tag{11}$$

Complementary to the progress score that compares new candidate solutions with the current solution from a local perspective, the breakthrough bonus considers information from a global perspective to compare and try to break through the global best-known solution, thus extending the selection and modification of variables according to the global search performances. Moreover, the robustness bonus score makes distinctions of inequalities, providing a finer evaluation of the satisfaction of constraints.

Consequently, we combine the progress score and the bonus score in a two-level manner to hierarchically evaluate operations in our solver. This design considers both the local improvement and global improvement for operations and balances the objective optimizing and constraints satisfaction, enhancing the adaptability and efficiency of our algorithm.

5 The Local-MIP Algorithm

Based on the ideas proposed in previous sections, we develop a local search solver for solving MIP called *Local-MIP*. The pseudo-code of *Local-MIP* is outlined in Algorithm 1.

Algorithm 1 The Local-MIP Algorithm.

```
   Input: MIP instance $Q$, time limit $cutoff$
   Output: Best-found solution $\boldsymbol{s}^*$ of $Q$ and its objective value $obj(\boldsymbol{s}^*)$
 1 $\boldsymbol{s^{cur}} \leftarrow$ all variables are set to the value closest to 0 within their global bounds;
 2 $\boldsymbol{s}^* \leftarrow \emptyset$; $obj(\boldsymbol{s}^*) \leftarrow +\infty$;
 3 while $running\ time < cutoff$ do
 4     if $\boldsymbol{s^{cur}}$ is feasible then
 5         Improve the objective value while maintaining feasibility by lift move process;
 6         if $obj(\boldsymbol{s^{cur}}) < obj(\boldsymbol{s}^*)$ then
 7             $\boldsymbol{s}^* \leftarrow \boldsymbol{s^{cur}}$; $obj(\boldsymbol{s}^*) \leftarrow obj(\boldsymbol{s^{cur}})$;
 8     $\boldsymbol{candOP} \leftarrow Get\_Candidate\_Operations(Q, \boldsymbol{s^{cur}})$ ;
 9     $\boldsymbol{candOP}^+ \leftarrow$ operation(s) with the greatest progress score in $\boldsymbol{candOP}$;
10     $\boldsymbol{op} \leftarrow$ an operation with the greatest bonus score in $\boldsymbol{candOP}^+$;
11     $\boldsymbol{s^{cur}} \leftarrow$ a new solution generated by performing $\boldsymbol{op}$ to modify $\boldsymbol{s^{cur}}$ ;
12 return $\boldsymbol{s}^*$ and $obj(\boldsymbol{s}^*)$;
```

We use $\boldsymbol{s^{cur}}$ to denote the current solution which is maintained during the search process, while $\boldsymbol{s}^*$ and $obj(\boldsymbol{s}^*)$ denote the best-found solution and its objective value. In the beginning, each variable of $\boldsymbol{s^{cur}}$ is initialized as the value closest to 0 within its global bounds (Line 1). $\boldsymbol{s}^*$ is initialized as an empty set, and $obj(\boldsymbol{s}^*)$ is initialized as $+\infty$ (Line 2).

After initialization, *Local-MIP* conducts the search process until a given time limit *cutoff* is reached (Lines 3–10). In each step of local search, the best-found solution and the corresponding objective value are updated once a new better feasible solution is discovered (Lines 4-7). The lift move process was proposed in [30] to improve the objective value while maintaining feasibility for a feasible solution of integer programming, a subclass of MIP, which can be easily applied for a feasible solution of MIP, and thus we employ the process once a feasible solution is found in the search process (Line 6). The lift move process improves the quality of the objective function via the best local feasible domain derived by the local domain reduction, until reaches the local optimum [30].

The algorithm generates neighborhood solutions through candidate operations by calling the function $Get\_Candidate\_Operations(Q, \boldsymbol{s^{cur}})$ (Line 8), which is detailedly described in Algorithm 2. Afterward, *Local-MIP* selects the operations with the greatest progress score, if there is only one such operation, it is applied directly; otherwise the second level of the bonus score function is applied and an operation with the greatest bonus score is selected and performed (Line 9-10), to modify the current solution to get a new $\boldsymbol{s^{cur}}$ (Line 11).

Candidate Operations in Each Step

The candidate solutions in each step are constructed by Algorithm 2, containing the breakthrough move operation and the mixed tight move operation proposed in Section 3. For brevity, we denote the breakthrough move as ***bm***, and denote the mixed tight move as ***mtm***.

At the beginning of the search, the top priority is to find the first feasible solution, thus it first considers the positive ***mtm*** operations in violated constraints (Lines 1-3). For violated constraints under an infeasible solution, there are always ***mtm*** operations that improve the satisfaction of these violated constraints. However, there may be no positive operations since their scores are defined on all constraints, since they perhaps reduce the satisfaction of other constraints.

Algorithm 2 Get_Candidate_Operations.

```
   Input: MIP instance $Q$, current solution $s^{cur}$
   Output: Candidate operations set $candOP$
 1 if no feasible solution is found then
 2     if ∃ positive mtm operation in violated constraints then
 3         $candOP$ ← positive mtm operations in violated constraints;
 4 else
 5     if ∃ positive bm operation or mtm operation in violated constraints then
 6         $candOP$ ← positive bm operations ∪ mtm operations in violated
             constraints;
 7     else if ∃ positive mtm operation in satisfied constraints then
 8         $candOP$ ← positive mtm operations in satisfied constraints;
 9 if $op == \emptyset$ then
10     if ∃ positive Boolean flip operation then
11         $candOP$ ← positive Boolean flip operations;
12     else
13         Activate the weighting scheme to update the weights;
14         $candOP$ ← bm operations ∪ random selected mtm operations;
15 return $candOP$;
```

Once any feasible solution has been found, the goal of the search process is transformed into discovering feasible solutions with high-quality objective value. Therefore, the primal candidate operations are the union of positive ***bm*** operations and positive ***mtm*** operations in violated constraints (Lines 5-6). If there are no such positive operations, it tries to construct the candidate operations set with the positive ***mtm*** operations in satisfied constraints (Lines 7-8).

If the algorithm fails to find any positive operation in the previous process, it tries to search for positive Boolean flip operations as candidate operations (Lines 9-11), which we will describe later. Once the above exploration fails, it is indicated that local search falls into the local optimum (Line 12). It first activates the weighting scheme to update the weights of objective function and constraints (Line 13). Afterward, it randomly selects a violated constraint if one exits, and then generates candidate operations set by the union of ***bm*** operations and ***mtm*** operations in the selected constraint (Line 14).

Note that we use the Boolean flip operation in Lines 9-11 to flip the values of Boolean variables. The motivation is due to the importance of Boolean variables in MIP modeling (involved in 96.3% of the MIPLIB instances) and the feature of the flip operation that it could generate more operations than the above operators for Boolean candidate variables. Thus we adopt the flip operator as a special treatment for Boolean variables and apply it to complement the proposed operators to generate more operations.

Besides the main part as shown in Algorithm 1 and 2, we also introduce a forbidding strategy to further improve efficiency. Local search methods may often stuck in suboptimal regions. To address the cycling phenomenon of revisiting the same regions, we employ a forbidding strategy, the tabu strategy [19, 7]. The tabu strategy is directly applied to *Local-MIP*. Once a variable is modified, it forbids the modification for the reverse direction in the following tt iterations, where tt is called tabu tenure, and we set $tt = 3 + rand(10)$ as mentioned in [7].

6 Experimental Evaluations

Here, we introduce the preliminaries and results of our experiments. First, we compare *Local-MIP* with 5 state-of-the-art MIP solvers, including *Gurobi*, *CPLEX*, *SCIP*, *HiGHS* and *Feasibility Jump*. Moreover, we report new records established by *Local-MIP* for 4 open instances in the MIPLIB dataset. Additionally, we analyze the effectiveness of proposed strategies and the stability of *Local-MIP* with different random seeds.

6.1 Experiment Preliminaries

6.1.1 Benchmarks

Our experiments are carried out with 4 benchmarks of the mainstream dataset for MIP, i.e., MIPLIB [18] with hard[3] and open[4] instances. In each instance of MIPLIB, at least one variable is an integer variable. Depending on the type of variables, the MIPLIB instances are classified into the 4 benchmarks.

- **MIPLIB-BP**: the binary programming (66 instances), only contains Boolean variables.
- **MIPLIB-IP**: the integer programming (32 instances), where all variables are integer variables, and at least one variable is not a Boolean variable.
- **MIPLIB-MBP**: the mixed binary programming (195 instances), where all variables are Boolean or real variables, and at least one variable is a real variable.
- **MIPLIB-MIP**: the mixed integer programming (62 instances), where integer variable and real variable both exist and at least one of integer variables is not a Boolean variable.

Additionally, two practical problems are tested: the bin packing and the scheduling problem. They are challenging combinatorial optimization problems, and also have significant applications in real-world industry. We evaluate solvers on 1 standard bin packing benchmark for [15], and 2 standard scheduling benchmarks provided by Taillard's instances [42].

- **BBP**: the Bin Packing problem, This benchmark consists of 60 instances with 500 and 1000 items to pack, encoded by the modeling method proposed in [14].
- **JSP**: the Job-shop Scheduling problem. This benchmark consists of 80 instances encoded by the modeling method proposed in [26].
- **OSP**: the Open-shop Scheduling problem. This benchmark consists of 60 instances encoded by the modeling method proposed in [35].

6.1.2 State-of-the-art Competitors

In Section 6.2, we compare *Local-MIP* with 5 state-of-the-art MIP solvers.

- ***HiGHS*** [23]: an academic solver for large-scale sparse MIP (version 1.6.0).[5]
- ***SCIP*** [5]: one of the fastest academic solvers for MIP (Version 8.1.0, using SoPlex 6.0.4).[6]
- ***Gurobi*** [21]: the most powerful commercial MIP solvers (version 11.0.0). We use both its complete and heuristic versions, denoted by $\text{Gurobi}_{\text{comp}}$ and $\text{Gurobi}_{\text{heur}}$, respectively.[7]
- ***CPLEX*** [36]: a famous commercial MIP solver to solve complex models (version 22.1.0).[8]
- ***Feasibility Jump*** [33]: *FJ* for short, the state-of-the-art local search MIP algorithm, which won 1st place in MIP 2022 Computational Competition.[9]

[3] https://miplib.zib.de/downloads/hard-v22.test
[4] https://miplib.zib.de/downloads/open-v22.test
[5] https://github.com/ERGO-Code/HiGHS
[6] https://www.scipopt.org
[7] https://www.gurobi.com/solutions/gurobi-optimizer/
[8] https://www.ibm.com/products/ilog-cplex-optimization-studio
[9] https://github.com/sintef/feasibilityjump

All of the competitors are downloaded from their websites, and always use default settings. Note that *HiGHS*, *SCIP*, *Gurobi*, and *CPLEX* are trying to do much more than find a high-quality solution quickly. They are also finding optimality certificates and trying to guarantee a feasible solution.

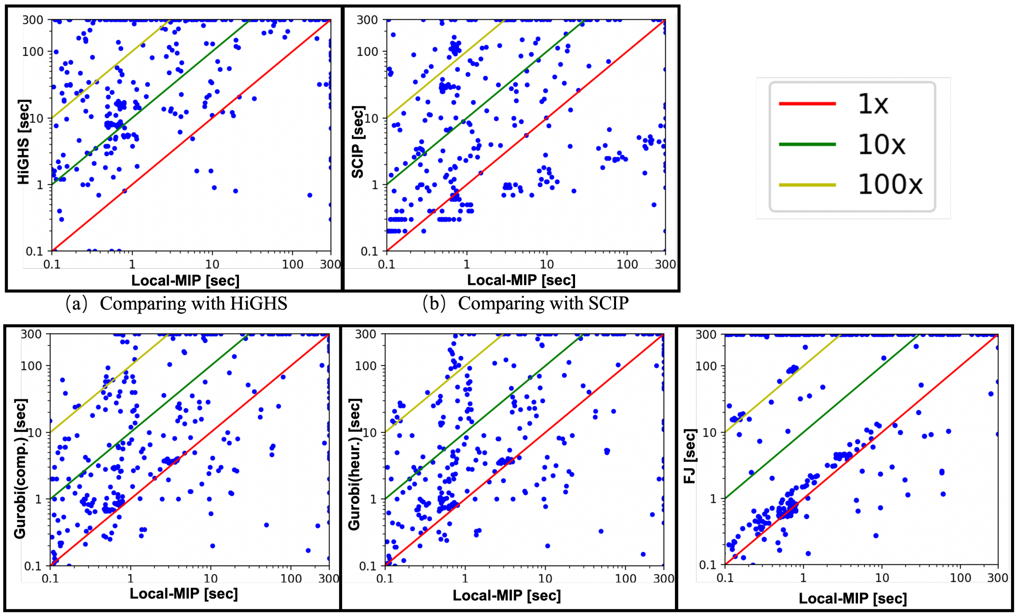
6.1.3 Experiment Setup

Local-MIP is implemented in C++ and compiled in g++ with the '-O3' option. All experiments are carried out on a server with AMD EPYC 9654 CPU and 2048G RAM under the system Ubuntu 20.04.4. we use two metrics to evaluate the performance of each solver for the ability to find high-quality feasible solutions in a reasonable time:

- ***#Feas***: the number of instances where a solver can find a feasible solution within given time limits. This evaluates a solver's ability to find feasible solutions.
- ***#Win***: the number of instances in which the solver yields the best solution among all solvers within time limits. This evaluates the ability to find high-quality feasible solutions.

For both *#Feas* and *#Win*, a larger metric value on a benchmark indicates better performance on the corresponding benchmark. For each instance, each solver is executed by one thread with time limits of 10, 60, and 300 seconds as mentioned in [30]. For each time limit setting in the table, the best performance for the corresponding metric is highlighted in **bold**. Additionally, the number of instances in each benchmark is denoted by *#Inst*. Detailed results and the sourced code are made publicly available on GitHub.[10]

6.2 Comparison with State-of-the-art MIP Solvers



(a) Comparing with HiGHS (b) Comparing with SCIP

(c) Comparing with Gurobi(comp.) (d) Comparing with Gurobi(heur.) (e) Comparing with FJ

Figure 2 Run time comparison on each instance for finding the first feasible solution. Note that the comparison with CPLEX is absent, as it do not provide the exact time for finding solutions.

[10] `https://github.com/shaowei-cai-group/Local-MIP/`

Table 1 Empirical results on comparing *Local-MIP* with the state-of-the-art MIP solvers in terms of *#Feas* and *#Win* within each given time limit.

Benchmark	*#Inst*	*HiGHS*		*SCIP*		*CPLEX*		Gurobi$_{\text{comp}}$		Gurobi$_{\text{heur}}$		*FJ*		*Local-MIP*	
		#Feas	*#Win*	*#Feas*	*#Win*	*#Feas*	*#Win*	*#Feas*	*#Win*	*#Feas*	*#Win*	*#Feas*	*#Win*	*#Feas*	*#Win*
time limit 10 seconds															
MIPLIB-BP	66	6	0	29	2	42	11	**44**	9	**44**	7	42	4	**44**	**31**
MIPLIB-IP	32	7	0	11	2	**17**	4	**17**	**8**	**17**	**8**	11	1	**17**	7
MIPLIB-MBP	195	57	1	80	7	116	31	117	43	**119**	**51**	56	10	103	35
MIPLIB-MIP	62	9	2	21	0	32	10	**37**	11	**37**	13	18	4	35	**15**
BPP	60	9	0	0	0	**60**	0	**60**	0	**60**	0	**60**	0	**60**	**60**
JSP	80	22	0	**70**	25	31	0	10	1	12	8	0	0	45	**36**
OSP	60	48	22	**60**	20	28	7	47	27	42	25	1	0	**60**	**45**
Total	555	158	25	271	56	326	63	332	99	331	112	188	19	**364**	**229**
time limit 60 seconds															
MIPLIB-BP	66	14	0	35	2	43	8	46	10	47	15	**49**	2	48	**28**
MIPLIB-IP	32	12	1	14	1	20	6	20	7	20	**9**	12	1	**21**	6
MIPLIB-MBP	195	96	6	109	2	129	32	**137**	49	134	**65**	62	9	119	23
MIPLIB-MIP	62	15	3	28	1	36	7	41	8	41	**18**	20	3	**43**	17
BPP	60	40	0	20	0	**60**	11	**60**	13	**60**	15	**60**	0	**60**	**33**
JSP	80	41	0	**70**	15	52	1	23	3	26	13	1	0	54	**38**
OSP	60	58	27	**60**	20	30	10	53	37	51	31	9	0	**60**	**42**
Total	555	276	37	336	41	370	75	380	127	379	166	213	15	**405**	**187**
time limit 300 seconds															
MIPLIB-BP	66	22	1	42	4	43	6	47	10	48	**23**	**49**	0	**49**	17
MIPLIB-IP	32	14	2	17	2	21	5	21	10	**22**	**14**	12	1	**22**	4
MIPLIB-MBP	195	115	7	122	7	137	22	150	59	**152**	**69**	67	11	123	14
MIPLIB-MIP	62	24	1	34	1	38	7	44	10	43	**23**	21	3	**45**	16
BPP	60	47	0	40	0	**60**	**31**	**60**	30	**60**	13	**60**	0	**60**	3
JSP	80	49	0	**70**	0	68	1	36	10	34	20	1	0	**70**	**41**
OSP	60	**60**	38	**60**	24	33	13	55	40	**60**	43	19	0	**60**	**44**
Total	555	331	49	385	38	400	85	413	169	419	**205**	229	15	**429**	139

The results of comparison with 4 state-of-the-art MIP solvers are shown in Tables 1.

The ability to find feasible solutions (*#Feas*). *Local-MIP* performs best on 4 benchmarks in the 10s and 60s time limits, and 5 benchmarks in the 300s. For all benchmarks, *Local-MIP* performs best on most benchmarks in the 60s and 300s time limits, and the second most in the 10s. In terms of the total instances of all benchmarks, *Local-MIP* establishes the best performance for all the time limits. In general, this result confirms the powerful ability of *Local-MIP* to find a feasible solution within reasonable times.

The ability to find high-quality feasible solutions (*#Win*). *Local-MIP* performs best on 5 benchmarks in the 10s time limit, 4 benchmarks in 60s, and 2 benchmarks in 300s. For all benchmarks, *Local-MIP* performs best on most benchmarks in the 10s and 60s, and the second most in the 300s. Particularly, *Local-MIP* exhibits the best performance for JSP and OSP benchmarks on all time limits, indicating the advantages of our solver in solving scheduling problems. In terms of total instances, *Local-MIP* consistently establishes the best performance for 10s and 60s, but in 300s, *Gurobi* wins more instances than *Local-MIP*, especially its heuristic version. Overall, for the ability to find high-quality feasible solutions, *Local-MIP* significantly outperforms the two academic MIP solvers *HiGHS* and *SCIP*, and another local search solver *Feasibility Jump*. Moreover, *Local-MIP* performs better than the commercial MIP solver *CPLEX*, and is competitive with the most powerful solver *Gurobi*.

According to Table 1, *Local-MIP* outperforms *Feasibility Jump* on almost all settings, indicating a significant improvement in the field of local search solver for MIP.

Furthermore, the run time comparison on all instances for finding the first feasible solution is presented in Figure 2. For comparison with each solver, there are obviously more instances above the red line, which confirms the powerful solving ability of *Local-MIP*.

6.3 New Records to Open Instances

In the MIPLIB dataset, the instances labeled as open are those that the optimal solution has not yet been solved. The current best solutions known for each open instance are available on the corresponding page on the MIPLIB website. These open instances are representative of the hard-solving problems.

Table 2 *Local-MIP* establishes new records to 4 open instances. #var and #cons denote the number of variables and constraints of the corresponding instance, respectively.

Instance name	#var	#cons	constraint types	Previous best	*Local-MIP*
genus-sym-g31-8	3484	32073	knapsack, precedence, etc.	-21	-23
genus-sym-g62-2	12912	78472	set partitioning, set covering, etc.	-34	-38
genus-g61-25	14380	94735	cardinality, general linear, etc.	-34	-40
neos-4232544-orira	87060	180600	aggregations, variable bound, etc.	17540506.0	15108527.512195

Excitingly, *Local-MIP* established the new best-known solutions for 4 open instances. The new records have been submitted to MIPLIB 2017 and have been accepted; the links to the website are denoted in the footnotes.[11] [12] [13] [14] As shown in Table 2, each of these 4 instances contains multiple different constraint types,[15] simultaneously indicating the powerful solving ability and its extensive applicability.

6.4 Analysis on the Proposed Ideas

Table 3 Comparison bettewn *Local-MIP* and its modified versions. *#better* and *#worse* denote the number of instances where *Local-MIP* obtains better and worse best-found solution, respectively.

Benchmark	*#Inst*	10 seconds		60 seconds		300 seconds		10 seconds		60 seconds		300 seconds	
		#better	*#worse*	*#better*	*#worse*	*#better*	*#worse*	*#better*	*#worse*	*#better*	*#worse*	*#better*	*#worse*
		Comparison with V_{no-bm}						Comparison with $V_{no-weight}$					
MIPLIB-BP	66	**28**	7	**23**	17	**24**	18	**33**	4	**34**	7	**34**	8
MIPLIB-IP	32	**10**	2	**11**	5	**12**	4	**15**	0	**19**	0	**20**	0
MIPLIB-MBP	195	**61**	20	**68**	26	**63**	35	**95**	5	**112**	5	**116**	5
MIPLIB-MIP	62	**22**	7	**27**	8	**27**	8	**35**	0	**40**	1	**41**	0
BPP	60	**59**	0	**59**	0	**58**	0	**35**	10	**56**	0	**60**	0
JSP	80	**32**	13	**45**	8	**60**	10	**45**	0	**54**	0	**70**	0
OSP	60	**48**	5	**49**	1	**46**	3	**60**	0	**60**	0	**60**	0
Total	555	**260**	54	**282**	65	**290**	78	**318**	19	**375**	13	**401**	13
		Comparison with V_{random}						Comparison with V_{age}					
MIPLIB-BP	66	**26**	10	**27**	15	**26**	15	**27**	10	**23**	16	**26**	13
MIPLIB-IP	32	**10**	4	**13**	3	**15**	4	**11**	3	**15**	1	**15**	1
MIPLIB-MBP	195	**69**	27	**80**	30	**71**	43	**66**	28	**68**	42	**71**	41
MIPLIB-MIP	62	**22**	12	**19**	18	**18**	17	**23**	10	**23**	13	**22**	15
BPP	60	**28**	15	11	**27**	11	**34**	**34**	11	16	**29**	13	**25**
JSP	80	**29**	16	**31**	23	**39**	31	**30**	15	**31**	23	**42**	26
OSP	60	**25**	20	**27**	15	**21**	12	**29**	18	**31**	13	**22**	13
Total	555	**209**	104	**208**	131	**201**	156	**220**	95	**207**	137	**211**	134

[11] https://miplib.zib.de/instance_details_genus-sym-g31-8.html
[12] https://miplib.zib.de/instance_details_genus-sym-g62-2.html
[13] https://miplib.zib.de/instance_details_genus-g61-25.html
[14] https://miplib.zib.de/instance_details_neos-4232544-orira.html
[15] https://miplib.zib.de/statistics.html

To verify the effectiveness of the proposed strategies, we conduct comparative experiments on 4 alternative versions of *Local-MIP*, which are obtained as follows.

- V_{no-bm}: to analyze the effectiveness of the breakthrough move operator, we modify *Local-MIP* by removing all the breakthrough move operations in Algorithm 2.
- $V_{no-weight}$: to analyze the weighting scheme, we modify *Local-MIP* by removing the activation of the weighting scheme in Algorithm 2 and making all weights equal to 1.
- V_{random} and V_{age}: to analyze the effectiveness of the bonus score, we modify *Local-MIP* by utilizing the random selection and the *age* strategy instead of bonus score to break ties in Algorithm 1, resulting the versions V_{random} and V_{age}, respectively.

As shown in Table 3, *Local-MIP* significantly outperforms other variations in almost all settings, confirming the effectiveness of the proposed strategies.

6.5 Stability with Repetitive Experiments

To examine the stability of *Local-MIP* which involves randomness, we run *Local-MIP* 10 times with seeds ranging from 1 to 10, and measure the coefficient of variation [1, 11].

For each instance, we calculate the average value AVG and the standard deviation STD for the absolute objective values of the best-found solutions from 10 different seeds. The coefficient of variation of each instance is STD/AVG, and the lower value indicates greater stability. The experimental results are presented in Table 4, where over 85% have a coefficient of variation less than 0.1, indicating *Local-MIP* exhibits stable performance.

Table 4 Experimental results of *Local-MIP* with 10 different seeds on each benchmark, where $\#CV$ denotes the number of instances in each range of the coefficient of variation.

Time limit	#CV			
	$[0, 0.01)$	$[0.01, 0.1)$	$[0.1, 0.5)$	$[0.5, +\infty)$
10 second	**335**	140	56	24
60 second	**322**	150	60	23
300 second	**316**	160	49	30

7 Conclusions and Future Work

In this paper, for solving MIP, we proposed two operators, a weighting scheme, and a tow-level scoring function structure. Based on these novel strategies, we developed an efficient local search solver for MIP. Experimental results demonstrate our solver's excellent performance for solving MIP. Moreover, we establish 4 new records for MIPLIB open instances by finding new best solutions.

For future work, we would like to develop more sophisticated operators and scoring functions to improve the performance of the local search solver for MIP.

References

1 Hervé Abdi. Coefficient of variation. *Encyclopedia of research design*, 1(5), 2010.

2 Tobias Achterberg. SCIP: solving constraint integer programs. *Math. Program. Comput.*, 1(1):1–41, 2009. doi:10.1007/S12532-008-0001-1.

3 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.

4 VL Beresnev, EN Goncharov, and AA Mel'nikov. Local search with a generalized neighborhood in the optimization problem for pseudo-boolean functions. *Journal of Applied and Industrial Mathematics*, 6:22–30, 2012.

5 Ksenia Bestuzheva, Mathieu Besanc$_c$on, Weikun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros M. Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco E. Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. Enabling research through the SCIP optimization suite 8.0. *ACM Trans. Math. Softw.*, 49(2):22:1–22:21, 2023. doi:10.1145/3585516.

6 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.

7 Shaowei Cai, Bohan Li, and Xindi Zhang. Local search for SMT on linear integer arithmetic. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*, volume 13372 of *Lecture Notes in Computer Science*, pages 227–248. Springer, 2022. doi:10.1007/978-3-031-13188-2_12.

8 Shaowei Cai, Chuan Luo, and Kaile Su. Scoring functions based on second level score for k-sat with long clauses. *J. Artif. Intell. Res.*, 51:413–441, 2014. doi:10.1613/JAIR.4480.

9 Shaowei Cai and Kaile Su. Local search for boolean satisfiability with configuration checking and subscore. *Artif. Intell.*, 204:75–98, 2013. doi:10.1016/J.ARTINT.2013.09.001.

10 Byungki Cha, Kazuo Iwama, Yahiko Kambayashi, and Shuichi Miyazaki. Local search algorithms for partial MAXSAT. In Benjamin Kuipers and Bonnie L. Webber, editors, *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA*, pages 263–268. AAAI Press / The MIT Press, 1997. URL: http://www.aaai.org/Library/AAAI/1997/aaai97-041.php.

11 Yi Chu, Shaowei Cai, Chuan Luo, Zhendong Lei, and Cong Peng. Towards more efficient local search for pseudo-boolean optimization. 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 12:1–12:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CP.2023.12.

12 George B Dantzig, Alex Orden, Philip Wolfe, et al. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

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 Maxence Delorme, Manuel Iori, and Silvano Martello. Bin packing and cutting stock problems: Mathematical models and exact algorithms. *Eur. J. Oper. Res.*, 255(1):1–20, 2016. doi:10.1016/J.EJOR.2016.04.030.

15 Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing. *J. Heuristics*, 2(1):5–30, 1996. doi:10.1007/BF00226291.

16 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.

17 Sethavidh Gertphol and Viktor K. Prasanna. MIP formulation for robust resource allocation in dynamic real-time systems. *J. Syst. Softw.*, 77(1):55–65, 2005. doi:10.1016/J.JSS.2003.12.040.

18 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.

19 Fred Glover and Manuel Laguna. *Tabu search*. Springer, 1998.

20 Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs *and* an algorithm for the mixed integer problem. 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 77–103. Springer, 2010. doi:10.1007/978-3-540-68279-0_4.

21 LLC Gurobi Optimization. Gurobi optimizer ref. manual, 2024.

22 Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

23 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.

24 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.

25 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.

26 Wen-Yang Ku and J. Christopher Beck. Mixed integer programming models for job shop scheduling: A computational analysis. *Comput. Oper. Res.*, 73:165–173, 2016. doi:10.1016/J.COR.2016.04.006.

27 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.

28 Eugene L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Oper. Res.*, 14(4):699–719, 1966. doi:10.1287/OPRE.14.4.699.

29 Zhendong Lei and Shaowei Cai. Solving (weighted) partial maxsat by dynamic local search for SAT. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1346–1352. ijcai.org, 2018. doi:10.24963/IJCAI.2018/187.

30 Peng Lin, Shaowei Cai, Mengchuan Zou, and Jinkun Lin. New characterizations and efficient local search for general integer linear programming. *arXiv preprint arXiv:2305.00188*, 2023. arXiv:2305.00188.

31 Peng Lin, Mengchuan Zou, and Shaowei Cai. Local-MIP. Software, version 1.0., swhId: swh:1:dir:883191ffb9b4503105cce3e9d3da6d50421956f3 (visited on 2024-08-19). URL: https://github.com/shaowei-cai-group/Local-MIP.

32 Andrea Lodi and Michele Monaci. Integer linear programming models for 2-staged two-dimensional knapsack problems. *Math. Program.*, 94(2-3):257–278, 2003. doi:10.1007/S10107-002-0319-9.

33 Bjørnar Luteberget and Giorgio Sartor. Feasibility jump: an lp-free lagrangian MIP heuristic. *Math. Program. Comput.*, 15(2):365–388, 2023. `doi:10.1007/S12532-023-00234-8`.

34 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`.

35 B Naderi and M Zandieh. Modeling and scheduling no-wait open shop problems. *International Journal of Production Economics*, 158:256–266, 2014.

36 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.

37 Gábor Pataki. Teaching integer programming formulations using the traveling salesman problem. *SIAM Rev.*, 45(1):116–123, 2003. `doi:10.1137/S00361445023685`.

38 Yves Pochet and Laurence A Wolsey. *Production planning by mixed integer programming*, volume 149. Springer, 2006.

39 SD Prestwich and S Verachi. Constructive vs perturbative local search for general integer linear programming. In *Proceedings of the Fifth International Workshop on Local Search Techniques in Constraint Satisfaction (LSCS)*, 2008.

40 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`.

41 J Cole Smith and Z Caner Taskin. A tutorial guide to mixed-integer programming models and solution techniques. *Optimization in medicine and biology*, pages 521–548, 2008.

42 Eric Taillard. Benchmarks for basic scheduling problems. *european journal of operational research*, 64(2):278–285, 1993.

43 John Thornton. Clause weighting local search for SAT. *J. Autom. Reason.*, 35(1-3):97–142, 2005. `doi:10.1007/S10817-005-9010-1`.

44 John Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira Jr. Additive versus multiplicative clause weighting for SAT. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 191–196. AAAI Press / The MIT Press, 2004. URL: `http://www.aaai.org/Library/AAAI/2004/aaai04-031.php`.

45 John Thornton and Abdul Sattar. Dynamic constraint weighting for over-constrained problems. In Hing-Yan Lee and Hiroshi Motoda, editors, *PRICAI'98, Topics in Artificial Intelligence, 5th Pacific Rim International Conference on Artificial Intelligence, Singapore, November 22-27, 1998, Proceedings*, volume 1531 of *Lecture Notes in Computer Science*, pages 377–388. Springer, 1998. `doi:10.1007/BFB0095285`.

46 Laurence A. Wolsey. Mixed integer programming. In Benjamin W. Wah, editor, *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2008. `doi:10.1002/9780470050118.ECSE244`.

Constraint Modelling with LLMs Using In-Context Learning

Kostis Michailidis
DTAI, KU Leuven, Belgium

Dimos Tsouros
DTAI, KU Leuven, Belgium

Tias Guns
DTAI, KU Leuven, Belgium

Abstract

Constraint Programming (CP) allows for the modelling and solving of a wide range of combinatorial problems. However, modelling such problems using constraints over decision variables still requires significant expertise, both in conceptual thinking and syntactic use of modelling languages. In this work, we explore the potential of using pre-trained Large Language Models (LLMs) as coding assistants, to transform textual problem descriptions into concrete and executable CP specifications. We present different transformation pipelines with explicit intermediate representations, and we investigate the potential benefit of various retrieval-augmented example selection strategies for in-context learning. We evaluate our approach on 2 datasets from the literature, namely NL4Opt (optimisation) and Logic Grid Puzzles (satisfaction), and a heterogeneous set of exercises from a CP course. The results show that pre-trained LLMs have promising potential for initialising the modelling process, with retrieval-augmented in-context learning significantly enhancing their modelling capabilities.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Natural language generation; Computing methodologies → Discrete space search

Keywords and phrases Constraint Modelling, Constraint Acquisition, Constraint Programming, Large Language Models, In-Context Learning, Natural Language Processing, Named Entity Recognition, Retrieval-Augmented Generation, Optimisation

Digital Object Identifier 10.4230/LIPIcs.CP.2024.20

Supplementary Material *Software (Source Code)*: `https://github.com/kostis-init/CP-LLMs-ICL` [37], archived at `swh:1:dir:5e4383ad6c4329796c9f21c51bbff4882dca8271`

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Grant No. 101002802, CHAT-Opt).

Acknowledgements We want to thank the reviewers for their valuable feedback.

1 Introduction

Constraint Programming (CP) is a powerful paradigm for solving complex combinatorial decision-making and optimisation problems. It is widely applicable in various industrial tasks such as scheduling, resource allocation, and various assignment problems [49, 55]. However, utilizing CP requires the translation of real-world problems into a formal model; defining decision variables, constraints, and potentially an optimisation function. This is a complex process, necessitating expertise in the specific application domain, in addition to CP modelling and the semantic and syntactic formalisms of CP solvers or modelling languages.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 20; pp. 20:1–20:27
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The expertise needed for modelling CP problems has been recognised to be a bottleneck for the wider use of CP [16, 15]. This has motivated the development of tools and methods to simplify the modelling process, e.g. high-level modelling languages and automatic reformulation libraries such as MiniZinc [40] or CPMpy [21]. There is also active research on assisting the user through *constraint acquisition*, where either the user has to provide examples of solutions and optionally non-solutions (passive learning) [45, 35, 44] or the user interacts with the system, classifying partial assignments as solutions or not [3, 51, 50].

While these advancements show potential in assisting the user through the modelling process, there is still a gap between a natural language description of the problem and its corresponding CP model. As a result, a number of works have looked at utilizing Natural Language Processing (NLP) techniques in modelling. For example, in [27], a method to detect constraints from textual descriptions of combinatorial problems is presented, while [9] shows how Logic Grid Puzzle (LGP) clues can be formulated in first-order logic expressions. A classification-based approach to detect and rewrite the typical LGP clues as constraints was also explored [24]. Recently, the NL4Opt competition at NeurIPS[1] formulated the challenge of using NLP methods to transform textual descriptions of small linear programming optimisation problems into an LP formulation [47]. The most successful approaches of the competition [25, 22, 41, 18] primarily utilized fine-tuned BERT [12] and BART [29] architectures.

Orthogonally, the rise of new-generation Large Language Models (LLMs) pre-trained on web-scale data, such as GPT-3 [4] and similar models, has enabled the introduction of *coding assistants* [8]. A coding assistant is an LLM that can write a certain piece of code as a response to a user request in natural language. Given that these LLMs are trained on large portions of web data, they can likewise be asked to formulate constraint models for publicly documented constraint-solving systems. Indeed, there are preliminary reports on prompting strategies that present initial successes in CP modelling [52, 1]. On arithmetic and logical reasoning tasks, LLMs have also been used to rewrite the task into a formal specification that can be solved with an SMT Theorem Prover [62]. For LGPs, LLMs have been prompted to detect the constants and then rewrite complex clues and translate them to ASP rules [23]. On the NL4Opt dataset, pre-trained LLMs have been shown to outperform the fine-tuned BERT/BART models [43, 31].

These are promising precursors to employing LLMs to formulate CP models from natural language descriptions of any combinatorial problem, including logic grid puzzles, integer linear programs and more. While this is still a distant dream, it requires evaluating on both satisfaction (CSP) and optimisation problems (COP), for well-defined evaluation measures. We will define these and systematically investigate two orthogonal techniques that are commonly used in LLM approaches: 1) using multiple prompts and intermediate representations to get a final formal CP specification; and 2) using in-context examples and retrieval-augmented example selection techniques. Our goal is to contribute to the development and better understanding of coding assistants for CP modelling.

The contributions of this paper are as follows:

- We present a framework that transforms natural language descriptions of constraint problems into formal solvable CP models using LLMs.
- We compare prompting LLMs to generate solutions, to generating CP models that are then used to solve the problem at hand. We employ different approaches using intermediate representations, and various retrieval-augmented in-context examples selection techniques, including novel variations.

[1] `https://neurips.cc/virtual/2022/competition/50079`

- We define precise evaluation metrics, to standardize the evaluation of the performance of LLMs in modelling CP problems.
- We systematically evaluate the ability of pre-trained LLMs to generate accurate constraint models and solutions for satisfaction and optimisation problems, on 2 datasets from the literature and exercises from a CP course.

2 Background

This section formalizes some fundamental concepts needed for our methodology.

2.1 Constraint Programming

CP is a paradigm for modelling and solving combinatorial problems, by declaratively stating decision variables, their possible values (domains), the constraints that express the relationships between these variables, as well as potentially an objective function to optimise. We now formalize these elements for both satisfaction and optimisation problems.

Constraint Satisfaction Problems

A CSP can be formally defined as a tuple $(\mathbf{X}, \mathbf{D}, \mathbf{C})$, where:

- $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$ is a set of n decision variables.
- $\mathbf{D} = \{D_1, D_2, \ldots, D_n\}$ represents the domains of these variables, where each D_i is a subset of $\mathbb{Z}$ and specifies the allowable values for the decision variable x_i.
- $\mathbf{C} = \{C_1, C_2, \ldots, C_m\}$ is a set of m constraints, where each constraint C_j includes tuples of allowed values for a subset of the decision variables, formally $C_j \subseteq D_{j1} \times D_{j2} \times \ldots \times D_{jk}$ for some subset $\{x_{j1}, x_{j2}, \ldots, x_{jk}\} \subseteq \mathbf{X}$.

Let $sol(\mathbf{C})$ denote the set of all solutions to a CSP. An assignment $\mathbf{a} = \{x_1 = v_1, x_2 = v_2, \ldots, x_n = v_n\}$ is in $sol(\mathbf{C})$ if each $v_i \in D_i$ and if all constraints in $\mathbf{C}$ are satisfied, i.e., $(v_{j1}, v_{j2}, \ldots, v_{jk}) \in C_j$ for each $j \in \{1, 2, \ldots, m\}$.

Constraint Optimisation Problems

A COP extends a CSP by incorporating an objective function f that needs to be minimized or maximized. Formally, a COP is defined as a tuple $(\mathbf{X}, \mathbf{D}, \mathbf{C}, f)$, where $f : \prod_{i=1}^{n} D_i \to \mathbb{R}$ represents the objective function to be optimised. An optimal solution to a COP is an assignment $\mathbf{a}$, such that $\mathbf{a} \in sol(\mathbf{C})$, and $\mathbf{a}$ optimises the objective function f. For maximization, $f(\mathbf{a}) \geq f(\mathbf{b})$, for all $\mathbf{b} \in sol(\mathbf{C})$. For minimization, the inequality is reversed.

2.2 Large Language Models

Large Language Models (LLMs) are deep learning architectures with billions of parameters, based on the Transformer [54, 2]. They are extensively trained on diverse textual data, which allows them to learn and produce complex language patterns [30]. During inference, LLMs generate text by predicting the next token in a sequence, based on the previous tokens. Given an input text (or prompt) p, they produce a sequence of w tokens as follows:

$$LLM(p) = (x_1, x_2, \ldots, x_w), \quad \text{where } x_{t+1} = \arg\max_{x_{t+1}} P(x_{t+1} | p, x_{\leq t}) \tag{1}$$

Here, $x_{\leq t}$ denotes the sequence of tokens generated up to time step t, for all $t \in \{0, 1, ..., w-1\}$. $P(x_{t+1} | p, x_{\leq t})$ is the conditional probability of token x_{t+1} given the initial prompt p concatenated with the preceding tokens $x_{\leq t}$. Also, $x_{\leq 0}$ is the empty sequence and x_1 is the first generated token. The sequence generation process terminates when a predefined stop token is generated or when the maximum context length of the LLM is reached. In this work, we utilize greedy decoding for LLMs and assume the transformation of text into a token sequence and vice versa is done automatically.

In-Context Learning

In-Context Learning (ICL) received significant attention with the introduction of GPT-3 [4]. As a learning paradigm, it enables LLMs to adapt to new tasks at inference time by including examples directly in their input prompts [14, 28, 59]. This capability allows these models to generate responses that are contextually aligned without requiring retraining.

Formally, ICL specifies an ordered set E of k input-output pairs, denoted as $E = \{(i_j, o_j)\}_{j=1}^{k}$. During inference, the examples (or shots) in E are inserted before a new input to influence the model's output towards the expected task-oriented response. This can be represented as $LLM(E \oplus i_{new}) \approx o_{new}$, where o_{new} is the expected response for the input i_{new} based on the patterns outlined by the in-context examples E.

Retrieval Augmented ICL

A challenge in ICL is how to choose the in-context examples, with various studies on how to select [33, 61, 57, 65, 32], order [36, 34, 60], or formulate [58, 20, 26, 38] them. Some works used a predefined static set of examples [58], while others employed a more dynamic approach by selecting them from a database at inference time [63]. The latter, referred to as retrieval-augmented ICL (RAICL), aims to specifically adapt the context to each new input. To implement RAICL, we define a retrieval function R that selects the in-context examples E as follows:

$$E = R(i_{new}, S, k) \tag{2}$$

where i_{new} is a text input, and $S = \{(i_j, o_j)\}_{j=1}^{n}$ is the database of examples that the retrieval function will choose from, with $n \geq k$ and thus $E \subseteq S$. Various strategies – i.e. implementations of the retrieval function R – can be utilised for dynamically selecting in-context examples. They are typically relevance-based metrics with respect to i_{new}. We describe some in 3.4.

3 Methodology

We now formalize the framework of our study; a baseline solution generation method, the proposed modular pipeline to produce an executable CP model, and the RAICL strategies.

3.1 Problem Formulation

The objective is to create a system that solves constraint problems given a natural language description P_{NL}. More formally:

Input: A natural language description of a constraint problem P_{NL}, from which decision variables, constraints, and potentially an objective function must be inferred.

Output: A valid solution or assignment $\mathbf{a}$ of values to the decision variables, satisfying all constraints and optimising the objective function if specified, as described in 2.1.

The methods to derive $\mathbf{a}$ from P_{NL} will be explored through different approaches. Firstly, we will consider the LLMs as constraint problem solvers, where they generate reasoning and ultimately the solution. More fundamentally, we will prompt LLMs to produce solvable CP models, through optional intermediate representations, and use a solver to compute the solution. The steps and pathways that we will explore are visible in Figure 1.

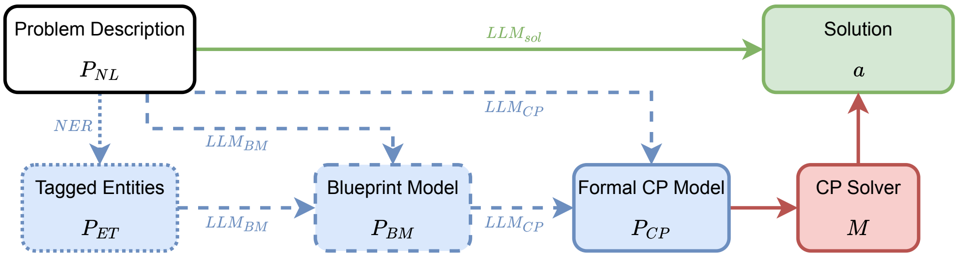


Figure 1 The various pathways for solving constraint problems: direct LLM-based solving (LLM_{sol}), or solvable model generation (LLM_{CP}), optionally including entity tagging (NER) and blueprints (LLM_{BM}) as intermediate representations.

3.2 LLMs as CP Solvers

As a baseline method, we investigate the use of LLMs to directly generate the solution from the natural language problem description P_{NL} of a constraint problem. This approach is based on Chain-of-Thought (CoT) techniques, where the model is prompted to produce a sequence of reasoning steps towards solving a given arithmetic or symbolic reasoning problem [58]. The solution generation process is formalized as follows:

$$\hat{\mathbf{a}} = LLM(E \oplus P_{NL}), \quad \text{where } E = R(P_{NL}, S_{NL-CoT}, k) \tag{3}$$

Here, LLM represents the large language model as defined in (1), E is the set of in-context examples retrieved from the full examples database S_{NL-CoT}, and R is the retrieval function (2). In S_{NL-CoT}, each i_j is a natural language description of a constraint problem and o_j is its CoT reasoning that ends with the solution. The input to the LLM is the concatenation of the in-context examples E with the textual description of the problem P_{NL}, and the output is the generated reasoning sequence $\hat{\mathbf{a}}$. The final solution $\mathbf{a}$ is then extracted from the end of this generated sequence.

3.3 LLMs as CP Modellers

Recognizing the limitations of LLMs in performing arithmetic and logical computations [17], some recent works shifted towards using them to generate formal models or programs [19, 52, 23, 1]. Inspired by these studies, we introduce an additional step in which the LLM is tasked with generating a formal CP model P_{CP} from the given constraint problem description P_{NL}. Our methodology requires that P_{CP} is formatted to fit a predefined CP solver or modelling framework. We define a two-step model-and-solve process as follows:

1. **Model Generation:**

$$P_{CP} = LLM(E \oplus P_{NL}), \quad \text{where } E = R(P_{NL}, S_{NL-CP}, k) \tag{4}$$

The only difference with Equation 3 is the content of the database S_{NL-CP}. Here, each i_j is a constraint problem description in natural language and o_j is its corresponding formal CP model, formatted according to the predefined CP solver or modelling framework. An example of a CP model can be seen in Figure 2, bottom right.

2. **Model Solution:**

$$\mathbf{a} = M(P_{CP}) \tag{5}$$

M represents the predefined off-the-shelf CP solver or modelling framework that computes the solution **a** from the generated formal model P_{CP}.

3.3.1 Blueprint Model Generation

Transforming textual descriptions directly into formal runnable CP models presents a complex challenge. It could be seen as involving two separate non-trivial operations: identifying elements of the constraint problem and generating solver-compatible output. We propose to use different LLM calls and examples for each of the two. We refer to the intermediary result as the blueprint model (P_{BM}). It outlines in plain text the decision variables, constraints and objective function (if applicable) based on the problem description, each of them in both natural language and mathematical notation. An example is provided in Figure 2, bottom left.

We first prompt an LLM to transform the problem description P_{NL} into a blueprint model P_{BM}. Then, we use the description combined with the blueprint to generate the formal CP model. This approach aims to decompose the complex task of direct model generation and also provides an additional interpretable layer. We define this pipeline as follows:

1. **Blueprint Generation:**

$$P_{BM} = LLM(E \oplus P_{NL}), \quad \text{where } E = R(P_{NL}, S_{NL-BM}, k) \tag{6}$$

In S_{NL-BM}, each i_j is a constraint problem description in natural language and o_j is its corresponding blueprint model.

2. **Model Generation:**

$$P_{CP} = LLM(E \oplus P_{NL} \oplus P_{BM}), \quad \text{where } E = R(P_{NL} \oplus P_{BM}, S_{NL-BM-CP}, k) \tag{7}$$

In $S_{NL-BM-CP}$, each i_j is the concatenation of a constraint problem description with its blueprint model and o_j is their corresponding formal CP model.

3. **Model Solution:** Same as (5).

3.3.2 Named Entity Recognition

As part of the first subtask of the NL4Opt competition, there have been numerous approaches for accurately tagging and identifying linear optimisation entities from textual descriptions [56, 41, 13]. Based on these works, we integrate Named Entity Recognition (NER) into our methodology to systematically identify and extract decision variables, parameters, constraints, and objective keywords from natural language descriptions P_{NL} of constraint problems. Parameters are fixed coefficients or constants in the formal definition of the CP model. For example, in Figure 2 the numeric values in the description are parameters of the problem.

This integration of entity tags aims to improve the available information for constructing blueprints and formal CP models as follows:

1. **Entity Tagging:** $P_{ET} = NER(P_{NL})$, where NER is an automated system designed to detect and label specific entities relevant to CP problems within the text of P_{NL}. This step could also be performed with an LLM, but we chose NER4Opt as a specialized framework trained for entity tagging in an optimisation context [11].

A retired professor wants to invest up to $50000 in the airline and railway industries. Each dollar invested in the airline industry yields a $0.30 profit and each dollar invested in the railway industry yields a $0.10 profit. A minimum of $10000 must be invested in the railway industry and at least 25% of all money invested must be in the airline industry. How to maximize the professor's profit?

Decision Variables:

- Amount invested in the airline industry: *Airline*
- Amount invested in the railway industry: *Railway*

Constraints:

- Total investment should not exceed 50000 dollars: $\mathit{Airline} + \mathit{Railway} <= 50000$
- Minimum investment of 10000 dollars in the railway industry: $\mathit{Railway} >= 10000$
- At least 25% of all money invested must be in the airline industry: $\mathit{Airline} >= 0.25 \times (\mathit{Airline} + \mathit{Railway})$

Objective:

- Maximize profit ($0.30 profit per dollar invested in the airline, $0.10 profit per dollar invested in the railway industry): $0.30 \times \mathit{Airline} + 0.10 \times \mathit{Railway}$

```python
from cpmpy import Model, intvar

# Decision Variables
Airline = intvar(0, 1_000_000)
Railway = intvar(0, 1_000_000)

# Constraints
m = Model()

# Total investment
m += Airline + Railway <= 50000
# Minimum investment in railway
m += Railway >= 10000
# Minimum investment in airline
m += Airline >= 0.25 * (Airline + Railway)

# Objective: Maximize profit
m.maximize(0.3 * Airline + 0.1 * Railway)

m.solve()
```

Figure 2 An example of textual description (top), blueprint (bottom left), and formal CP model written with the CPMpy library (bottom right) for the investment problem from NL4Opt.

2. **Blueprint Generation:**

$$P_{BM} = LLM(E \oplus P_{NL} \oplus P_{ET}), \quad \text{where } E = R(P_{NL} \oplus P_{ET}, S_{NL-ET-BM}, k) \tag{8}$$

In $S_{NL-ET-BM}$, each i_j is the concatenation of a constraint problem description with a textual representation of its tagged entities and o_j their corresponding blueprint model.

3. **Model Generation:**

$$P_{CP} = LLM(E \oplus P_{NL} \oplus P_{ET} \oplus P_{BM}),$$
$$\text{where } E = R(P_{NL} \oplus P_{ET} \oplus P_{BM}, S_{NL-ET-BM-CP}, k) \tag{9}$$

Here, in $S_{NL-ET-BM-CP}$ each i_j is a constraint problem description concatenated with its tagged entities and blueprint model, and o_j is its corresponding CP model.

4. **Model Solution:** Same as (5).

3.4 In-Context Examples Selection

A significant component of the presented pipelines is the retrieval function R, as it is responsible for the dynamic selection of in-context examples from the full database S (2.2).

Static & Random Strategies

As a baseline method, we can employ a traditional approach where the same predefined examples are selected for any input [58]. These examples are selected in advance and remain unchanged, providing a consistent basis for evaluation. In our research, we retrieve the first k example pairs from S, formalized as:

$$R(i_{new}, S, k) = \{S_j \mid j \in \{1, 2, \dots, k\}\} \tag{10}$$

where S_j represents the j-th example tuple in the database.

To provide a more stochastic view, we will also consider random selection where we use the same selection strategy but on a randomly shuffled database.

Semantic Similarity (SIM)

The first retrieval-augmented strategy that we will utilize selects examples that are semantically close to the input [6]. As it provides context that is relevant to the current query, it has been shown to improve ICL on various tasks [33]. We define semantic similarity *Sim* between two texts using the cosine similarity between their vector embeddings:

$$Sim(\text{text}_1, \text{text}_2) = \frac{\vec{v}(\text{text}_1) \cdot \vec{v}(\text{text}_2)}{\|\vec{v}(\text{text}_1)\| \|\vec{v}(\text{text}_2)\|} \tag{11}$$

where $\vec{v}(\text{text})$ is the vector embedding of a text. Vector embeddings are numerical representations of tokens that capture semantic meanings, often derived from the embedding layers of LLMs. For a new input i_{new}, we first compute its embedding vector and then the retrieval function R selects the k most semantically similar examples from S as follows:

$$R(i_{\text{new}}, S, k) = \{(Sorted(i_{\text{new}}, S))_j \mid j \in \{1, 2, \dots, k\}\}, \tag{12}$$

$$Sorted(i_{\text{new}}, S) = \{S_j \mid Sim(i_{new}, i_j) \geq Sim(i_{new}, i_{j+1}) \quad \forall j \in \{1, 2, \dots, n-1\}\}, \tag{13}$$

where i_j is the input element of the j-th example pair (i_j, o_j) in S (2.2).

Maximal Marginal Relevance (MMR)

To add more information and variety in the context, we will also consider the MMR metric [5, 63]. It selects examples that balance relevance to the input with diversity within the chosen set. This balance is achieved by selecting an example S_j that maximizes both its relevance to the input and its difference from previously selected examples:

$$\arg\max_{S_j \in S \setminus T} \left[\lambda \cdot Sim(i_{new}, i_j) - (1 - \lambda) \cdot \max_{S_t \in T} Sim(i_j, i_t) \right] \tag{14}$$

Here, λ is the hyperparameter that controls diversity, *Sim* is the similarity measure (11), S is the total set of examples, and T is the set of already selected examples. The retrieval function R will first select the most similar example from the database S, and select the remaining $k - 1$ according to (14).

Last-Similar Variations

We also introduce variations of example selection strategies, inspired by the recency effect [10, 53, 64]. This suggests that the content positioned towards the end of an input sequence has a more significant influence on the output of an LLM [34]. Based on this insight, the next proposed strategies involve placing the most semantically similar example last in the sequence to exploit this recency bias.

- **Reversed Semantic Similarity (R-SIM)**: The retrieved examples based on the Semantic Similarity metric are reordered so that the most relevant to the current problem is last. This was also explored in Liu et al. [33].
- **Reversed Maximal Marginal Relevance (R-MMR)**: Similarly with R-SIM, the original MMR order is reversed, so that the most similar example is placed last.
- **Last-Similar, Rest-Random (LSRR)**: Places the most similar example from S last, and the rest are selected randomly.

4 Experiments

This section outlines our experimental framework, devised to evaluate the effectiveness of using LLMs to convert natural language descriptions into formal CP models. We seek to answer the following research questions:

Q1: How does each intermediate representation impact the efficiency of LLMs in generating CP models, and how do they compare to direct solution generation from the LLMs?

Q2: How do different in-context example selection strategies influence the correctness of the generated CP models and their solutions?

Q3: How many in-context examples should be used, depending on the type of the problems?

Q4: How effectively can LLMs with RAICL generate CP models for a small dataset of problems that human students learn to solve?

4.1 Setup & Datasets

For our experimental setup, we utilized Python 3.9 along with several specialized tools and libraries[2]. For entity tagging, we employed NER4Opt [11] and for CP modelling we used CPMpy [21]. To implement and test RAICL strategies, we used LangChain [7] and Chroma DB[3], and the OpenAI API[4] was used for accessing gpt-3.5-turbo-0125 and vector embeddings. Our framework is evaluated on the following datasets:

- **NL4Opt [47]:** This dataset includes NL descriptions of linear optimisation problems, with 289 test and 713 training instances, such as in Figure 2. We use the test instances for evaluation, and the training instances to compose the examples database S.
- **Logic Grid Puzzles (LGPs) [39]:** Consists of 50 train and 100 test instances featuring logical puzzles described with clues and entities; these can be expressed as CSPs. As above, we use the test instances for evaluation and the training instances for S.
- **Mixed CP Dataset:** Comprises 18 diverse CP problems (a mix of 13 CSPs and 5 COPs) drawn from a university-level CP modelling course, arranged by increasing complexity. Due to its small size, we use a leave-one-out strategy for evaluation, testing each problem individually while utilizing the rest for S.

To facilitate both RAICL and evaluation, we extended the datasets by generating entity tags, blueprint models and formal CPMpy models. This generation process is described in Section 4.2. For the mixed CP dataset, we manually curated and assessed all the blueprint and formal CP models. For more information regarding these datasets, such as the average number of decision variables, constraints, and more, refer to Appendix E.

4.2 Data Annotations Generation

For NL4Opt and LGPs, we utilized NER4Opt [11] to generate the entity tags for all instances (3.3.2). Then, we created detailed CoT-including solutions for 4 instances to evaluate the baseline of our methodology (3.2). Finally, we manually created the blueprint and formal CP models of their first six instances, ensuring clarity and correctness. For generating the remaining ground-truth blueprint and CP models, we employed gpt-4-0125-preview [42] with static ICL using the manually produced ones. We ensured the correctness of the generated

[2] The code is available at `https://github.com/kostis-init/CP-LLMs-ICL`.

[3] `https://docs.trychroma.com/`

[4] `https://platform.openai.com/docs/api-reference`

CP models – correcting them when needed – as follows: For NL4Opt, we asserted that the generated models are equivalent (18) and produce the same solutions with the already existing canonical formulations. This is a deterministic procedure, as the linear constraints can be automatically transformed into a CP model. For LGPs, we validated that the solutions produced by the generated models match the ground-truth solutions already present in the dataset.

4.3 Evaluation

Defining a unified evaluation framework for modelling varying CP problem types from textual descriptions is not trivial. Evaluations at either the solution, constraint, or model level must include some form of mapping between the decision variables (and/or their values) of the predictions and those of the ground truth. We implemented a process that identifies the best match for each decision variable by first considering exact matches, then prefixes, substrings, and finally, a composite textual and numerical similarity metric.

Additionally, LLMs generate executable CP models that may contain syntax errors[5]. Therefore, we also track and report the number of models containing at least one error, denoted as **#Err** in the results. To ensure precise evaluation, we adopt a strict criterion: if a generated CP model cannot be executed due to syntax errors, or if the decision variables of the predicted model cannot all be mapped to those of the ground truth models, then the instance is considered incorrect on all metrics.

We now present three separate accuracy measures, each focused on different aspects.

Solution Accuracy

The solution accuracy metric evaluates the correctness of the solutions produced by the generated CP models compared to the ground truth solutions. A solution is deemed correct if it satisfies all constraints and, if applicable, achieves the optimal objective value as defined in the ground truth CP model. We formalize the solution accuracy as follows:

$$acc_{sol} = \frac{\sum_{i=1}^{N} valid(\mathbf{a}_i, true_i)}{N} \tag{15}$$

where N is the total number of test instances, $\mathbf{a}_i$ represents the solution derived from the LLM or the predicted CP model $pred_i$; $valid(\mathbf{a}_i, true_i)$ equals 1 if $\mathbf{a}_i \in sol(\mathbf{C}_{true_i})$, and for COPs $f_{true_i}(\mathbf{a}_i)$ must be optimal. We do not compute all solutions, but simply check whether $\mathbf{a}_i$ is a satisfying solution and optionally whether its objective value equals the optimal value.

Declaration Accuracy

As part of the evaluation metrics used in the NL4Opt competition [47], declaration accuracy measures the percentage of individual declarations (constraints and objectives) predicted accurately with respect to the ground truth declarations. It is calculated using the formula:

$$acc_{decl} = 1 - \frac{\sum_{i=1}^{N} \min(\mathrm{FP}_i + \delta_i, Q_i)}{\sum_{i=1}^{N} Q_i} \tag{16}$$

where N is the total number of test instances and Q_i is the total number of declarations, in the i-th ground-truth model. FP_i denotes the number of false positives, and δ is the difference in the number of constraints between the ground-truth model and the predicted model, only counted when the predicted model has fewer constraints.

[5] An example is demonstrated in Appendix C.2

To calculate FP_i, we count all the declarations of the predicted model for which an *equivalent* declaration was not found in the ground-truth model. Two constraints from different models are considered *equivalent* if they imply each other for the selected decision variable mapping. For verifying the correctness of a linear objective function, equivalence is established by asserting that the coefficients of the mapped variables are identical.

Model Accuracy

Model accuracy assesses the semantic correctness of the generated CP models relative to the ground truth models. It quantifies how well the entire set of constraints and the logical structure in a predicted model capture the problem as defined in the ground truth. This metric is defined as:

$$acc_{model} = \frac{\sum_{i=1}^{N} equiv(pred_i, true_i)}{N} \tag{17}$$

where $equiv(pred_i, true_i)$ equals 1 if the predicted model $pred_i$ is logically equivalent to the ground truth model $true_i$ for a specified mapping of variables. Logical equivalence is confirmed if the constraints of the predicted model imply and are implied by the constraints of the ground truth model, ensuring a bidirectional logical consistency:

$$equiv(pred, true) \Leftrightarrow ((\bigwedge pred \implies \bigwedge true) \wedge (\bigwedge true \implies \bigwedge pred)) \tag{18}$$

This verifies that the complete set of constraints in one model logically corresponds to those in the other, confirming their semantic equivalence. The equivalence algorithm is described in Appendix B.

4.4 Results

This section details the results and findings from our experiments with the proposed methods and systematically explores the proposed research questions. We do not present any 0-shot attempts, primarily due to the infeasibility of producing runnable CPMpy models without examples in the prompt context[6]. For the core experiments, we employed gpt-3.5-turbo-0125 with a temperature value of 0. Additional results on other LLMs are available in Appendix A.

4.4.1 Intermediate Representations

An incremental analysis is necessary to understand the effect of each intermediate component in our pipeline. As an initial baseline, we employ static (3.4) in-context examples selection, with $k = 4$ examples in the context. The results are shown in Table 1.

Using LLMs as CP modellers, instead of direct combinatorial problem-solvers, proved to be essential for obtaining substantial performance across the datasets. Even if the LLM is instructed to solve the problem with 4 pairs of descriptions and CoT solutions in the context, the direct solving approach results in significantly lower solution accuracy.

Interestingly, appending NER in the pipeline of the LGP instances nearly doubled the number of errors and dropped accuracy compared to direct CP modelling. Our qualitative analysis showed that decision variables were often misclassified as parameters, leading to inconsistencies and irrelevant context at subsequent steps in the pipeline [48]. Conversely, in the NL4Opt instances, where NER tagging was more accurate, the performance improvements were comparable to those achieved with BM. The inclusion of BM was slightly more beneficial

[6] An example with errors is demonstrated in Appendix C.1

Table 1 Comparison of our methods for both NL4Opt (#289 test instances) and LGPs (#100 test instances). Configuration: gpt-3.5-turbo-0125, 4-shot static in-context examples selection.

Dataset	Method	#Err	Accuracy (%)		
			Solution	Declaration	Model
NL4Opt	Direct	-	11.46	-	-
	CP	**7**	81.31	87.81	79.24
	+ BM	8	84.43	**89.93**	**82.01**
	+ NER	8	**85.47**	88.60	80.62
LGPs	Direct	-	9.36	-	-
	CP	**11**	57.00	**80.45**	55.00
	+ BM	18	**58.00**	70.69	**58.00**
	+ NER	20	54.00	67.77	50.00

than directly generating CP models, but it produced a larger number of errors in the LGP dataset. This can be attributed to the larger structure of the intermediate models in LGPs[7]. As such, LGP blueprints are prone to be generated with errors, which would introduce irrelevant context in the LLMs, similar to NER.

Comparing the results of the two datasets, we can see a discrepancy that can be attributed to their different nature and modelling difficulties. On the one hand, NL4Opt contains simple linear optimisation problems with around 2 decision variables and 3 constraints on average per instance. LGPs, in contrast, include 12 decision variables and 4 clues on average, including complex constraints such as *all different* and *pairwise exclusive disjunction*.

Overall, the best approach for both datasets is the blueprint and CP model generation, thus we will utilize this to investigate RAICL in the following experiments.

4.4.2 Examples Selection Strategy

To understand the impact of RAICL and different in-context example selection strategies, we evaluated the presented methods with a fixed number of $k = 4$ in-context examples. The results are shown in Table 2.

In both datasets, the dynamic retrieval algorithms consistently outperform static and random selection methods, underlining the importance of semantic relevance in the prompt context.

As proposed, applying the recency effect by placing the most similar example last in the context considerably improves accuracy. This is mostly evident in the large accuracy difference between the random and LSRR methods across both datasets; and additionally between the reversed retrieval methods in the LGPs with direct CP modelling. In this configuration, LSRR also outperforms MMR and R-MMR, which suggests that adding even more diversity further improves performance in LGPs.

In the NL4Opt dataset, the dynamic methods showed relatively similar performance, with all of them achieving higher accuracies than the static and random strategies. Declaration accuracy in this dataset is on par with the LP-specialized top-ranked approaches in the NL4Opt competition[8] [18]. In LGPs, employing retrieval-augmented ICL not only improved accuracy over all metrics but also significantly diminished the number of errors. This showcases that relevance in the context is important for generating valid and runnable code.

[7] An example is available in Appendix F

[8] `https://nl4opt.github.io/`

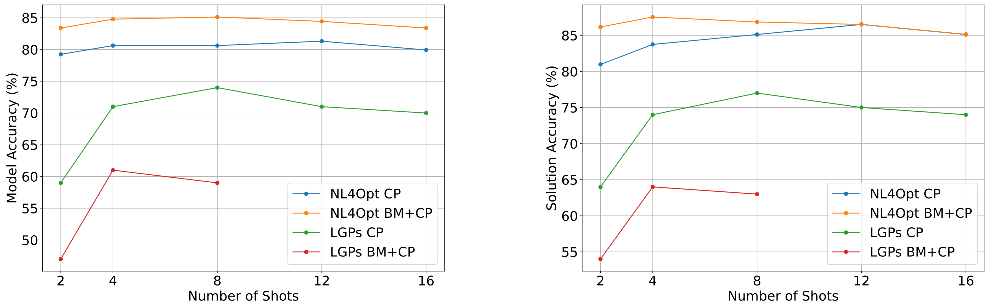
Table 2 Comparison of example selection strategies for NL4Opt and LGPs datasets, gpt-3.5-turbo-0125, 4-shot ICL, $\lambda = 0.5$.

Dataset	Method	CP				BM + CP			
		#Err	Accuracy (%)			#Err	Accuracy (%)		
			Sol.	Decl.	Mod.		Sol.	Decl.	Mod.
NL4Opt	Static	7	81.31	87.81	79.24	8	84.43	89.93	82.01
	Random	11	77.16	85.60	75.09	14	78.55	83.13	77.16
	SIM	8	**85.12**	87.72	80.28	**5**	86.16	90.02	84.08
	R-SIM	11	83.39	88.16	80.62	11	85.47	89.31	83.04
	MMR	8	84.08	87.99	**80.97**	7	86.51	**90.81**	**85.47**
	R-MMR	**6**	83.74	87.10	80.62	8	**87.54**	89.93	84.78
	LSRR	12	83.74	**88.34**	80.62	11	83.39	88.25	82.70
LGPs	Static	11	57.00	80.45	55.00	18	58.00	70.69	58.00
	Random	14	66.00	76.75	62.00	8	59.00	81.51	52.00
	SIM	9	68.00	85.34	66.00	9	66.00	81.24	63.00
	R-SIM	**4**	72.00	89.83	69.00	19	61.00	73.18	58.00
	MMR	10	66.00	83.09	64.00	7	63.00	83.75	58.00
	R-MMR	6	74.00	87.98	71.00	7	64.00	81.77	61.00
	LSRR	**4**	**76.00**	**89.96**	**72.00**	**5**	**75.00**	**86.39**	**70.00**

The overall results indicate that RAICL outperforms static selection in generating formal CP models by a large margin for both datasets. Additionally, the results of R-MMR and LSRR highlight the importance of both the last in-context example and the context diversity in the prompt.

4.4.3 Number of In-Context Examples

The results shown in Figure 3, illustrate how the number of in-context examples influences the model and solution accuracy in NL4Opt and LGPs, both for including blueprint model generation and not. We selected R-MMR as a retrieval strategy as it demonstrated consistently fewer errors and high accuracy across the two datasets.



(a) Model accuracy changes with the number of shots.

(b) Solution accuracy changes with the number of shots.

Figure 3 Comparison of model and solution accuracies with a varying number of in-context examples. Config: gpt-3.5-turbo-0125 R-MMR ($\lambda = 0.5$).

In the NL4Opt dataset, we observe a progressive gain in accuracy metrics as the number of in-context examples increases, reaching an optimal performance at 12 shots when not including BM. However, beyond this point, a further increase shows a slight performance degradation. Including the blueprint model caused this decline to start earlier, at 4–8 shots, hinting towards a ceiling effect on the size of the prompt context. The LGPs dataset presents a similar picture but with the BM inclusion consistently underperforming in comparison to the direct CP model generation. As in NL4Opt, including blueprint models peaks at a smaller k value (4) than when directly generating the CP model (peaking at $k = 8$). It should be noted that with blueprint models, the examples that generate the CP model are much larger as they also contain these blueprints.

Based on these observations, a balance is needed between providing adequate context and avoiding information overload, which can influence the quality of the provided answers. The number of in-context examples should also be adapted according to their size so that the prompt does not become overly dense. Notably, the LGPs with the blueprint model generation could not accommodate more than 8 in-context examples due to their length and the limitations imposed by the 16k context window of gpt-3.5-turbo-0125.

4.4.4 Mixed CP Dataset

We complete this section by assessing the potential of LLMs to generate correct CP models from a dataset with diverse and more complex constraint problems. We focused exclusively on solution accuracy since there can be multiple different CP models for such complex problems, which may involve varying selections of decision and auxiliary variables. As a result, the variable mapping required to use the other metrics proved impractical.

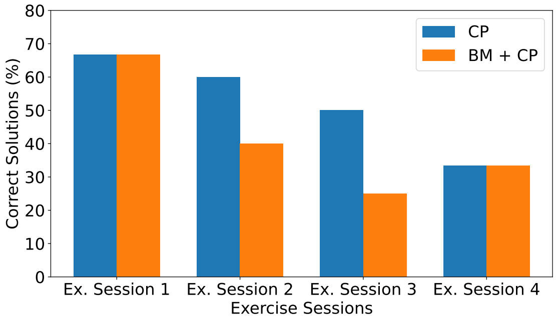
For these problems, we included additional instructions in the problem descriptions, specifying how solutions should be printed and formatted by the generated CP model code. This allowed us to directly validate this output with the ground truth model, ensuring that the solutions can be assessed without needing to map decision variables. Also, we chose R-SIM as the selected retrieval-augmented technique to balance the diversity of the dataset. The results are shown in Table 3, with supplementary results in Appendix D.

While adding more in-context examples improves the accuracy of the produced solutions, blueprint inclusion degrades performance. This suggests that its integration might be less beneficial for complex problems, in which the generated blueprints are even more likely to contain errors. Additionally, using retrieval-augmented R-SIM does not seem to improve much over static selection. Due to the small dataset size, static and R-SIM often share some of the same examples. This limited size and high diversity of the dataset proves challenging for LLMs, with only a bit over half of the exercises solved.

Table 3 Solution Accuracy across different numbers of in-context examples for both static and R-SIM retrieval strategies in the mixed CP dataset. LLM: gpt-3.5-turbo-0125.

#Shots	Solution Accuracy (%)			
	Static		R-SIM	
	CP	BM + CP	CP	BM + CP
2	33.33	16.67	50.00	38.89
4	50.00	50.00	55.56	38.89
8	50.00	44.44	55.56	44.44
12	55.56	50.00	61.11	44.44
16	50.00	-	61.11	-

As these problems are based on a university course, we arranged them in increasing difficulty based on the exercise session they were covered in. Figure 4 shows that as the difficulty of the problems increases, there is also an analogous decrease in accuracy, pointing to the challenge LLMs face when required to produce CP models for complex problems.



Exercise sessions:

1. 6 problems: Five Floors, Bank Card, Guards and Apples, Magic Square, Thick as Thieves, Money Change.
2. 5 problems: Color Simple, Movie Scheduling, Subset Sum, Subsets 100, Maximal Independent Sets.
3. 4 problems: Exodus, People in a Room, Kidney Exchange, Farmer and Cows.
4. 3 problems: Grocery, Climbing Stairs, Hardy 1729.

Figure 4 Solution accuracy per session when increasing difficulty. Config: R-SIM, 12-shot ICL.

5 Conclusion & Future Work

In this paper, we explored the potential of leveraging pre-trained Large Language Models to model CP problems from textual problem descriptions. We introduced and systematically evaluated LLM-based approaches, investigating the use of intermediate representations with multiple prompting steps, to formulate an executable CP model specification. We utilized in-context learning and retrieval techniques for the in-context examples. As the use of LLMs for modelling constraint problems has not been explored in much depth in the literature, we also focused on both augmenting existing datasets and defining evaluation measures. We augmented the NL4Opt and LGP datasets, with intermediate representations and formal model specifications in CPMpy, and created a small course-based diverse CP dataset. We presented an evaluation framework with precise metrics for the correctness of solutions, the semantic equivalence of the individual constraints, as well as the overall constraint model.

Our experiments demonstrated how challenging it is for LLMs to solve combinatorial problems directly. However, using them to write constraint models was significantly more successful. Including NER, as proposed in the NL4OPT challenge, did not always improve the quality of the final models. On the other hand, using a human-interpretable blueprint modelling before the executable model generation was beneficial only for the simple linear optimisation problems of NL4Opt. We observed that such intermediate steps can introduce additional errors, due to incorrect tagging and irrelevant prompt context. What mostly improved the performance of LLMs is: a) retrieval-augmented selection of the in-context examples, based on a balance of relevance to the current input and context diversity, and b) increasing the number of such examples up to a certain threshold per dataset. We also demonstrated that the small mixed CP dataset was a harder challenge, because of both the complexity of the problems and the small amount of available CP problems to effectively utilize RAICL, highlighting the need for creating more such datasets.

Further advancements towards the development and evaluation of CP modelling assistants require more high-quality NL-CP datasets, including pairs of problem descriptions and their formal CP model, potentially inspired by educational material or examples from modelling systems. This could also allow going beyond in-context learning and towards supervised fine-tuning of LLMs. In addition, addressing larger (parameterized) problems that include external data files is an interesting challenge. Finally, as the generated models will likely have occasional errors, the integration with other (code-specialized) LLMs, debugging techniques, or example-based constraint acquisition techniques holds much promise.

References

1 Boris Almonacid. Towards an automatic optimisation model generator assisted with generative pre-trained transformer. *CoRR*, abs/2305.05811(arXiv:2305.05811), 2023. `doi:10.48550/arXiv.2305.05811`.

2 Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. `doi:10.48550/arXiv.1409.0473`.

3 Christian Bessiere, Clément Carbonnel, Anton Dries, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, Kostas Stergiou, Dimosthenis C. Tsouros, and Toby Walsh. Learning constraints through partial queries. *Artif. Intell.*, 319:103896, 2023. `doi:10.1016/j.artint.2023.103896`.

4 Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. URL: `https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html`.

5 Jaime G. Carbonell and Jade Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In W. Bruce Croft, Alistair Moffat, C. J. van Rijsbergen, Ross Wilkinson, and Justin Zobel, editors, *SIGIR '98: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 24-28 1998, Melbourne, Australia*, SIGIR '98, pages 335–336, New York, NY, USA, 1998. ACM. `doi:10.1145/290941.291025`.

6 Dhivya Chandrasekaran and Vijay Mago. Evolution of semantic similarity - A survey. *ACM Comput. Surv.*, 54(2):41:1–41:37, February 2022. `doi:10.1145/3440755`.

7 Harrison Chase. Langchain, October 2022. URL: `https://github.com/langchain-ai/langchain`.

8 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. `doi:10.48550/arXiv.2107.03374`.

9 Jens Claes, Bart Bogaerts, Rocsildes Canoy, Emilio Gamba, and Tias Guns. Zebratutor: Explaining how to solve logic grid puzzles. In Katrien Beuls, Bart Bogaerts, Gianluca Bontempi, Pierre Geurts, Nick Harley, Bertrand Lebichot, Tom Lenaerts, Gilles Louppe, and Paul Van Eecke, editors, *Proceedings of the 31st Benelux Conference on Artificial Intelligence (BNAIC 2019) and the 28th Belgian Dutch Conference on Machine Learning (Benelearn 2019), Brussels, Belgium, November 6-8, 2019*, volume 2491 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019. URL: `https://ceur-ws.org/Vol-2491/demo96.pdf`.

10 Cathleen Cortis Mack, Caterina Cinel, Nigel Davies, Michael Harding, and Geoff Ward. Serial position, output order, and list length effects for words presented on smartphones over very long intervals. *Journal of Memory and Language*, 97:61–80, 2017. `doi:10.1016/j.jml.2017.07.009`.

11 Parag Pravin Dakle, Serdar Kadioglu, Karthik Uppuluri, Regina Politi, Preethi Raghavan, SaiKrishna Rallabandi, and Ravisutha Srinivasamurthy. Ner4opt: Named entity recognition for optimization modelling from natural language. 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 299–319, Cham, 2023. Springer. `doi:10.1007/978-3-031-33271-5_20`.

12 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. `doi:10.18653/v1/n19-1423`.

13 Xuan-Dung Doan. VTCC-NLP at nl4opt competition subtask 1: An ensemble pre-trained language models for named entity recognition. *CoRR*, abs/2212.07219(arXiv:2212.07219), 2022. `doi:10.48550/arXiv.2212.07219`.

14 Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. A survey for in-context learning. *CoRR*, abs/2301.00234(arXiv:2301.00234), 2023. `doi:10.48550/arXiv.2301.00234`.

15 Eugene C. Freuder. Progress towards the holy grail. *Constraints An Int. J.*, 23(2):158–171, 2018. `doi:10.1007/s10601-017-9275-0`.

16 Eugene C. Freuder and Barry O'Sullivan. Grand challenges for constraint programming. *Constraints An Int. J.*, 19(2):150–162, 2014. `doi:10.1007/s10601-013-9155-1`.

17 Simon Frieder, Luca Pinchetti, Ryan-Rhys Griffiths, Tommaso Salvatori, Thomas Lukasiewicz, Philipp Christian Petersen, Alexis Chevalier, and Julius Berner. Mathematical capabilities of chatgpt. *CoRR*, abs/2301.13867, 2023. `doi:10.48550/arXiv.2301.13867`.

18 Neeraj Gangwar and Nickvash Kani. Highlighting named entities in input for auto-formulation of optimization problems. In Catherine Dubois and Manfred Kerber, editors, *Intelligent Computer Mathematics - 16th International Conference, CICM 2023, Cambridge, UK, September 5-8, 2023, Proceedings*, volume 14101 of *Lecture Notes in Computer Science*, pages 130–141. Springer, Springer, 2023. `doi:10.1007/978-3-031-42753-4_9`.

19 Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: program-aided language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR, PMLR, 2023. URL: `https://proceedings.mlr.press/v202/gao23f.html`.

20 Hila Gonen, Srini Iyer, Terra Blevins, Noah A. Smith, and Luke Zettlemoyer. Demystifying prompts in language models via perplexity estimation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 10136–10148, Singapore, December 2023. Association for Computational Linguistics. `doi:10.18653/v1/2023.findings-emnlp.679`.

21 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.

22 Jianglong He, Mamatha N, Shiv Vignesh, Deepak Kumar, and Akshay Uppal. Linear programming word problems formulation using ensemblecrf NER labeler and T5 text generator with data augmentations. *CoRR*, abs/2212.14657(arXiv:2212.14657), 2022. doi:10.48550/arXiv.2212.14657.

23 Adam Ishay, Zhun Yang, and Joohyung Lee. Leveraging large language models to generate answer set programs. In Pierre Marquis, Tran Cao Son, and Gabriele Kern-Isberner, editors, *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023*, KR '23, pages 374–383, 2023. doi:10.24963/kr.2023/37.

24 Elgun Jabrayilzade and Selma Tekir. Lgpsolver - solving logic grid puzzles automatically. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1118–1123. Association for Computational Linguistics, 2020. doi:10.18653/v1/2020.findings-emnlp.100.

25 Sanghwan Jang. Tag embedding and well-defined intermediate representation improve autoformulation of problem description. *CoRR*, abs/2212.03575(arXiv:2212.03575), 2022. doi:10.48550/arXiv.2212.03575.

26 Hyuhng Joon Kim, Hyunsoo Cho, Junyeob Kim, Taeuk Kim, Kang Min Yoo, and Sang-goo Lee. Self-generated in-context learning: Leveraging auto-regressive language models as a demonstration generator. *CoRR*, abs/2206.08082(arXiv:2206.08082), 2022. doi:10.48550/arXiv.2206.08082.

27 Zeynep Kiziltan, Marco Lippi, and Paolo Torroni. Constraint detection in natural language problem descriptions. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, IJCAI'16, pages 744–750. IJCAI/AAAI Press, 2016. URL: http://www.ijcai.org/Abstract/16/111.

28 Andrew K. Lampinen, Ishita Dasgupta, Stephanie C. Y. Chan, Kory W. Mathewson, Michael Henry Tessler, Antonia Creswell, James L. McClelland, Jane Wang, and Felix Hill. Can language models learn from explanations in context? In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 537–563, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi:10.18653/v1/2022.findings-emnlp.38.

29 Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 7871–7880. Association for Computational Linguistics, Association for Computational Linguistics, 2020. doi:10.18653/v1/2020.acl-main.703.

30 Junyi Li, Tianyi Tang, Wayne Xin Zhao, and Ji-Rong Wen. Pretrained language model for text generation: 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 4492–4499. ijcai.org, 2021. doi:10.24963/ijcai.2021/612.

31 Qingyang Li, Lele Zhang, and Vicky Mak-Hau. Synthesizing mixed-integer linear programming models from natural language descriptions. doi:10.48550/arXiv.2311.15271[math].

32 Xiaonan Li and Xipeng Qiu. Finding support examples for in-context learning. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 6219–6235, Singapore, December 2023. Association for Computational Linguistics. doi:10.18653/v1/2023.findings-emnlp.411.

33 Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? In Eneko Agirre, Marianna Apidianaki, and Ivan Vulic, editors, *Proceedings of Deep Learning Inside Out: The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures, DeeLIO@ACL 2022, Dublin, Ireland and Online, May 27, 2022*, pages 100–114, Dublin, Ireland and Online, May 2022. Association for Computational Linguistics. `doi:10.18653/v1/2022.deelio-1.10`.

34 Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Trans. Assoc. Comput. Linguistics*, 12:157–173, 2024. `doi:10.1162/tacl_a_00638`.

35 Michele Lombardi and Michela Milano. Boosting combinatorial problem modeling with machine learning. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 5472–5478. ijcai.org, July 2018. `doi:10.24963/ijcai.2018/772`.

36 Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 8086–8098, Dublin, Ireland, May 2022. Association for Computational Linguistics. `doi:10.18653/v1/2022.acl-long.556`.

37 Kostis Michailidis, Dimos Tsouros, and Tias Guns. CP-LLMs-ICL. Software, swhId: `swh:1:dir:5e4383ad6c4329796c9f21c51bbff4882dca8271` (visited on 2024-08-16). URL: `https://github.com/kostis-init/CP-LLMs-ICL`.

38 Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 11048–11064, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. `doi:10.18653/v1/2022.emnlp-main.759`.

39 Arindam Mitra and Chitta Baral. Learning to automatically solve logic grid puzzles. In Lluís Màrquez, Chris Callison-Burch, Jian Su, Daniele Pighin, and Yuval Marton, editors, *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1023–1033, Lisbon, Portugal, September 2015. The Association for Computational Linguistics. `doi:10.18653/v1/d15-1118`.

40 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. `doi:10.1007/978-3-540-74970-7_38`.

41 Yuting Ning, Jiayu Liu, Longhu Qin, Tong Xiao, Shangzi Xue, Zhenya Huang, Qi Liu, Enhong Chen, and Jinze Wu. A novel approach for auto-formulation of optimization problems. *CoRR*, abs/2302.04643(arXiv:2302.04643), 2023. `doi:10.48550/arXiv.2302.04643`.

42 OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023. `doi:10.48550/arXiv.2303.08774`.

43 Ganesh Prasath and Shirish Karande. Synthesis of mathematical programs from natural language specifications. *CoRR*, abs/2304.03287(arXiv:2304.03287), 2023. `doi:10.48550/arXiv.2304.03287`.

44 Steven Prestwich and Nic Wilson. A statistical approach to learning constraints. *International Journal of Approximate Reasoning*, page 109184, 2024. `doi:10.1016/j.ijar.2024.109184`.

45 Luc De Raedt, Andrea Passerini, and Stefano Teso. Learning constraints from examples. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial*

Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, pages 7965–7970. AAAI Press, 2018. doi:10.1609/aaai.v32i1.12217.

46 Rindranirina Ramamonjison, Haley Li, Timothy T. L. Yu, Shiqi He, Vishnu Rengan, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. Augmenting operations research with auto-formulation of optimization models from problem descriptions. *CoRR*, abs/2209.15565(arXiv:2209.15565), 2022. doi:10.48550/arXiv.2209.15565.

47 Rindranirina Ramamonjison, Timothy T. L. Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. Nl4opt competition: Formulating optimization problems based on their natural language descriptions. In Marco Ciccone, Gustavo Stolovitzky, and Jacob Albrecht, editors, *NeurIPS 2022 Competition Track, November 28 - December 9, 2022, Online*, volume 220 of *Proceedings of Machine Learning Research*, pages 189–203. PMLR, PMLR, 2021. URL: https://proceedings.mlr.press/v220/ramamonjison22a.html.

48 Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 31210–31227. PMLR, 2023. URL: https://proceedings.mlr.press/v202/shi23a.html.

49 Helmut Simonis. Building industrial applications with constraint programming. In Hubert Comon, Claude Marché, and Ralf Treinen, editors, *Constraints in Computational Logics: Theory and Applications, International Summer School, CCL'99 Gif-sur-Yvette, France, September 5-8, 1999, Revised Lectures*, volume 2002 of *Lecture Notes in Computer Science*, pages 271–309. Springer, 1999. doi:10.1007/3-540-45406-3_6.

50 Dimosthenis C. Tsouros, Senne Berden, and Tias Guns. Guided bottom-up interactive constraint acquisition. 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 36:1–36:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CP.2023.36.

51 Dimosthenis C. Tsouros, Senne Berden, and Tias Guns. Learning to learn in interactive constraint acquisition. 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 8154–8162. AAAI Press, 2024. doi:10.1609/aaai.v38i8.28655.

52 Dimosthenis C. Tsouros, Hélène Verhaeghe, Serdar Kadioglu, and Tias Guns. Holy grail 2.0: From natural language to constraint models. *CoRR*, abs/2308.01589(arXiv:2308.01589), 2023. doi:10.48550/arXiv.2308.01589.

53 Giuseppe Vallar and Costanza Papagno. Phonological short-term store and the nature of the recency effect: Evidence from neuropsychology. *Brain and Cognition*, 5(4):428–442, 1986.

54 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, volume 30, pages 5998–6008, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

55 Mark Wallace. Practical applications of constraint programming. *Constraints An Int. J.*, 1(1/2):139–168, 1996. doi:10.1007/BF00143881.

56 Kangxu Wang, Ze Chen, and Jiewen Zheng. Opd@nl4opt: An ensemble approach for the NER task of the optimization problem. *CoRR*, abs/2301.02459(arXiv:2301.02459), 2023. doi:10.48550/arXiv.2301.02459.

57 Xinyi Wang, Wanrong Zhu, Michael Saxon, Mark Steyvers, and William Yang Wang. Large language models are latent variable models: Explaining and finding good demonstrations for in-context learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, volume 36, pages 15614–15638. Curran Associates, Inc., 2023. URL: http://papers.nips.cc/paper_files/paper/2023/hash/3255a7554605a88800f4e120b3a929e1-Abstract-Conference.html.

58 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. URL: http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

59 Jerry W. Wei, Jason Wei, Yi Tay, Dustin Tran, Albert Webson, Yifeng Lu, Xinyun Chen, Hanxiao Liu, Da Huang, Denny Zhou, and Tengyu Ma. Larger language models do in-context learning differently. *CoRR*, abs/2303.03846, 2023. doi:10.48550/arXiv.2303.03846.

60 Zhiyong Wu, Yaoxiang Wang, Jiacheng Ye, and Lingpeng Kong. Self-adaptive in-context learning: An information compression perspective for in-context example selection and ordering. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 1423–1436, Toronto, Canada, July 2023. Association for Computational Linguistics. doi:10.18653/v1/2023.acl-long.79.

61 Jiacheng Ye, Zhiyong Wu, Jiangtao Feng, Tao Yu, and Lingpeng Kong. Compositional exemplars for in-context learning. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 39818–39833. PMLR, 23–29 July 2023. URL: https://proceedings.mlr.press/v202/ye23c.html.

62 Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Satlm: Satisfiability-aided language models using declarative prompting. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 45548–45580. Curran Associates, Inc., 2023. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/8e9c7d4a48bdac81a58f983a64aaf42b-Paper-Conference.pdf.

63 Xi Ye, Srinivasan Iyer, Asli Celikyilmaz, Veselin Stoyanov, Greg Durrett, and Ramakanth Pasunuru. Complementary explanations for effective in-context learning. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 4469–4484, Toronto, Canada, July 2023. Association for Computational Linguistics. doi:10.18653/v1/2023.findings-acl.273.

64 Ying-Jung Yvonne Yeh and Min-Hung Chen. Examining the primacy and recency effect on learning effectiveness with the application of interactive response systems (irs). *Technol. Knowl. Learn.*, 27(3):957–970, 2022. doi:10.1007/s10758-021-09521-6.

65 Yiming Zhang, Shi Feng, and Chenhao Tan. Active example selection for in-context learning. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 9134–9148, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi:10.18653/v1/2022.emnlp-main.622.

Table 4 Performance of various LLMs on LGPs using CP generation with R-MMR ($\lambda = 0.5$).

LLM	#Shots	#Err	Accuracy (%)		
			Model	Solution	Declaration
gpt-3.5-turbo-0125	4	6	71.00	74.00	87.98
gpt-4-0125-preview	4	16	61.00	64.00	73.85
deepseek-coder-33b-instruct	4	1	73.00	77.00	92.60
Mixtral-8x7B-Instruct-v0.1	4	19	52.00	52.00	70.01
Qwen1.5-72B-Chat	4	15	64.00	66.00	80.32
gpt-3.5-turbo-0125	8	4	74.00	77.00	90.49
gpt-4-0125-preview	8	10	77.00	82.00	85.87
deepseek-coder-33b-instruct	8	2	77.00	77.00	93.66
Mixtral-8x7B-Instruct-v0.1	8	18	58.00	61.00	74.90
Qwen1.5-72B-Chat	8	7	62.00	65.00	86.00

A Various LLMs

We also present preliminary results when running our methodology on LGPs for direct CP modelling using various LLMs in Table 4. We used the *OpenAI*[9], *Together AI*[10] and *DeepSeek*[11] APIs for these experiments. As a code-tuned LLM, the deepseek-coder-33b-instruct managed to produce the fewest models with errors. Additionally, gpt-4-0125-preview achieved the highest solution accuracy when prompted with 8 in-context examples.

B Model Equivalence Algorithm

To assert the equivalence of two constraint models, we first create a one-to-one mapping between their decision variables based on syntactic similarity. We then construct the conjunction of all constraints for each model. Assuming the two models are not equivalent, we add to a new combined model the negation of the equivalence formula between the two models, as defined in Equation (18). If the resulting model is unsatisfiable, the two models are considered equivalent. Otherwise, they are not. The detailed steps are provided in Algorithm 1.

C Errors in the generated CP model

C.1 0-shot

An example with a syntax error of a zero-shot attempt to directly model the first problem from the NL4Opt dataset with gpt-3.5-turbo-0125 is shown in Figure 5.

C.2 4-shot

An example with a syntax error on an LGP when prompting statically with 4 examples in the context is briefly shown in Figure 6.

[9] `https://platform.openai.com/docs/api-reference`
[10] `https://docs.together.ai/reference/chat-completions`
[11] `https://platform.deepseek.com/docs`

Algorithm 1 Model Equivalence.

```
function ModelsEquivalent($m1$, $m2$)
    $vars1 \leftarrow$ GetVariables($m1$), $vars2 \leftarrow$ GetVariables($m2$)
    if length($vars1$) $\neq$ length($vars2$) then
        return False
    end if
    $mapping \leftarrow$ CreateMapping($vars1$, $vars2$)    ▷ Map variables based on syntax matching of their names
    $conj1 \leftarrow \bigwedge_{c \in m1.constraints} c$
    $conj2 \leftarrow \bigwedge_{c \in m2.constraints} c$
    $negatedImplications \leftarrow mapping \wedge ((conj1 \wedge \neg conj2) \vee (\neg conj1 \wedge conj2))$
    if IsUnsatisfiable($negatedImplications$) then
        return True    ▷ Models are equivalent
    else
        return False    ▷ Models are not equivalent
    end if
end function
```

```python
import cpmpy.solvers

# Define the variables
num_sled_dog_trips = intvar(0, 10)  # Number of sled dog trips
num_truck_trips = intvar(0, 10)  # Number of truck trips

# Define the objective function to maximize the number of fish transported
total_fish_transported = 100*num_sled_dog_trips + 300*num_truck_trips

# Define the constraints
cost_sled_dog_trips = 50*num_sled_dog_trips  # Cost of sled dog trips
cost_truck_trips = 100*num_truck_trips  # Cost of truck trips
total_cost = cost_sled_dog_trips + cost_truck_trips <= 1000  # Total cost constraint

# Number of sled dog trips must be less than the number of truck trips
num_sled_dog_trips <= num_truck_trips

# Define the LP problem
problem = cpmpy.Model(
    cpmpy.maximize(total_fish_transported),
    [total_cost]
)

# Solve the LP problem
cpmpy.solvers.cplex(problem)

# Error: module 'cpmpy' has no attribute 'maximize'
```

Figure 5 Generated code with syntax error. Configuration: gpt-3.5-turbo-0125, 0-shot.

```
from cpmpy import *

# Code omitted

# Clue 4: Of the runner who paid $9 and the contestant who ordered fettuccine,
     one was Margie and the other was Velma:
m += ((margie == price9) & (velma == fettuccine)) | ((margie == fettuccine) &
    (velma == price9)

# Error: ')' expected
```

Figure 6 Generated code with syntax error. Configuration: gpt-3.5-turbo-0125, 4-shot static ICL.

Table 5 Mixed CP Dataset. Configuration: gpt-3.5-turbo-0125, 4-shot static in-context examples selection.

Pathway	Solution Accuracy (%)
Direct	16.67
CP	50.00
+ BM	50.00
+ NER	44.44

D Mixed CP Dataset: Supplementary Results

Adding to the main results presented in the paper, Table 5 shows results for all four pathways and Table 6 displays the solution-level accuracy of the direct CP modelling pathway for all seven example retrieval strategies.

E Datasets Metadata

In this section, we present details of the datasets that we utilized in the experimental part. Table 7 describes the NL4Opt dataset. All values refer to the average number across each instance in the dataset. Regarding the LGPs, they contain on average 12 entities (or decision variables) and 4.55 clues per instance, while each instance has one unique solution. Table 8 outlines the type of each clue along with an example and its constraint representation. Finally, Table 9 provides some information about the instances of the mixed CP dataset that we curated.

Table 6 Mixed CP Dataset. Configuration: gpt-3.5-turbo-0125, 4 examples in the context, direct CP modelling pathway.

Strategy	Solution Accuracy (%)
Static	50.00
Random	38.89
SIM	55.56
R-SIM	55.56
MMR	61.11
R-MMR	55.56
LSRR	50.00

Table 7 NL4Opt information. To calculate the average number of optimal solutions for the train split, we did not take into consideration some outlier instances that had over 100 optimal solutions. All constraints and objectives are linear in this dataset. For more details please refer to the original paper [46].

Split (#)	#Decision Var.	#Constraints	#Obj. Terms	#Optimal Solutions
Train (713)	2.09	2.79	2.05	1.13
Test (289)	2.02	2.92	1.43	1.20

Table 8 LGPs clue types. In our CP models, we treat all puzzle entities as decision variables. Also refer to the other works employing LGPs [39, 24].

Type	Clue Example	`Constraint Expression (CPMpy [21])`
Equivalence	The Luzagueil is a chardonnay	`luzagueil == chardonnay`

XOR	The Annata Branco is either the 1992 wine or the syrah.	`Xor([annata == vintage1992,`
		`annata == syrah])`

Pairwise XOR	Of the pinot gris and the 1984 bottle, one is the Luzagueil and the other is the Zifennwein	`Xor([(pinot_gris == luzagueil) &`
		`(vintage1984 == zifennwein),`
		`(pinot_gris == zifennwein) &`
		`(vintage1984 == luzagueil)])`

AllDifferent	The four people are Deep Shadow, the superhero who started in 2007, the hero who started in 2009 and Matt Minkle	`AllDifferent([deep_shadow, _2007,`
		`_2009, matt_minkle])`

Arithmetic Comparison	The pinot gris was bottled 4 years after the merlot	`[((v1 == pinot_gris) & (v2 == merlot))`
		`.implies(vintage_to_int[v1] ==`
		`vintage_to_int[v2] + 4)`
		`for v1 in vintages for v2 in vintages]`

Table 9 Mixed CP Dataset information.

Instance	#Dec. Vars	#Const.	Optimisation	#Vars per Constraint	#(Optimal) Solutions
Five Floors	5	7	No	2	1
Bank Card	4	3	No	4	1
Guards and Apples	6	6	No	2	1
Magic Square	16	11	No	5	10
Thick as Thieves	6	7	No	3	1
Money Change	6	1	Yes	6	1
Colour Simple	6	9	Yes	2	>50
Movie Scheduling	9	26	Yes	2	3
Subset Sum	6	1	No	6	1
Subsets 100	20	4	No	15	>50
Maximal Ind. Sets	8	24	Yes	2	2
Exodus	20	9	No	10	17
People in a Room	17	13	No	9	>50
Kidney Exchange	64	24	Yes	8	1
Farmer and Cows	25	10	No	25	>50
Grocery	4	2	No	4	1
Climbing Stairs	20	22	No	12	>50
Hardy 1729	4	5	No	4	>50

F An LGP example

In Figure 7 we showcase an example from the LGPs dataset [39].

The Luzagueil is a chardonnay. The Annata Branco is either the 1992 wine or the syrah. The Friambliss is a syrah. Of the pinot gris and the 1984 bottle, one is the Luzagueil and the other is the Zifennwein. The pinot gris was bottled 4 years after the merlot. Vintages: 1984, 1988, 1992, 1996. Wines: Annata Branco, Friambliss, Luzagueil, Zifennwein. Types: chardonnay, merlot, pinot gris, syrah.

Decision Variables:

- Names of the wines: AnnataBranco, Friambliss, Luzagueil, Zifennwein (all different, 1 to 4)
- Vintages of the wines: Vintage1984, Vintage1988, Vintage1992, Vintage1996 (all different, 1 to 4)
- Types of the wines: Chardonnay, Merlot, PinotGris, Syrah (all different, 1 to 4)

Constraints:

- The Luzagueil is a chardonnay: Luzagueil == Chardonnay.
- The Annata Branco is either the 1992 wine or the syrah: Xor([AnnataBranco == Vintage1992, AnnataBranco == Syrah]).
- The Friambliss is a syrah: Friambliss == Syrah.
- Of the pinot gris and the 1984 bottle, one is the Luzagueil and the other is the Zifennwein: Xor([(PinotGris == Luzagueil, Vintage1984 == Zifennwein), (PinotGris == Zifennwein, Vintage1984 == Luzagueil)]).
- The pinot gris was bottled 4 years after the merlot: v1 == PinotGris and v2 == Merlot => v1_to_int == v2_to_int + 4, for all v1, v2 in vintages.

```python
from cpmpy import *

# Decision variables
# (variables with the same value will correspond to the same triplet of vintage, wine, and type)
# e.g. if annata == 1 and vintage1984 == 1 and chardonnay == 1, then Annata Branco is a 1984 chardonnay
annata, friambliss, luzagueil, zifennwein = wines = intvar(1, 4, shape=4)
vintage1984, vintage1988, vintage1992, vintage1996 = vintages = intvar(1, 4, shape=4)
chardonnay, merlot, pinot_gris, syrah = types = intvar(1, 4, shape=4)
# Integer representation, for comparison constraints
vintage_to_int = {vintage1984: 1984, vintage1988: 1988, vintage1992: 1992, vintage1996: 1996}  # in years

# Helper functions (for formulating comparison constraints)
def bottled_exactly_after_than(var1, var2, diff):
    """
    Formulate the constraint that var1 was bottled exactly diff years after var2.
    """
    return [((v1 == var1) & (v2 == var2)).implies(vintage_to_int[v1] == vintage_to_int[v2] + diff) for v1 in vintages for v2 in vintages]

# Constraints
m = Model()
# All entities are different per category
m += AllDifferent(wines)
m += AllDifferent(vintages)
m += AllDifferent(types)
# Clue 1: The Luzagueil is a chardonnay:
m += luzagueil == chardonnay
# Clue 2: The Annata Branco is either the 1992 wine or the syrah:
m += Xor([annata == vintage1992,annata == syrah])
# Clue 3: The Friambliss is a syrah:
m += friambliss == syrah
# Clue 4: Of the pinot gris and the 1984 bottle, one is the Luzagueil and the other is the Zifennwein:
m += Xor([(pinot_gris == luzagueil) & (vintage1984 == zifennwein), (pinot_gris == zifennwein) & (vintage1984 == luzagueil)])
# Clue 5: The pinot gris was bottled 4 years after the merlot:
m += bottled_exactly_after_than(pinot_gris, merlot, 4)
```

Figure 7 From top to bottom: Problem description, Blueprint Model, CPMpy model.

Strengthening Relaxed Decision Diagrams for Maximum Independent Set Problem: Novel Variable Ordering and Merge Heuristics

Mohsen Nafar[1]
Bielefeld University, Germany

Michael Römer
Bielefeld University, Germany

Abstract

Finding high-quality bounds is key to devising efficient exact solution approaches for Discrete Optimization (DO) problems. To this end, Decision Diagrams (DDs) provide strong and generic bounding mechanisms. This paper focuses on so-called relaxed DDs which, by merging nodes, over-approximate the solution space of DO problems and provide dual bounds the quality of which hinges upon the ordering of the variables in the DD compilation and on the selection of the nodes to merge. Addressing the Maximum Independent Set Problem, we present a novel dynamic variable ordering strategy relying on induced subgraphs of the original graph, and a new tie-based merge heuristic. In a set of computational experiments, we show that our strategies yield much stronger bounds than the standard state-of-the-art approaches. Furthermore, implementing our heuristics in a DD-based branch-and-bound, we reduce the solution times by around 33 % on average and by more than 50 % on hard instances.

2012 ACM Subject Classification Theory of computation → Discrete optimization

Keywords and phrases Decision Diagram, Dynamic Programming, Maximum Independent Set Problem, Dual Bound

Digital Object Identifier 10.4230/LIPIcs.CP.2024.21

Funding This research was funded by the Return Programme of the Federal State of North Rhine Westphalia (NRW Rückkehrprogramm).

1 Introduction

As becomes clear from the recent survey [6], Decision Diagrams (DDs) form a versatile tool for discrete optimization (DO), as they allow a compact representation of the solution space of DO problems in the form of a layered graph and provide generic mechanisms to obtain primal and dual bounds. Specifically, given a Dynamic Programming (DP) formulation of a DO problem, one can create a so-called *exact DD* such that the set of all paths in the DD corresponds to the set of feasible solutions to the DO problem. While the size of such an exact DD grows exponentially in the number of decision variables, there are two types of approximate DDs for which the number of nodes only grows linearly as the width of each DD layer is not allowed to exceed a threshold: *Restricted DDs*, which are obtained by removing feasible nodes (which are associated with states defined by the DP formulation), provide an under-approximation of the solution space, and *relaxed DDs*, which are obtained by merging nodes associated with non-equivalent states, provide an over-approximation of the solution space. As proposed in [3], relaxed and restricted DDs can be used within an exact branch-and-bound algorithm entirely based on DDs; the authors show that this method

[1] Corresponding author

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 21; pp. 21:1–21:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

achieves an excellent performance on DO problems such as the Maximum Independent Set Problem (MISP), the Maximum Cut Problem and the 2-Satisfiability Problem. For an in-depth discussion of DD-based solution approaches illustrated using a wide variety of DO problems, we refer to the monograph [1]. For an efficient open source implementation of DD-based branch-and-bound algorithm in Rust, we refer to the solver DDO presented in [10]. An alternative to the classical DD-based branch-and-bound with an open source implementation in the Julia language is *Peel and Bound* [16, 17].

The performance of DD-based branch-and-bound algorithms is highly dependent on the quality of the bounds of the approximate DDs involved. These approximate DDs are typically compiled using the so-called top-down approach, in which the DD is constructed layer by layer. For a given DO problem and a given maximum DD width, the strength of the approximate DD bounds depends on two key heuristic decisions within the compilation process: (i) the *variable ordering*, that is, the order in which the variables are considered in the top-down compilation, and (ii) the *node selection*, that is, the selection of the nodes in a DD layer to be removed (for restricted DDs) or to be merged (for relaxed DDs) in case the maximum width of a layer is exceeded.

The first decision, that is, devising a good variable ordering, is relatively straightforward for certain problems such as the 0/1-Knapsack Problem. For other problems such as the MISP, however, finding a good variable ordering turns out to be more intricate and has been considered by various authors, see e.g. the review in [6]. In particular, it has been shown that for the MISP it is useful to determine the variable ordering *dynamically*, that is, to decide upon the next variable to consider based on information becoming available during the DD compilation, e.g. in order by minimizing the number of nodes appearing in the next layer, or by performing some lookahead steps, see e.g. [1] for a comparison of different generic strategies. The MISP was also considered in several papers proposing to use Machine Learning to support the dynamic ordering of variables: As an example, in [5] the authors use Deep Reinforcement Learning to determine the variable ordering. They show that for a given maximum width of each layer, the ML-supported approach can substantially improve the bounds compared to the standard variable ordering heuristics considered in the literature. In two follow-up works [15, 4], the authors show that despite the fact that the ML-based compilation of approximate DDs is slower than the standard approaches, this bound improvement leads to a significant overall speed-up of an exact DD-based branch-and-bound solver.

The second decision, that is, the node selection decision, was also investigated by many authors. Following the monograph [1], a generic and often highly efficient strategy is to sort the states according to some criterion (e.g. the length of the partial path ending at each node), to keep the “best” nodes until the maximum width is reached and to merge the remaining nodes with the last node in the list to form one large “tail node”. In [8], the authors propose a classification-based mechanism to predict the most promising node selection heuristic for each layer of a relaxed DD. The paper [9] proposes a tie-breaking strategy to deal with the problem of identical criterion values in sorting-based approaches. Other node selection approaches do not rely on sorting nodes (and creating a single large node) but aim at grouping nodes to merge according to some similarity measure. As an example, [12] proposes to use so-called collector nodes that aim at merging states that have the same value with respect to a labeling function. A similar approach was recently used in [7] who merge nodes based on partitioning the state space for a single machine scheduling problem with release times, deadlines setup times and rejection. Another technique for top-down compilation of relaxed DDs which is based on DD reduction is proposed in [13]. In that paper, the authors partition the nodes in a lookahead layer, which is then used to reduce the target layer, i.e. the layer whose width exceeds the given maximum width. Finally, the paper [14] proposes using clustering approaches to group the nodes according to the state attributes and to form a single merged node for each group.

Contribution. Dealing with relaxed DDs for the MISP, this paper proposes both a new dynamic variable ordering strategy and a new heuristic to select which nodes to merge. The variable ordering strategy exploits the problem structure of the MISP by relying on graph-theoretical properties that can be inferred from the states of the partially compiled DDs. The new merge heuristic aims at reducing the approximation error in sorting-based merge strategies resulting from merging the whole tail into a single large node by introducing an additional merged node from nodes around the maximum width border having the same value of the sorting criterion. In a set of computational experiments with randomly generated graph instances with 100 vertices and different densities, we observe that each of the proposed approaches independently provides significantly stronger DD relaxations for the MISP than DDs compiled with standard approaches. When combined, i.e. using our proposed variable ordering and merge heuristic at the same time, the bounds become much stronger. Furthermore, implementing these heuristics in a DD-based branch-and-bound algorithm, the solution time reduces by 33% on average compared to a branch-and-bound using standard variable ordering and merge strategies. The solution time reduction grows with the hardness of the instances; for the hardest instances, the solution time reduction amounts to more than 50%.

2 Exact and Approximate Decision Diagrams

A decision diagram $\mathcal{D} = (\mathcal{N}, \mathcal{A})$ is a layered directed acyclic graph with node set $\mathcal{N}$ and arc set $\mathcal{A}$. The paths in $\mathcal{D}$ represent solutions to a discrete optimization problem $\mathscr{P}$ with a maximization objective function f and an n-dimensional vector of decision variables $x_1, \cdots, x_n \in \{0, 1\}$. $\mathcal{N}$ is partitioned into $n + 1$ layers $\mathcal{N}_1, \ldots, \mathcal{N}_{n+1}$, where $\mathcal{N}_1 = \{\mathbf{r}\}$ and $\mathcal{N}_{n+1} = \{\mathbf{t}\}$ for a *root* $\mathbf{r}$ and a *terminal* $\mathbf{t}$. Each arc $a = (u, u')$ connects nodes of two consecutive layers $\ell(u), \ell(u') = \ell(u) + 1$ and is associated with a decision $d(a)$ representing the assignment $x_{\ell(u)} = d(a)$. This means that a path $p = (a_1, \ldots, a_n)$ starting from $\mathbf{r}$ and ending at $\mathbf{t}$ represents the solution $x(p) = (d(a_1), \ldots, d(a_n))$. We denote the set of all $\mathbf{r}$-$\mathbf{t}$ paths with $\mathcal{P}$, and we refer to the solutions to $\mathscr{P}$ represented by $\mathcal{P}$ with $\text{Sol}(\mathcal{D})$. Moreover, each arc a has length $w(a)$ and $\sum_{i=1}^{n} w(a_i)$ provides the length $w(p)$ of path p. We refer to $\mathcal{D}$ as exact if $\text{Sol}(\mathcal{D}) = \text{Sol}(\mathscr{P})$ if for each path $p \in \mathcal{P}$ we have $w(p) = f(x(p))$; then a longest path in $\mathcal{D}$ forms an optimal solution to $\mathscr{P}$.

To deal with the exponential growth of the DD size, DD-based solution approaches such as DD-based branch-and-bound [2] rely on so-called approximate DDs that can be used to obtain upper or lower bounds for the solutions of $\mathscr{P}$. There are two types of approximate DDs: in a *restricted* DD $\mathcal{D}$, one aims at considering only promising nodes and arcs, meaning that $\text{Sol}(\mathcal{D}) \subseteq \text{Sol}(\mathscr{P})$, and thus, the longest path in a restricted DD provides a lower bound to $\mathscr{P}$. The second type of approximate DD, which is the one we focus on in this paper, is the *relaxed* DD providing an upper bound: in a relaxed DD, we have $\text{Sol}(\mathcal{D}) \supseteq \text{Sol}(\mathscr{P})$, that is, the set of paths may contain paths associated with infeasible solutions to $\mathscr{P}$. Regarding the objective function value, every path in a relaxed DD needs to satisfy $w(p) \geq f(x(p))$. In both restricted and relaxed DDs, a common approach to control the size of the DD is to impose a maximum width W for each layer in the DD which is enforced by removing nodes (in a restricted DD) or merging nodes (in a relaxed DD).

A common approach to compile an exact DD is to provide a Dynamic Programming (DP) formulation of $\mathscr{P}$ and to compile the DD in a top-down fashion. To do so, every node u is associated with a state S_u and every arc a is associated with a state transition induced by the decision $d(a)$ associated with a. S_u is an element of the state space $\mathcal{S}$; the state $S_{\mathbf{r}}$ associated with the $\mathbf{r}$ is the so-called *initial state*. The state S_v of the target node v of the arc depends on the state S_u of the arc's source node as well as on d and is computed by the state-transition function $f(S_u, d)$. The objective function contribution of a decision are computed by a reward function $g(S_u, d)$. Finally, the set of out-arcs of a node u is determined by the set of feasible decisions $X(S_u)$ given state S_u. The top-down compilation then proceeds layer-by-layer until reaching layer $\mathcal{N}_n$; all arcs emanating from that layer point to the terminal node $\mathbf{t}$. In a DD compiled in the sketched top-down fashion, any pair of nodes in a layer has different states, that is, partial paths ending in the same state point to the same node.

In case of approximate DDs, after having created all nodes of a given layer, one reduces its size to W by removing or merging nodes. Nodes are merged by redirecting the incoming arcs of the nodes to be merged to a single merged node. In order to ensure that no feasible completions of any of the merged nodes is lost, one requires a problem-specific merge operator $\oplus$ for the states associated with the two nodes, see [11] for a discussion of the conditions a valid merge operator needs to satisfy.

Algorithm 1 displays the pseudocode for a top-down compilation procedure for DD construction. The procedure takes a DP formulation DP (comprising the definition of the state space $\mathcal{S}$ including the initial state $S_{\mathbf{r}}$, the functions X, f and g), a DD $\mathcal{D}$ containing only the root node and the maximum width W. Calling the algorithm with an unlimited width W will yield an exact DD and depending on the operation performed in line 11, it will result in a restricted or relaxed DD. In addition to these, the algorithm requires node selection (for restricting and/or relaxing the layers) and variable ordering heuristics (an order for considering the decision variables for layer by layer construction, since every layer corresponds to one decision variable). In order to allow for a dynamic variable ordering, Algorithm 1 introduces the set *unfixed* of variables that have not been considered so far in the compilation as well as the the generic procedure *NextVariable* which chooses the next variable according to a given variable ordering strategy. Note that in case of a static variable ordering strategy, *NextVariable* simply returns the next variable according to a pre-specified order.

Algorithm 1 Top-Down DD Compilation.

```
1:  CompileTopDown ($DP$, $\mathcal{D}$, $W$)
2:  unfixed = set of all decision variables
3:  for $k = 1$ to $n$ do
4:     $x_k$ = NextVariable ($\mathcal{N}_k$, unfixed)
5:     unfixed = unfixed/$\{x_k\}$
6:     for all $u \in \mathcal{N}_k$ do
7:        for all $d \in X(S_u)$ do
8:           $v$ = GetOrAddNode ($\mathcal{N}_k, f(S_u, d)$)
9:           AddArc (u,v,d)
10:    if $|\mathcal{N}_{k+1}| > W$ then
11:       RelaxLayer/RestrictLayer ($\mathcal{N}_{k+1}$)
12: return $\mathcal{D}$
```

As mentioned in the introduction, the strength of the bounds obtained with approximate DDs compiled using a top-down approach crucially depends on the strategies used to determine the variable ordering (determined in line 4 of Algorithm 1), and on the strategies for selecting the nodes to remove or merge (line 11 of Algorithm 1). In this paper, we devise new heuristics for both of these decisions for the compilation of relaxed DDs for the Maximum Independent Set Problem which will be discussed next.

3 Decision Diagrams for the Maximum Independent Set Problem

In this section, we briefly introduce the Maximum Independent Set Problem (MISP) and its DP formulation. We then illustrate how this DP formulation can be used to construct exact and relaxed DDs.

3.1 The Maximum Independent Set Problem

Given a graph $G = (V, E)$ with n vertices, where $V = \{v_1, v_2, \cdots, v_n\}$ is the set vertices and E is the set of edges, the Maximum Independent Set Problem (MISP) asks for the largest subset $I \subseteq V$ such that no two vertices in I are connected via an edge, i.e. $I = \{v \in V | (u, v) \notin E, \forall u \in I\}$.

Example. Fig. 1 shows an example that will serve for illustration purposes in the remainder of this paper. It shows a graph G with five vertices. As can be easily verified, there are multiple optimal solutions, each of which contains two vertices, e.g. $I = \{v_1, v_4\}$ or $I = \{v_3, v_5\}$ or $I = \{v_2, v_3\}$.

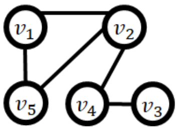


Figure 1 Example Graph G for a Maximum Independent Set Problem.

In order to be able to compile a decision diagram for MISP we need to formulate the MISP in terms of a DP. To begin with, a state S_u associated with a node u in the DD corresponds to a subset of the vertices V of the original graph G, namely with the vertices that are still available to be part of an independent set. The initial state S_r associated with the root node r thus corresponds to V, the terminal state corresponds to the empty set. Each layer j in the DD is associated with the decision variable x_j which consists in adding the j-th vertex $v(j)$ (according to the chosen variable order) in the original graph G to the independent set or not. Given a state S_u and a decision $d(a)$ ($d = 1$ means adding the vertex to the solution, $d = 0$ not adding it) associated with arc $a = (u, u')$ emanating from node u, the state transition function $f_j(S_u, d(a))$ determines the state of node u' in the next layer $j + 1$ of the DD. Specifically, $f_j(S_u, 0) = S_u / \{v(j)\}$, and $f_j(S_u, 1) = S_u / \{\Gamma(v(j))\}$ where $\Gamma(v(j))$ is the set of vertices adjacent to $v(j)$ in V. Note that if $v(j) \notin S_u$, the decision $d = 1$ is not feasible, and thus the DD will not contain an arc a emanating from u with $d(a) = 1$. The reward function $g_j(S_u, d)$ is $g_j(S_u, 0) = 0$ and $g_j(S_u, 1) = 1$.

Example (continued). Fig. 2 shows an exact DD for the MISP on graph G from Fig. 1 with the variable order according to the indexes of the vertices in V, that is, $v(j) = v_j \forall v \in V$. Every node of the DD is framed using blue color, where its corresponding state (i.e. a subset of the vertices in V) is placed inside its frame. Small orange labels next to each node are their partial objective value. Each dashed red and solid green arc show the assignment of value 0 and 1 to the corresponding decision variable, respectively. The exact DD in this example has a width of 4, and one of the longest r-t-paths with length 2 is $[x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 1, x_5 = 1]$, giving us an optimal solution, namely $\{v_4, v_5\}$ with the value 2.

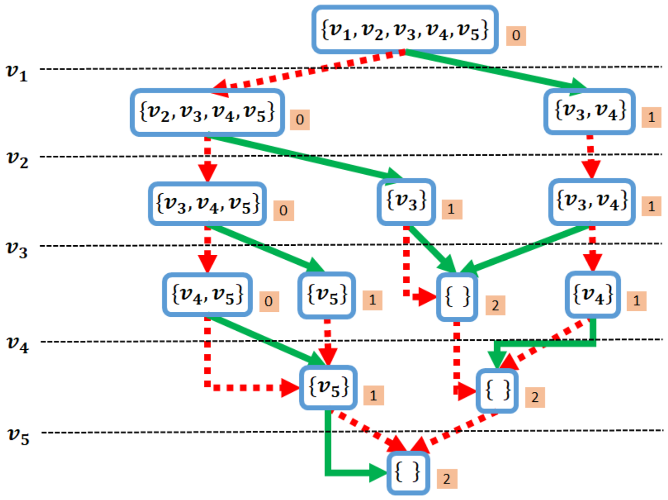


Figure 2 Exact DD for the MISP example graph G.

3.2 Relaxed Decision Diagrams for the MISP

As previously explained, a relaxed DD over-approximates the solution space of the problem under consideration, resulting in a dual bound. This relaxation is obtained by merging nodes with non-equivalent states. In order to obtain a valid relaxation, this merge process must ensure that no feasible solution is lost, that is, that the whole feasible solution space of the problem is included in the space represented by the relaxed DD. Merging two nodes u and u' into a node M involves two steps: First, all incoming arcs to nodes u and u' must be redirected to merged node M. Second, the state S_M of the new node must be determined in a way that the solution space of the tail problem starting from M contains the tail problem solutions of both nodes u and u'. As explained above, the state $S_M = S_u \oplus S_{u'}$, where $\oplus$ is a so-called merge operator. As discussed e.g. in [1], a valid merge operator for MISP is $\cup$, that is, S_M is given by the union of the vertex sets (subsets of V in the problem graph G) forming the states S_u and $S_{u'}$.

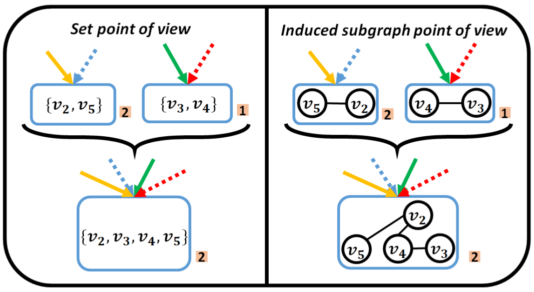


Figure 3 Merging two nodes in a relaxed DD for the MISP.

Fig. 3 illustrates the merge of two nodes into a single node. Observe that the state of the resulting merged node is the union of states of the two nodes on top. Moreover, recall that while a state in MISP is a subset of the vertices, it can be interpreted to represent an induced subgraph of the original graph on those vertices. Therefore, the edge between v_2 and v_4 which did not exist in any of the two induced subgraphs of the nodes to be merged now will be included in the induced subgraph of the state of the merged node. Actually, this augmented interpretation of the DD states for the MISP carries more information about the DD nodes – this fact will later be used to develop the new variable ordering heuristic proposed in this paper.

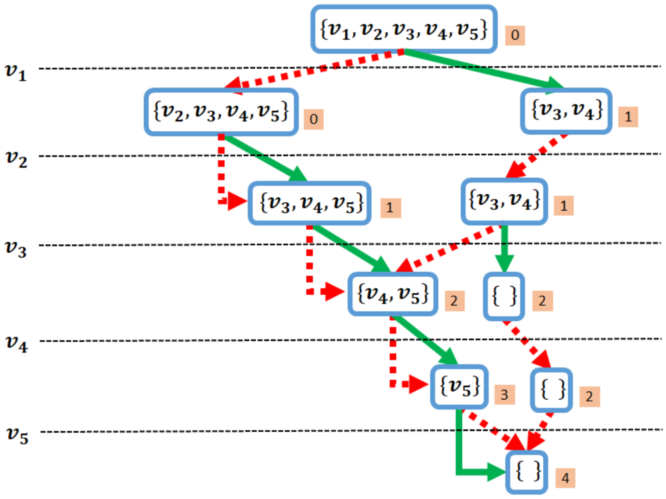


Figure 4 Relaxed DD for graph G, where variables are ordered according to their indices in the original graph and $W = 2$. Its obtained gap is 100%.

Example (continued). Fig. 4 represents the relaxed DD for graph G where $W = 2$, using the variable ordering given by indexes of the vertices, i.e. $v(1) = v_1, v(2) = v_2, v(3) = v_3, v(4) = v_4, v(5) = v_5$, in Fig. 1 that was also used for the exact DD. It turns out that the resulting dual bound has the value of 4 which is two times the optimum value.

4 A New Dynamic Variable Ordering for the MISP

The order of the decision variables according to which the DD is being compiled heavily affects both the size of an exact DD, and quality of the bounds from approximate DDs with a fixed maximum width. In the examples above, we used a static variable ordering, that is, an ordering that is specified before the compilation of the DD and that is independent of the configuration of the layers during the compilation process. However, it turns out that the best variable ordering strategies for the MISP are dynamic, that is, they use information of the partially compiled DD to choose the next decision variable (in case of MISP, the next vertex).

In the following, we will first consider the dynamic variable ordering strategy most commonly used for the MISP in the literature. Then, we propose a novel dynamic variable ordering strategy that exploits the information gained by interpreting the DD states in terms of induced subgraphs.

4.1 Standard Strategy: Minimum Number of States (MIN)

In this ordering, vertices are assigned a value that corresponds to the number of times they appear in the states of the nodes in the current layer $\mathcal{N}_k$. Then, the vertex exhibiting the minimum number appearances is selected as the next vertex (variable) to be considered in the top-down compilation of the DD. Algorithm 2 shows the corresponding heuristic which is called *Minimum Number of States (MIN)*. The worst-case time complexity to perform this selection is $O(W \cdot |V|)$ per layer. In this algorithm, $\mathcal{N}_k$ is the current layer and *unfixed* is the set of the vertices to which no decision has been assigned yet.

Algorithm 2 NextVariableMIN ($\mathcal{N}_k$, unfixed).

```
NS := a dictionary where keys are unfixed variables (associated with vertices of $G$) and
  values 0
for $x \in$ unfixed do
  for all $S \in \mathcal{N}_k$ do
    if $v(x) \in S$ then
      NS[$x$]+=1
return the unfixed variable with minimum NS
```

Example (continued). Using the MIN variable ordering in the top-down compilation of a relaxed DD with $W = 2$ for the example graph G from Fig. 1 results in the following variable order: $[v(1) = v_1,\ v(2) = v_2,\ v(3) = v_5,\ v(4) = v_3,\ v(5) = v_4]$. Following this order of variables (vertices), the corresponding relaxed decision diagram which provides a dual bound with value 3 is illustrated in Figure 5.

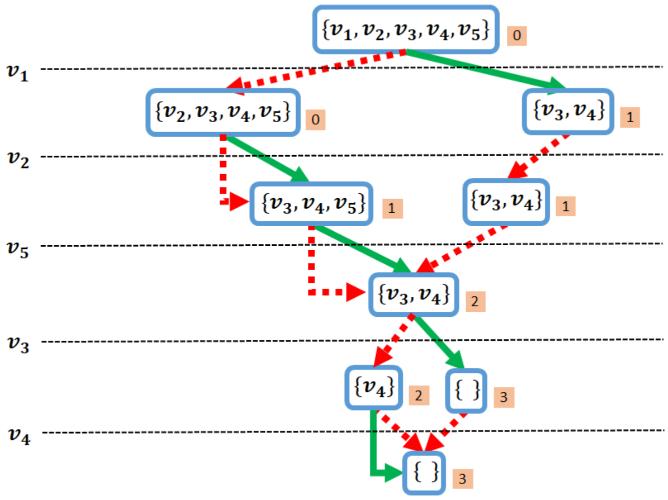


Figure 5 Relaxed DD for graph G, compiled via MIN and $W = 2$, yielding a gap of 50%.

The intuition behind the effectiveness of the MIN variable ordering strategy is that a node u only has an outgoing 1-arc for the variable associated with a vertex v if $v \in S_u$. As a result, a vertex appearing only a few times in the states of the layer under consideration will result in a small number of outgoing arcs of the layer and thus in a small number of nodes in the next layer.

4.2 A New Strategy: Current Degree Sum (CDS)

Now, we present a new dynamic variable ordering heuristic which is based on the interpretation of node states as induced subgraphs of the original graph introduced in Section 3. Previously, we mentioned that the intuition behind the MIN strategy is to improve the quality of the dual bounds in a relaxed DD by controlling the growth of the layers by choosing a variable that will result in less feasible decisions to be taken. Adding to this, the idea behind our new strategy is to reduce the "destructive" effect of the subsequent merge operations: We aim at choosing the variable order in a way that the difference between the states to be merged in a layer is somewhat small such that the resulting merged state is not too different from the states of the nodes to be merged.

Intuitively, a vertex with a lower degree in a given graph is likely to belong to a higher number of independent sets in that graph than a vertex with a larger degree. Therefore, it can be beneficial to decide about such vertices (i.e. vertices with smaller degree) sooner than later, because it can result in exploration of the search space that is closer to the optimum solution. Since the graph-theoretical information of vertices in a MISP evolves a lot during the compilation (every subproblem corresponding to a node/state is associated with an induced subgraph of the original graph on the members of that state), it is crucial to recompute them for each state and then take the best decision.

Current Degree Sum (CDS) is a novel variable ordering that can account for all of these intuitions, i.e. resulting in a lower number of feasible decisions, reducing the destructive effect of subsequent merge operations, and providing a better chance of resulting in partial solutions (tail solutions) closer to optimum. Recall that every state in a DP formulation of MISP is represented via a subset of the vertices of the original graph. Considering the induced subgraph on members of these states, every member (i.e. a vertex) has a degree which might be different from its degree in another state (induced subgraph), we call it the *current degree* of a vertex, i.e. dg_v^S reads degree of vertex v in state S. The strategy is to sum up the current degrees of each vertex, i.e. $\sum_{S\in\mathcal{N}_k} dg_v^S$ in the layer under consideration, and to choose the vertex with minimum sum.

Algorithm 3 shows the process for CDS variable ordering, which has a worst-case time complexity of $O(W \cdot |V|^2)$ per layer.

Algorithm 3 NextVariableCDS procedure ($\mathcal{N}_k$, unfixed).

```
CDS = a dictionary where keys are unfixed variables and values 0
for all $S \in \mathcal{N}_k$ do
    for $x \in S$ do
        for $y \in S$ do
            if $x$ is adjacent to $y$ then
                CDS[$x$]+ = 1
return the unfixed variable with minimum CDS
```

Example (continued). Using CDS variable ordering in the top-down compilation of a relaxed DD with $W = 2$ for the example graph G from Fig. 1 results in the following order of variables / vertices: $[v(1) = v_3,\ v(2) = v_4,\ v(3) = v_2,\ v(4) = v_1,\ v(5) = v_5]$. The corresponding relaxed decision diagram provides a dual bound with value 2 which is exactly the optimum value of the original problem (see Fig. 6).

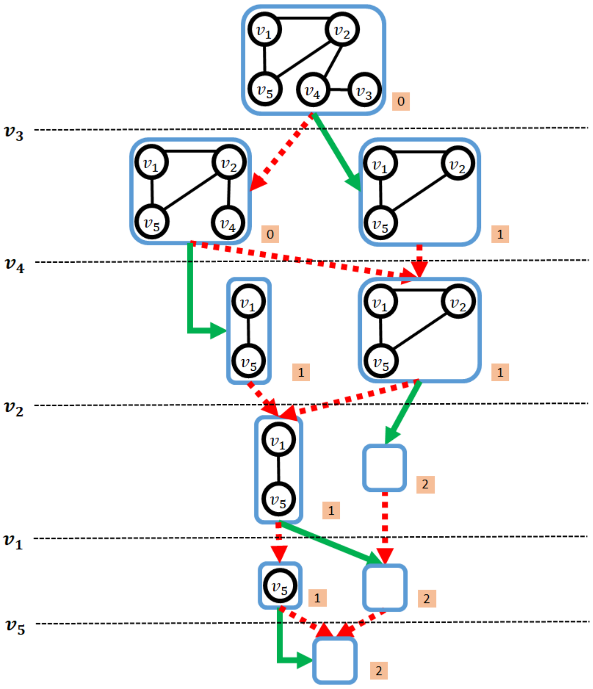


Figure 6 Relaxed DD for graph G compiled using **CDS** and $W = 2$, yielding a gap of 0%.

5 A New Merge Heuristic

The second important decision during the top-down compilation of relaxed DDs that has a huge impact on the quality of the achievable dual bounds is to select the nodes to merge in case the maximum width is exceeded. A (heuristic) strategy for taking this decision is also called a merge heuristic. In this section, we will first describe a problem-agnostic merge heuristic that is commonly used in the literature and then propose a new merge heuristic.

5.1 Standard Strategy: SortObj Merging (SO)

A highly generic merge heuristic for the top-down compilation of relaxed DDs that we will subsequently refer to as *SortObj (SO)* works as follows: First, all nodes in the layer the size of which exceeds the given maximum width W are sorted according to their objective function values. Then, the first $W - 1$ nodes, i.e. the nodes having the highest objective values, will remain in the layer, and the rest of the nodes, which we refer to the tail (of the sorted list) will be merged into a single node called M_{tail}.

Fig. 7 shows an example of SortObj being applied in a layer with 12 nodes, the maximum width is $W = 8$. In the figure, the nodes are sorted according to their objective values (written inside the orange boxes under the nodes). When applying SortObj, the first 7 nodes, i.e. $\{A, B, C, D, E, F, G\}$, will be retained in the layer and the rest of the nodes, i.e. $\{H, I, J, K, L\}$, will be merged and form the merged node M_{tail} that will replace all the nodes in the tail. The vertical dashed line in the figure marks the *border* between the nodes to be retained and those that are merged into M_{tail}.

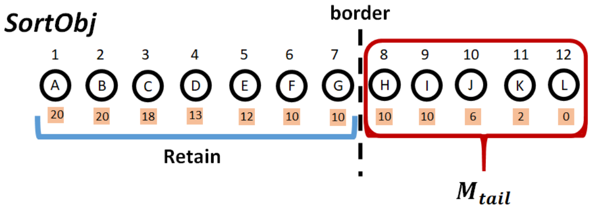


Figure 7 SortObj merging heuristic where $W = 8$.

5.2 A new Strategy: Border Tie Merging (BT)

Here, we present a new merge heuristic which is based on identifying certain ties that may arise in the layers of relaxed DDs. This merging heuristic that we call *Border Tie* merging and denote by **BT** is described after the following definition:

▶ **Definition 1** (Border Tie). *Let the nodes in a layer of a DD be sorted according to some criterion C and SortC be its corresponding sorted list, and let W be the given maximum width. A subsequence of SortC in which all nodes have the same criterion value and which includes SortC $[W-1]$ and SortC $[W]$ is called a Border Tie.*

Note that the smallest border tie includes at least 2 nodes, i.e. SortC $[W-1]$ and SortC $[W]$. Furthermore, there can exist at most one border tie in a layer. The criterion we use in border tie merging for the MISP is the same as in SortObj, that is, objective function value associated with each node.

A merged node in MISP is a “two-sided” over-approximation of all the nodes in the merge: From one side it over-approximates the states of the merged nodes by forming a super-set of their states. From the other side, redirecting the in-arcs causes an over-approximation of the longest path from root to the merged node. The intuition behind this new merge heuristic is to control the approximation error caused by the merge operation from one side, i.e. the over-approximation of the length of the partial solution caused by redirecting the in-arcs. Therefore, if we merge the nodes that have the same objective value (the lengths of their partial solutions are the same), then we can control the over-approximation error from one side. Moreover, we reckon that this merging heuristic will be a perfect fit for **CDS**, since one of the intuitions behind the design of **CDS** was to keep the diversity of the states in layers in a small range so as to control the destructive effect of the merge, that is the other side of the over approximation in MISP. Therefore, it is expected that coupling these two heuristics, i.e. **BT** and **CDS**, will strengthen the dual bounds obtained via relaxed DDs for MISP as they decrease the over-approximation error from both sides.

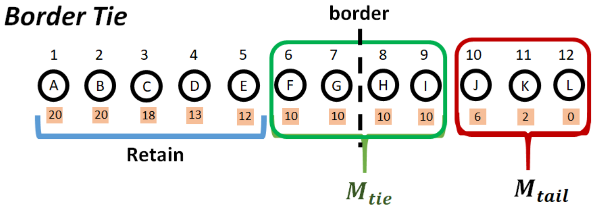


Figure 8 Border Tie merging heuristic where $W = 8$.

Fig. 8 shows the border tie merge heuristic applied to a layer with 12 nodes where $W = 8$. Note that the first two nodes in the list have the same objective value and therefore form a tie; however, their tie does not play a role in border tie merging heuristic. The **border tie** heuristic first identifies the nodes in the border tie (i.e., $\{F, G, H, I\}$) and merges them into a single merged node called M_{tie}. The nodes on the left side of the border tie are unchanged and remain in the layer. Finally, the nodes in the tail of the list, if there exist any, will be merged into M_{tail}. Note the following special cases regarding the tail of the sorted list may arise:

1. the tail is empty: in this case, M_{tie} is the only one merged node in the relaxed layer,
2. the tail comprises a single node: M_{tie} is the only one merged node in the relaxed layer, and the single tail node is unchanged and remains in the relaxed layer.

6 Computational Results

In this section, we present the results of computational experiments with two different dynamic variable ordering approaches (i.e. MIN, CDS) and two merge heuristics (i.e. SO, BT) and their combinations for the MISP. First, we assess the performance of the different strategy combinations with respect to the time-bound trade-off when compiling relaxed DDs. Second, we investigate the effects of these strategy choices for compiling relaxed DDs on the performance of an otherwise standard DD-based branch-and-bound algorithm [3]. For performing the experiments, we created nine instance sets, each of which contains 20 randomly generated graphs with 100 vertices. The instances set differ with respect to the graph density which ranges from 0.9 to 0.1. We implemented all approaches in the Julia programming language and ran the experiments on a Windows machine with 16GB RAM and an 11th Gen Intel(R) Core(TM) i7-11800H processor with 2.30 GHz.

6.1 Results on Dual Bounds

Fig. 9 shows the average dual gaps provided by using the strategy combinations (MIN,SO), (**CDS**, SO), (MIN,**BT**), and (**CDS**, **BT**) in relaxed DDs for the MISP on graph instances with different densities. The gaps reported in this figure are computed as $\frac{\text{dual bound}-\text{optimum}}{\text{optimum}} \times 100$ and then averaged over the 20 instances. Moreover, all relaxed DDs are compiled using given maximum widths $W \in \{50, 100, 200, 500, 1000\}$; the X-axis displays the time needed for compiling the relaxed DDs in ms. Every sub-plot in this figure corresponds to one density. In the sub-plots red-dashed, blue-solid, green-solid, and black-solid curves show (MIN,SO), (**CDS**, SO), (MIN,**BT**), and (**CDS**, **BT**) performances, respectively.

From the figure it is clear that all combinations involving our new heuristics, i.e. (**CDS**, SO), (MIN,**BT**), and (**CDS**, **BT**), provide stronger dual bounds than the standard strategy, i.e. (MIN,SO). Looking into the plots reveals that as the density decreases, the instances become harder and the performance difference between standard method and our proposed methods increases, meaning that for harder instances our strategies provide significantly stronger bounds. Moreover, (**CDS**, **BT**) which is the combination of our proposed dynamic variable ordering with our proposed merge heuristic clearly outperforms the other approaches.

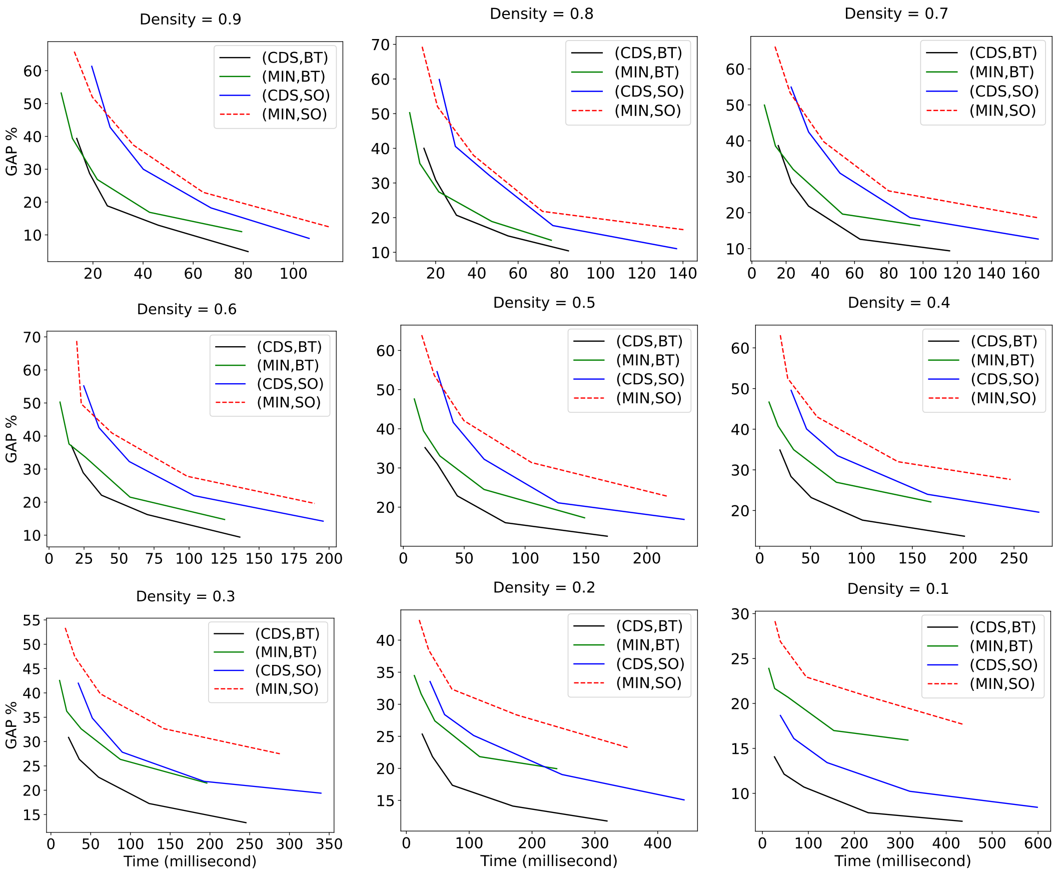


Figure 9 Gap-time performance of different strategies when being used in compiling relaxed DDs.

Table 1 shows the average size of the relaxed DDs built using different strategy combinations for a given maximum width $W = 1000$. A comparison between (MIN,SO) and (**CDS**,SO) supports one of the intuitions behind the design of CDS, i.e. reducing the destructive effect of merge operation by controlling the diversity of the states (subproblems / induced subgraphs). While MIN, which was designed to decrease the width of the layers, provides smaller relaxed DDs than those compiled using CDS, the relaxed DDs compiled via CDS are stronger, hinting at the successful control of the diversity of the states when using CDS. Another interesting observation is that when comparing the combinations that include **BT** to those that do not is the significant drop of the sizes of the DDs compiled having **BT** as their merge heuristic. It shows that although the DDs compiled using this heuristic contain more merged nodes (in some layers they can contain up to two merged nodes whereas in SortObj every layer has at most 1 merged node) and one might expect worse bound quality, it actually results in stronger dual bounds (see Fig. 9).

6.2 Performance within a DD-based Branch-and-Bound

We now present the results of implementing the proposed variable ordering and merge heuristics and their combinations in a DD-based branch-and-bound algorithm proposed in [3]. In the implemented branch-and-bound algorithm, the primal bounds are obtained using restricted DDs. For the compilation of the restricted DDs, no advanced approaches are used: The variables are ordered only once according to their degree in the original graph

Table 1 Average size of relaxed DDs compiled via different methods with $W = 1000$.

Density	**Average DD Size (nodes)**			
	(MIN,SO)	(**CDS**,SO)	(MIN,**BT**)	(**CDS**,**BT**)
0.1	73513	76190	**58059**	59069
0.2	71324	73190	**56393**	57173
0.3	68484	69470	**54020**	55253
0.4	65088	66580	**51870**	53049
0.5	62053	63355	**50456**	51356
0.6	58946	59661	**47715**	47505
0.7	55058	56040	**43606**	45629
0.8	50808	51466	**35334**	36025
0.9	45770	47272	**37578**	41607

in increasing order; the node selection follows the SortObj heuristic. We are dealing with a relatively pure implementation of a DD-based branch-and-bound that does not make use of advanced techniques but basically follows the description in the monograph [1]. All relaxed and restricted DDs in the branch-and-bound are compiled with a maximum width $W = 100$.

Table 2 Average solution time of DD-based branch-and-bound for strategy combinations.

Density	**Time (Seconds)**			
	(MIN,SO)	(**CDS**,SO)	(MIN,**BT**)	(**CDS**,**BT**)
0.1	872.3	821.9	445.8	**320.9**
0.2	126.4	94.0	85.5	**56.4**
0.3	31.0	26.8	24.3	**17.0**
0.4	10.1	9.3	8.6	**5.9**
0.5	4.3	3.5	4.0	**2.5**
0.6	1.92	1.70	1.57	**1.4**
0.7	1.11	1.01	0.93	**0.89**
0.8	0.73	0.70	0.81	**0.66**
0.9	0.60	**0.56**	0.65	0.61

Table 2 shows the average solution time of DD-based branch-and-bound using relaxed DDs constructed via different strategies (all instances are solved to optimality). The results reveal that all combinations involving our new strategies are able to reduce the solution time considerably in comparison to the baseline combination (MIN,SO).

Table 3 Solution time reduction in percent of different strategy combination in comparison to the baseline, i.e. (MIN,SO).

Density	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	Average
(**CDS**,SO)	5.7	25.61	13.42	8.4	18.3	11.4	9.0	4.1	**6.6**	11.43
(MIN,**BT**)	48.8	32.3	21.4	15.6	6.2	18.2	16.2	-10.9	-8.3	15.52
(**CDS**,**BT**)	**63.2**	**55.3**	**45.2**	**42.0**	**40.6**	**25.52**	**19.8**	**9.5**	-1.6	**33.30**

Table 3 summarizes the solution time reductions of each combination per density compared to the baseline. As becomes clear from the table, using our proposed heuristics individually, i.e. (**CDS**,SO) and (MIN,**BT**), reduces the solution time by 11% and 15% on average. However, if we use both heuristics at the same time, i.e. (**CDS**,**BT**), the solution reduction increases to 33% on average. In all cases, as the hardness of the instances increases (that is, the graph density decreases), the superiority of the proposed methods becomes more significant, such that the best combination, i.e. (**CDS**,**BT**), has a solution time reduction of more than 50% compared to the baseline for instances with density 0.2 and 0.1.

Table 4 Average number of sub-problems solved in the DD-based branch-and-bound for different strategy combinations.

Density	**Node Size (Sub-problem Solved in B&B)**			
	(MIN,SO)	(**CDS**,SO)	(MIN,**BT**)	(**CDS**,**BT**)
0.1	63880	39051	34183	**17150**
0.2	15454	6623	9088	**3816**
0.3	5906	3053	3830	**1700**
0.4	2609	1581	1814	**802**
0.5	1452	796	947	**450**
0.6	702	397	440	**266**
0.7	399	256	283	**195**
0.8	291	210	245	**173**
0.9	247	**174**	224	178

Another interesting aspect of a branch-and-bound algorithm is the number of subproblems that are solved until an optimal solution is reached. We report the average number of the subproblems solved in the DD-based branch-and-bound using different methods in Table 4.

Table 5 Reduction of the number of the solved subproblems in percent in DD-based branch-and-bound using different strategy combinations in comparison to the base line, i.e. (MIN,SO).

Density	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	Average
(**CDS**,SO)	38.8	57.1	48.3	39.4	45.2	43.3	35.7	27.7	29.6	40.6
(MIN,**BT**)	46.4	41.1	35.1	30.4	34.7	37.3	29.0	15.8	9.3	31.0
(**CDS**,**BT**)	**73.1**	**75.30**	**71.2**	**69.2**	**68.9**	**62.0**	**51.0**	**40.3**	**27.6**	**59.8**

Table 5 summarizes the reduction of the number of subproblems solved in DD-based branch and bound using different combinations of strategies. A comparison between the methods shows that although (MIN,**BT**) have better dual gaps and better solution times than (**CDS**,SO), it requires more subproblems to be solved. This can be a sign that in a DD-based branch-and-bound, having good quality bounds is not the only factor for having a good solution time: If it was the only factor, we should have seen a smaller number of solved subproblems for (MIN,**BT**) as it gives better bounds than (**CDS**,SO). This suggests that perhaps the reason for this can be one of the intuitions behind the design of CDS, which was to move in the direction of having solutions that potentially have more intersections with optimal solutions. However, when combining CDS and BT, the algorithm has the benefits of the both, i.e. good quality bounds and a reduction of the number of solved subproblems by 50% on average.

To put our results into perspective, let us briefly mention the results reported in [4], where the authors compare the impact of their RL-based dynamic variable ordering strategy to the MIN strategy within a standared DD-based branch-and-bound algorithm for solving randomly generated MISP instances with a density of 0.3 and between 200 and 300 vertices. It turns out that for instances that could be solved to optimality, the reduction in solution time of their best approach compared to MIN is around 10%, whereas our reductions amount to $13.42\%, 21.4\%$, and 45.2% for ($\mathbf{CDS}$,SO), (MIN,$\mathbf{BT}$), and ($\mathbf{CDS}$,$\mathbf{BT}$) combinations, respectively, for instances with density 0.3.

7 Conclusion

In this paper, we propose a novel dynamic variable ordering and a new merge heuristic for the top-down compilation of relaxed DDs for the MISP. The dynamic variable ordering strategy relies on the information obtainable from induced subgraphs of the original graph and the merge heuristic merges the nodes among which there is a tie regarding their partial objective value. Our computational experiments from applying the new strategies to a set of randomly generated graph instances containing 100 vertices with densities ranging from 0.9 to 0.1 (20 instances per density) show that our proposed strategies, individually, are capable of significantly strengthening the dual bounds compared to the standard strategy from the literature where this strengthening becomes more significant when our methods are combined. For the harder instances, i.e. lower densities, the performance gap is higher in our favor. In the end, the implementation of the resulting relaxed DDs into a DD-based branch-and-bound reduces the solution time by 33% on average and more than 50% on harder instances.

References

1 David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and John Hooker. *Decision Diagrams for Optimization*. Springer Publishing Company, Incorporated, 1st edition, 2016.

2 David Bergman, Andre A Cire, Willem-Jan van Hoeve, and John Hooker. Branch-and-bound based on decision diagrams. In *Decision Diagrams for Optimization*, pages 95–122. Springer, 2016.

3 David Bergman, Andre A Cire, Willem-Jan Van Hoeve, and John N Hooker. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016.

4 Quentin Cappart, David Bergman, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Augustin Parjadis. Improving variable orderings of approximate decision diagrams using reinforcement learning. *INFORMS Journal on Computing*, 34(5):2552–2570, 2022.

5 Quentin Cappart, Emmanuel Goutierre, David Bergman, and Louis-Martin Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1443–1451, 2019.

6 Margarita P Castro, Andre A Cire, and J Christopher Beck. Decision diagrams for discrete optimization: A survey of recent advances. *INFORMS Journal on Computing*, 34(4):2271–2295, 2022.

7 Mathijs de Weerdt, Robert Baart, and Lei He. Single-machine scheduling with release times, deadlines, setup times, and rejection. *European Journal of Operational Research*, 291(2):629–639, 2021.

8 Nikolaus Frohner and Günther R Raidl. Merging quality estimation for binary decision diagrams with binary classifiers. In *International Conference on Machine Learning, Optimization, and Data Science*, pages 445–457. Springer, 2019.

9 Nikolaus Frohner and Günther R Raidl. Towards improving merging heuristics for binary decision diagrams. In *Learning and Intelligent Optimization: 13th International Conference, LION 13, Chania, Crete, Greece, May 27–31, 2019, Revised Selected Papers 13*, pages 30–45. Springer, 2020.

10 Xavier Gillard, Pierre Schaus, and Vianney Coppé. Ddo, a generic and efficient framework for mdd-based optimization. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 5243–5245, 2021.

11 John N Hooker. Job sequencing bounds from decision diagrams. In *International Conference on Principles and Practice of Constraint Programming*, pages 565–578. Springer, 2017.

12 Matthias Horn, Johannes Maschler, Günther R Raidl, and Elina Rönnberg. A*-based construction of decision diagrams for a prize-collecting scheduling problem. *Computers & Operations Research*, 126:105125, 2021.

13 Mohsen Nafar and Michael Römer. Lookahead, merge and reduce for compiling relaxed decision diagrams for optimization. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 74–82. Springer, 2024.

14 Mohsen Nafar and Michael Römer. Using clustering to strengthen decision diagram bounds for discrete optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 8082–8089, 2024.

15 Augustin Parjadis, Quentin Cappart, Louis-Martin Rousseau, and David Bergman. Improving branch-and-bound using decision diagrams and reinforcement learning. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5–8, 2021, Proceedings 18*, pages 446–455. Springer, 2021.

16 Isaac Rudich, Quentin Cappart, and Louis-Martin Rousseau. Peel-and-bound: Generating stronger relaxed bounds with multivalued decision diagrams. 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 35:1–35:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.CP.2022.35`.

17 Isaac Rudich, Quentin Cappart, and Louis-Martin Rousseau. Improved peel-and-bound: Methods for generating dual bounds with multivalued decision diagrams. *Journal of Artificial Intelligence Research*, 77:1489–1538, 2023.

Learning Lagrangian Multipliers for the Travelling Salesman Problem

Augustin Parjadis

Polytechnique Montréal, Canada

Quentin Cappart

Polytechnique Montréal, Canada

Bistra Dilkina

Center for Artificial Intelligence in Society, University of Southern California, Los Angeles, CA, USA

Aaron Ferber

Center for Artificial Intelligence in Society, University of Southern California, Los Angeles, CA, USA

Louis-Martin Rousseau

Polytechnique Montréal, Canada

Abstract

Lagrangian relaxation is a versatile mathematical technique employed to relax constraints in an optimization problem, enabling the generation of dual bounds to prove the optimality of feasible solutions and the design of efficient propagators in constraint programming (such as the weighted circuit constraint). However, the conventional process of deriving Lagrangian multipliers (e.g., using subgradient methods) is often computationally intensive, limiting its practicality for large-scale or time-sensitive problems. To address this challenge, we propose an innovative unsupervised learning approach that harnesses the capabilities of graph neural networks to exploit the problem structure, aiming to generate accurate Lagrangian multipliers efficiently. We apply this technique to the well-known Held-Karp Lagrangian relaxation for the traveling salesman problem. The core idea is to predict accurate Lagrangian multipliers and to employ them as a warm start for generating Held-Karp relaxation bounds. These bounds are subsequently utilized to enhance the filtering process carried out by branch-and-bound algorithms. In contrast to much of the existing literature, which primarily focuses on finding feasible solutions, our approach operates on the dual side, demonstrating that learning can also accelerate the proof of optimality. We conduct experiments across various distributions of the metric traveling salesman problem, considering instances with up to 200 cities. The results illustrate that our approach can improve the filtering level of the weighted circuit global constraint, reduce the optimality gap by a factor two for unsolved instances up to a timeout, and reduce the execution time for solved instances by 10%.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Theory of computation → Constraint and logic programming; Computing methodologies → Machine learning

Keywords and phrases Lagrangian relaxation, unsupervised learning, graph neural network

Digital Object Identifier 10.4230/LIPIcs.CP.2024.22

Supplementary Material
Software (Code source): `https://github.com/corail-research/learning-hk-bound`

Acknowledgements We sincerely thank the anonymous reviewers for their constructive feedback. Their comments helped us better position our contribution within the field. Furthermore, their insights have provided guidance for our future research directions.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 22; pp. 22:1–22:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The *travelling salesman problem* (TSP) has been the subject of extensive research and has broad practical applications. Due to its NP-hard nature, numerous approaches have been proposed to solve it efficiently, ranging from exact to heuristic methods. Exact solvers not only need to identify the optimal solution but also to prove that it is optimal, often via a dual bound. Held and Karp (1970) [25] proposed a relaxation that provides strong dual bounds in practice. For instance, these bounds are used in Concorde, the state-of-the-art TSP solver [3] or in the design of global constraints in constraint programming [6, 5]. An associated branch-and-bound algorithm using this relaxation was subsequently proposed by Held and Karp (1971) [26], which enabled the optimality proof for several open benchmark instances at the time of its publication. Briefly, this algorithm leverages a combinatorial structure, referred to as *minimum 1-tree*, that can serve as a valid relaxation for the TSP and obtain dual bounds. However, this algorithm is based on a few heuristic design choices which have an important impact on the tightness of the relaxation. One is the procedure to generate the bounds from Lagrangian multipliers (explained in the next section), which can be assimilated as a hill-climbing algorithm. Starting from initial bounds, the algorithm refines the bound iteratively with local perturbations until convergence. There are two drawbacks to this process. First, it requires several potentially costly iterations to get accurate bounds, and second, it only converges to local minima. Our research hypothesis is that this procedure can be improved thanks to a learning-based approach. The idea is to train a model in an unsupervised fashion with similar TSP instances and to use it to predict Lagrangian multipliers that can be used to obtain a valid dual bound instead of computing it iteratively. In the field of constraint programming, Lagrangian decomposition has been also considered to provide dual bounds [8, 23], but without resorting to a learning-based component. We also note that other algorithms have been considered to improve the bounds for arborescence problems in constraint programming [28].

Machine learning has helped guide heuristic components in branch-and-bound [31, 38, 22, 57], constraint programming [13], SAT solving [49], local search [56], and non linear optimization [20]. We refer to the survey of Bengio et al. (2021) [7] for an extended literature review on this topic. Most of such works operate on the branching decisions (e.g., selecting the next variable to branch on) or on the primal side. However, learning to improve the quality of relaxations by means of better dual bounds has been much less considered in the literature. To our knowledge, this has only been addressed for the restricted use case of solvers based on decision diagrams [11], for combinatorial optimization over factor graphs [15] (e.g., see the max-sum labeling problem [54] and soft arc consistency [14] for relevant applications) and for learning relaxations of integer linear programs [1]. This last work is contemporaneous to ours.

Additionally, recent work in *decision-focused learning* (DFL) [40] has approached settings where the problem formulation is not fully specified at the time of decision-making. Thus, these approaches train gradient-based deep learning models to predict the missing components, with a key component being to determine how to train the deep learning model to improve the downstream decision quality. As training for deep networks is done using gradient descent, the difficulty lies in deriving methods for differentiating the output of the optimization model with respect to its predicted inputs. Our proposed approach seeks to predict the parameters of the Held-Karp relaxation such that the resulting relaxed solution provides a dual bound as tight as possible. This is achieved by deriving gradients for the relaxation to learn parameters that directly optimize the related bound. Differentiation has been successfully deployed for quadratic programs [2], linear programs [55, 18, 39], mixed integer linear programs [19], MAXSAT [52], and blackbox discrete optimization [44, 42], among others discussed in these surveys [46, 35]. However, this approach is the first to consider using differentiable optimization together with learning to improve the filtering of a global constraint.

Coming back to the TSP, the design of learning-based solving approaches has also sparked a great interest in the research community [16, 34, 37]. In an industrial context, this methodology is relevant for practitioners who are solving similar problem instances every day and want to leverage historical decisions, e.g. in last-mile package delivery [41]. *Graph neural networks* (GNN) is a neural architecture [47, 33] widely considered for the TSP [29]. More generally, GNNs also play a crucial role in the success of applying deep learning to combinatorial optimization [30, 12]. They allow for the extraction of rich hidden representations by successively aggregating the weights of neighboring nodes in a graph, on which many combinatorial problems are defined.

Based on this context, the contribution of the paper is an approach based on unsupervised learning and graph neural networks to generate appropriate Lagrangian multipliers for the TSP, which are then used to improve the Held-Karp relaxation. We highlight that, compared to most of the related work, we do not learn a primal heuristic but a learning-based strategy that derives valid and tight dual bounds. Additionally, we integrate this mechanism inside a branch-and-bound algorithm with domain filtering and constraint propagation [5] to improve exact TSP solving. Experiments are carried out on three distributions of metric TSPs and the results show that our approach can improve the filtering level of the weighted circuit global constraint, reduce the optimality gap by a factor of two for unsolved instances up to a timeout, and reduce the execution time for solved instances by 10%.

The following section briefly overviews the Held-Karp relaxation principle for the TSP. Building upon this, we next describe the proposed learning approach for generating bounds through unsupervised learning on the Lagrangian multipliers of the Held-Karp relaxation. Finally, we discuss the training and integration of dual-bound generation within a branch-and-bound algorithm to evaluate their impact.

2 Held-Karp Lagrangian Relaxation

Finding optimal solutions for large TSP instances requires sophisticated approaches due to the combinatorial explosion of the solution space. With branch-and-bound, optimization bounds are employed to prune the search tree and accelerate the search, allowing solvers to prove optimality without exploring the entire tree. To achieve this, the *Held-Karp relaxation* [25] offers a robust dual bound based on a variant of minimum spanning trees.

Let $G = (V, E)$ be a complete graph with a cost attached to each edge. A *minimum 1-tree* is a minimum spanning tree of $G\backslash\{1\}$ to which we add the node 1 along with the two cheapest edges connecting it to the tree. We note that the choice of node 1 is arbitrary, depending on the labeling of V. A minimum 1-tree can be obtained by solving the integer program presented in Equations (1) to (5). Constraints (2) and (3) define the 1-tree structure and Constraint (4) enforces the elimination of sub-tours. This problem involves finding a minimum spanning tree that can be solved in $\mathcal{O}(E \log V)$ by Kruskal's algorithm. Here, $\delta(v)$ denotes the edges containing node $v \in V$ and μ_e, ν_e denotes the two nodes linked by an edge $e \in E$. We use $c_e \in \mathbb{R}$ to represent the cost of an edge $e \in E$, and $x_e \in \{0, 1\}$ is the decision variable indicating whether edge e is included in the 1-tree.

$$\min \sum_{e \in E} c_e x_e \tag{1}$$

$$\text{s. t.} \sum_{e \in \delta(1)} x_e = 2 \tag{2}$$

$$\sum_{e \in E} x_e = |V| \tag{3}$$

$$\sum_{\substack{\mu_e \in S \\ \nu_e \in S \\ \mu_e < \nu_e}} x_e \leq |S| - 1 \qquad \forall S \subset V \backslash \{1\} \wedge |S| \geq 3 \tag{4}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E \tag{5}$$

Let us note that every tour in G is a 1-tree, and if a minimum 1-tree is a tour, it is an optimal solution to the TSP. Therefore any minimum 1-tree is a valid relaxation for the TSP, which is an interesting property to leverage. However, a solution of this integer program is not ensured to be a tour. To do so, a new set of constraints must be enforced.

$$\sum_{e \in \delta(v)} x_e = 2 \qquad \forall v \in V \backslash \{1\} \tag{6}$$

These constraints force each node to have only two edges, an incoming and an outgoing one, and turn the problem in finding a minimum-cost Hamiltonian cycle, which is NP-hard. To obtain a valid 1-tree relaxation efficiently, one can then move these constraints (one for each node) into the Objective (1) and penalize their violations with associated Lagrangian multipliers $\theta_v \in \mathbb{R}$ for each $v \in V \backslash \{1\}$. The updated objective function is as follows.

$$\min \sum_{e \in E} c_e x_e - \sum_{v \in V \backslash \{1\}} \theta_v \left(2 - \sum_{e \in \delta(v)} x_e \right) \tag{7}$$

Intuitively, each node having a degree other than two will be penalized. An optimal 1-tree relaxation can be found by optimizing over the θ_v variables. To do so, an iterative approach has been proposed by Held and Karp [25, 26]. The idea is to adjust the Lagrangian multipliers θ step-by-step to build a sequence of 1-trees which provides increasingly better bounds. An overview of the process is proposed in Figure 1.

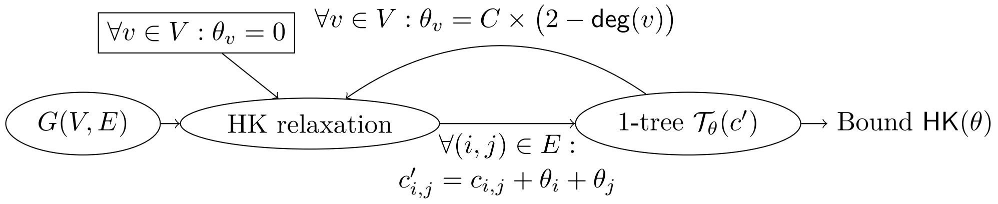

$\forall v \in V : \theta_v = 0$
$\forall v \in V : \theta_v = C \times \big(2 - \mathsf{deg}(v)\big)$
$G(V, E)$
HK relaxation
1-tree $\mathcal{T}_\theta(c')$
Bound $\mathsf{HK}(\theta)$
$\forall (i, j) \in E :$
$c'_{i,j} = c_{i,j} + \theta_i + \theta_j$

Figure 1 Approach of Held and Karp [25, 26] - Iterative process for improving θ multipliers.

First, an initial minimum 1-tree is computed by finding a minimum spanning tree on $G \backslash \{1\}$ and adding the two cheapest edges incident to node 1. If the optimal 1-tree is a tour, it corresponds to the optimal TSP solution. Otherwise, some constraints are penalized as at least one node has a degree greater than 2. The main idea of Held and Karp [25, 26] is to penalize such nodes by modifying the cost $c_{i,j}$ of edges $(i, j) \in E$, based on the values of θ_i and θ_j (i.e., the multipliers of adjacent nodes). Let $c'_{i,j} \in \mathbb{R}$ be the modified costs. They are computed as follows.

$$c'_{i,j} = c_{i,j} + \theta_i + \theta_j \qquad \forall (i, j) \in E \tag{8}$$

A property proved by Held and Karp [25, 26] is that the optimal TSP tour is invariant under this perturbation, whereas the optimal 1-tree is not. This gives room to improve the solution by finding better multipliers. Equation (9) proposes a standard choice to compute the multipliers, where $C \in \mathbb{R}$ is an arbitrary constant and $\mathsf{deg}(v)$ denotes the degree of node $v \in V$ in the current 1-tree.

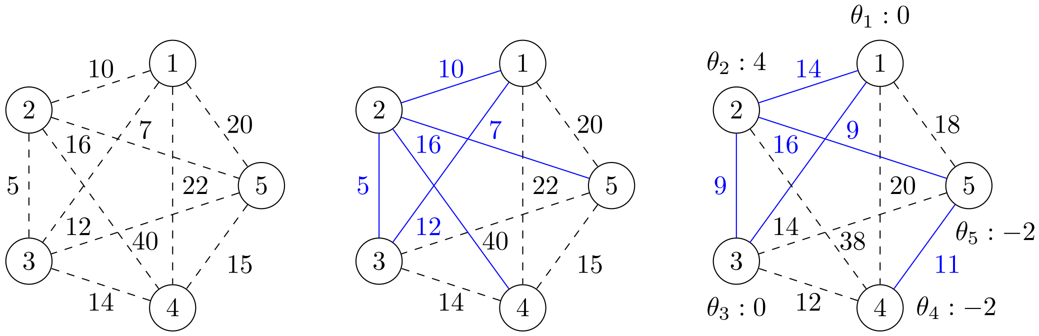
$$\theta_v = C \times \big(2 - \mathsf{deg}(v)\big) \qquad \forall v \in V \tag{9}$$

Finally, a new minimum 1-tree is computed from the graph with the updated costs $c'_{i,j}$. We note this 1-tree as $\mathcal{T}_\theta(c')$ where $c' = \{c_1, \dots, c_{|E|}\}$ is the set of all modified costs, and $\theta = \{\theta_1, \dots, \theta_{|V|}\}$ is the set of all multipliers. We also use the notation $\mathsf{cost}\big(\mathcal{T}_\theta(c')\big)$ to refer to the total cost of the 1-tree. This process is reiterated, and a new 1-tree $\mathcal{T}_\theta(c')$ is obtained until no improvement is obtained (i.e., when a local minimum is reached). The cost of the optimal 1-tree gives a lower bound on the objective value as follows.

$$\mathsf{HK}(\theta) = \mathsf{cost}\big(\mathcal{T}_\theta(c')\big) - 2\sum_{i=1}^{|V|} \theta_i \tag{10}$$

This bound, $\mathsf{HK}(\theta)$, is commonly referred to in the literature as the *Held-Karp bound.* This is a valid lower bound as if the related solution is not a tour, the optimal TSP tour will have a higher value. Otherwise, the optimal tour would have been obtained as tours are 1-trees. This approach is typically incorporated into a branch-and-bound algorithm, using this bound to prune the search. While computing a 1-tree is generally computationally efficient, the iterative adjustment of the θ multipliers can be computationally expensive. Our contribution is dedicated to mitigating this issue thanks to an unsupervised learning process.

Example. Figure 2 illustrates the Held-Karp relaxation for a graph with an optimal TSP tour value of 62 (a). A 1-tree is computed on the original graph without Lagrangian multipliers, which yields a bound of 50 (b). Considering Equation (9) with $C = 2$, we obtain the following multipliers: $\{\theta_1 : 0, \theta_2 : 4, \theta_3 : 0, \theta_4 : -2, \theta_5 : -2\}$. The corresponding penalized 1-tree with Lagrangian multipliers modifying the edge costs provides a bound of 59, which is tighter (c).



(a) Initial TSP instance. The optimal cost is 62.

(b) Minimum 1-tree under the initial costs. Lower bound obtained is 50 (gap of 20%).

(c) Minimum 1-tree with modified costs. Lower bound obtained is 59 (gap of 5%).

Figure 2 Illustration of a single iteration of Held-Karp relaxation for an arbitrary TSP instance.

In constraint programming, the *weighted circuit* constraint [4] ensures that a set of variables Y forms a Hamiltonian circuit on a graph, while also satisfying a specified condition on the total cost z of the circuit. This can be intuitively understood as a conjunction of a *circuit* constraint [36] and a second constraint enforcing the circuit to be lower than a threshold. Briefly, the standard filtering algorithms for this constraint typically involve: (1) identifying edges that must be included in any feasible solution, (2) eliminating edges that cannot be part of any solution, and (3) determining the minimum possible value of the cost threshold z. This information is then used for narrowing the domains of the variables Y. Given that establishing bounds consistency for this constraint is NP-hard, Benchimol et al. (2012) [5] proposed a filtering algorithm that uses relaxations of the weighted circuit constraint, specifically leveraging the Held-Karp relaxation. A stronger relaxation, such as a 1-tree that provides a tighter lower bound, allows for more extensive filtering. This in turn improves the efficacy of the weighted circuit propagator, which is the focus of this paper. More broadly, this propagator is part of the *cost-based filtering* family, which utilizes valid bounds for effective filtering. Generally, a tighter bound leads to more efficient filtering within this framework.

3 Learning Held-Karp Multipliers

The Held-Karp bound $\mathsf{HK}(\theta)$ has two interesting properties: (1) it can be parameterized thanks to the θ Lagrangian multipliers, and (2) it is always valid, meaning it will never exceed optimal TSP cost. Both properties open the opportunity to use a learning-based approach to compute the bound. To do so, we propose to build a model $\Phi_w : G(V, E) \rightarrow \mathbb{R}^{|V|}$ able to directly predict all the θ multipliers for a TSP instance given as input (i.e., a graph). The model is parameterized with p parameters $w = \{w_1, \dots, w_p\}$. There are two benefits to this. First, it eliminates parts of the iterative process of Held and Karp [25] depicted in Figure 2 and saves execution time. Second, it allows us to obtain tighter bounds. The process is illustrated in Figure 3.

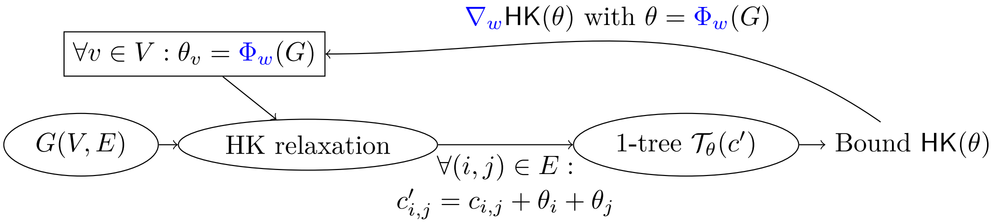

$\nabla_w \mathsf{HK}(\theta)$ with $\theta = \Phi_w(G)$
$\forall v \in V : \theta_v = \Phi_w(G)$
$G(V, E)$
HK relaxation
1-tree $\mathcal{T}_\theta(c')$
Bound $\mathsf{HK}(\theta)$
$\forall (i, j) \in E :$
$c'_{i,j} = c_{i,j} + \theta_i + \theta_j$

Figure 3 Our contribution - Unsupervised learning approach to obtain θ multipliers through backpropagation.

The goal is to find model parameters w yielding the highest possible bound. This corresponds to a maximization problem that can be solved by gradient-based optimization. The obtained bound is provably valid, regardless of the trained model's accuracy thanks to the second property. We consider this a major strength of our contribution, as obtaining guarantees with machine learning for combinatorial optimization is known to be a challenge [35]. We formulate the bound maximization problem and its gradient below.

$$\max_w \mathsf{HK}\big(\Phi_w(G)\big) \longmapsto \nabla_w \mathsf{HK}(\Phi_w(G)) \tag{11}$$

However, computing the gradient of this expression is not trivial, as the bound is obtained by means of the 1-tree combinatorial structure $\mathcal{T}_\theta(c')$ (see Equation (10)). As the tree is parameterized by θ, the chain rule can be applied to clarify the dependencies between model parameters w and Lagrangian multipliers θ.

$$\nabla_w \mathsf{HK}\big(\Phi_w(G)\big) = \frac{\partial \mathsf{HK}(\Phi_w(G))}{\partial \theta} \times \frac{\partial \theta}{\partial w} \tag{12}$$

The right term corresponds to the differentiation of the predictive neural network model and is easily obtained by backpropagation. However, the left term requires to differentiate the expression depicted in Equation (10) for all θ_i with $i \in V$.

$$\frac{\partial \mathsf{HK}(.)}{\partial \theta} = \frac{\partial \mathsf{cost}\big(\mathcal{T}_\theta(c')\big)}{\partial \theta} - 2\frac{\partial \sum_{i=1}^{|V|} \theta_i}{\partial \theta} \tag{13}$$

The cost of the 1-tree (i.e., $\mathsf{cost}\big(\mathcal{T}_\theta(c')\big)$ corresponds to the weighted sum of the selected edges (i.e., variables $x_{i,j}$ for each $(i,j) \in E$). The cost $c'_{i,j}$ defines the weights.

$$\frac{\partial \mathsf{HK}(.)}{\partial \theta} = \frac{\partial \Big(\sum_{(i,j)}^{\in E} c'_{i,j} x_{i,j} \Big)}{\partial \theta} - 2\frac{\partial \sum_{i=1}^{|V|} \theta_i}{\partial \theta} \tag{14}$$

Let us consider a specific multiplier θ_i associated to node $i \in V$ and let us unroll the cost as $c'_{i,j} = c_{i,j} + \theta_i + \theta_j$ (see Equation (8)). The partial derivative of θ_i is non-zero only for the node itself and its adjacent edges, i.e. $(i,j) \in \delta(i)$.

$$\frac{\partial \mathsf{HK}(.)}{\partial \theta_i} = \frac{\partial \sum_{(i,j)}^{\in \delta(i)} (c_{i,j} + \theta_i + \theta_j) x_{i,j}}{\partial \theta_i} - 2\frac{\partial \theta_i}{\partial \theta_i} \tag{15}$$

$$= \frac{\partial \sum_{(i,j)}^{\in \delta(i)} \theta_i x_{i,j}}{\partial \theta_i} - 2\frac{\partial \theta_i}{\partial \theta_i} \tag{16}$$

$$= \sum_{(i,j) \in \delta(i)} x_{i,j} - 2 \tag{17}$$

This gives the partial derivative for each θ_i and allows us to maximize a bound obtained by a neural network directly with gradient ascent. Interestingly, this signal is non-zero when the degree of the node is different than 2 in the 1-tree. This is aligned with the intuition that we want to adjust the multipliers of conflicting nodes.

We note that the derivative obtained is correct only *locally* and not *globally*. Indeed, the cost function in Equation (13) is an optimization problem consisting in finding a minimum 1-tree (i.e., setting variables x) from the current costs c' and that the values of x will depend on θ. We experimented with a globally valid derivative by computing all insertion and replacement costs to integrate them in the derivative of x, but this resulted latter in an unstable training, likely because of the non-convexity of the optimization landscape. For such a reason, we carried out a *subgradient ascent* on a locally valid derivative instead. Variations of variables x are then taken into account in the subsequent gradient ascent step. We have empirically observed increased stability as a result of this procedure.

The training procedure is formalized in Algorithm 1. It gives as output the parameters w of the trained neural network Φ_w. We note that this training loop can be easily improved with standard techniques in deep learning, such as mini-batches or using another gradient-based optimizer, such as Adam [32]. Unlike gradient descent, we aim to maximize the bound, explaining the $+$ term at Line 10. We highlight that the training is *unsupervised* as it does not require ground truth on known tight bounds for training the model, nor the corresponding Lagrangian multipliers. In each iteration of the algorithm, the values of x will change as the multipliers (θ) are updated. This explains how the variations of x are implicitly considered during each subgradient ascent step. Finally, two aspects of the methodology require clarification: the architecture of the network Φ_w and how the training set $\mathcal{D}$ is built. Both are discussed in the following sections.

Algorithm 1 Training phase from an input graph $G(V, E)$.

```
▷ Pre: $D$ is the set of instances used for training.
▷ Pre: $\Phi_w$ is the differentiable model to train.
▷ Pre: $w$ are randomly initialized parameters.
▷ Pre: $K$ is the number of training epochs.

for $k$ from $1$ to $K$ do
    $G := \mathsf{SampleFromTrainingSet}(\mathcal{D})$
    $\theta := \Phi_w(G)$
    $\mathcal{T}_\theta(c') := \mathsf{HeldKarpRelaxation}(G, \theta)$
    $\mathsf{HK}(\theta) := \mathsf{cost}(\mathcal{T}_\theta(c')) - 2\sum_{i=1}^{|V|} \theta_i$
    $w := w + \nabla_w \mathsf{HK}(\Phi_w(G))$
end for
return $w$
```

3.1 Training Set Construction

The training is carried out from a dataset $\mathcal{D}$ consisting of a set of graphs $G(V, E)$ serving as TSP instances. The graphs can either be obtained from historical problem instances (e.g., previous routing networks and costs for a delivery company) or randomly generated. Each graph has six features f_i for each node $i \in V$ and three features $k_{i,j}$ for each edge $(i, j) \in E$. The features we used are presented in Table 1 and in Table 2. We have incorporated the features we believe are important for this task, but we have not analyzed the individual impact of each feature. One strength of deep neural networks is their ability to learn to disregard features that are not beneficial for the task. Although most of the features are relatively standard, $k^2_{i,j}$ and $k^3_{i,j}$ introduce the notions of *mandatory* and *forbidden* edges. In the context of a branch-and-bound algorithm, some decision variables are fixed after branching operations. An edge is mandatory if it must be part of the TSP solution (i.e., $x_{i,j} = 1$) and it is forbidden if it cannot be in the solution (i.e., $x_{i,j} = 0$). This information is crucial as we plan to compute bounds several times during a branch-and-bound execution, with the intention of leveraging partial solutions to get better bounds.

Table 1 Summary of the features on nodes i for each $i \in V$ used in an input graph $G(V, E)$.

Symbol	Formalization	Description
$f^1_i, f^2_i \in \mathbb{R}^2$	$\mathsf{xPos}(i), \mathsf{yPos}(i)$	2-dimensional coordinate of the node.
$f^3_i \in \mathbb{R}$	$\frac{1}{\lvert V\rvert}\sum_{j=1}^{\lvert V\rvert} \lVert\mathsf{coord}(i) - \mathsf{coord}(j)\rVert_2$	Average euclidean distance with the other nodes.
$f^4_i \in \mathbb{R}$	$\min_{j \neq i} \left(f^3_1, \dots, f^3_j, \dots, f^3_{\lvert V\rvert}\right)$	Distance to the nearest node in the graph.
$f^5_i \in \mathbb{N}^+$	$\mathsf{deg}(i)$	Degree in terms of incoming and outgoing edges.
$f^6_i \in \{0, 1\}$	1 iff $i = 1, 0$ otherwise	Indication if it is the excluded node in $G\backslash\{1\}$.

A direct observation is that there are no fixed edges at the root node of a branch-and-bound tree, and consequently, for none of the instances in the training set. This causes a distributional shift between instances used for the training (only at the root node) and the ones occurring at the testing phase (also inside the branch-and-bound tree). To address this limitation, we propose to enrich the training set with partially solved TSP instances extracted from explored branch-and-bound nodes. In practice, it is done by fixing a threshold $k \in \mathbb{N}^+$ on the number of nodes to consider in the training set. This makes the computation tractable as it avoids considering all the nodes of an exponentially sized tree search.

Table 2 Summary of the features on edges (i, j) for each $(i, j) \in E$ used in an input graph $G(V, E)$.

Symbol	Formalization	Description
$k^1_{i,j} \in \mathbb{R}$	$c_{i,j}$	The cost of the edge.
$k^2_{i,j} \in \{0, 1\}$	1 iff (i, j) is forbidden, 0 otherwise	Binary value indicating if the edge is *forbidden.*
$k^3_{i,j} \in \{0, 1\}$	1 iff (i, j) is mandarory, 0 otherwise	Binary value indicating if the edge is *mandatory.*

3.2 Graph Neural Network Architecture

A TSP instance exhibits a natural graph structure. For this reason, we built the model Φ_w with a *graph neural network* [47, 33] (GNN). This architecture has been widely used in related works for the TSP, thanks to their ability to handle instances of different size, to leverage node and edge features, etc. In its standard version, GNNs are dedicated to computing a vector representation of each node of the graph. Such a representation is commonly referred to as an *embedding.* The embedding of a specific node is computed by iteratively transforming and aggregating information from the neighboring nodes. Each aggregation operation is referred to as a layer of the GNN and involves weights that must be learned. This operation can be performed in many ways, and there exist in the literature different variants of GNNs. An analysis on the performances of various architectures is proposed by Dwivedi et al. (2023) [17]. Our model is based on the *edge-featured graph attention network* [53] which is a variant of the well-known *graph attention network* [50] dedicated to handle features on the edges. The whole architecture is differentiable and is trained with backpropagation.

Let $G(V, E)$ be the input graph, $f_i \in \mathbb{R}^6$ be a vector concatenating the 6 features of a node $i \in V$, and $k_{i,j} \in \mathbb{R}^3$ be a vector concatenating the three features of an edge $(i, j) \in E$. The GNN architecture is composed of L layers. Let $h^l_i \in \mathbb{R}^d$ be a d-dimensional vector representation of a node $i \in V$ at layer $l \in \{1, \dots, L\}$, and let $h^{l+1}_i \in \mathbb{R}^{d'}$ a d'-dimensional vector representation of i at the next layer. The inference process consists in computing the next representation (h^{l+1}_i) from the previous one h^l_i for each node i. The first representation is set with the initial features of the node, i.e. $h^1_i = f_i$ for each $i \in V$. The computation is formalized in Equations (18) to (20), where w^l_1 and w^l_2 are two weight tensors that need to be trained for each layer.

$$h^{l+1}_i = \mathsf{ReLU}\Big(\sum_{j \in \mathcal{N}(i)} \alpha^l_{(i,j)} w^l_1 h^l_j\Big) \qquad \forall i \in V \ \wedge \ \forall l \in \{1, \dots, L-1\} \quad (18)$$

$$\alpha^l_{(i,j)} = \mathsf{Softmax}\Big(\mathsf{LeakyReLU}\Big(w^l_2 \times (h^l_i \| k_{i,j} \| h^l_j)\Big)\Big) \quad (19)$$

$$\theta_v = \mathsf{FCNN}\big(h^{|L|}_i\big) \qquad \forall i \in V \quad (20)$$

Equation (18) shows the message passing operation in a layer. Each node i aggregates information of all its neighbors $\mathcal{N}(i)$. The aggregation is subject to parameterized weights w^l_1 and a *self-attention score* $\alpha^l_{i,j}$. This score allows the model to put different weights on the incoming messages from neighboring nodes. We note that the attention integrates information about the node itself (h^l_i), its neighbor (h^l_j), and the features attached to the adjacent edge $(k_{i,j})$. Such information is concatenated $(.\|.)$ into a single vector. Non-linearities are added after each aggregation and the final node embeddings $h^{|L|}_i$ are given as input to a fully-connected neural network (FCNN) outputing the corresponding θ_i multiplier for each $i \in V$. The GNN has 3 graph attention layers with a hidden size of 32 and the fully-connected neural network has 2 layers with 32 neurons.

4 Experimental Evaluation

The goal of the experiments is to evaluate the efficiency of the approach to speed-up a TSP solver based on branch-and-bound and constraint programming [5]. To do so, the learned bounds are integrated into the Held-Karp relaxation used by the *weighted circuit* constraint [4] and are used to filter unpromising edges. The model is used only for the 10 first nodes expanded in the branch-and-bound tree (parameter k). We refer to HK for the standard solver of Benchimol et al. (2012) [5] and to HK+GNN for the one we introduce. We also considered a version using the learned multipliers but without the Held-Karp refinement (i.e., GNN without HK) but the results showed that the bounds obtained only with the learned multipliers alone were far from the optimal bound and are not included in the next experiments. Combining the learned bounds with the Held-Karp refinement is thus required.

4.1 Experimental Protocol

This section outlines the experimental protocol employed to evaluate the efficacy and reliability of our approach. It details the specific datasets, hardware configurations, software tools, and performance metrics used across various testing scenarios.

4.1.1 Datasets

Five datasets of different complexity are considered. They correspond to variants of the metric TSP (i.e., the graphs are complete and the distances are euclidean) on which the cities are localized with different patterns.

1. Random100 (and 200): the cities (100 or 200) are uniformly generated in the $[0, 1]^2$ plan.
2. Clustered100 (and 200): inspired by Fischetti and Toth (1989) [21], five clusters are uniformly generated in the $[0, 1]^2$ plan. Then, the cities (100 or 200) are uniformly generated inside the 0.1-radius circles for each cluster.
3. Hard: introduced by Hougardy and Zhong (2021) [27], these 50 instances ranging from 52 to 199 cities have been generated to have a large integrality gap and are provably hard to solve for branch-and-bound methods.

A test set of 50 instances is built for each configuration and is used for evaluation. For the last dataset, as it is relatively small, only 6 instances are taken for evaluation.

4.1.2 Implementation

The graph neural network has been implemented with *deep graph library* [51] and *Pytorch* [43]. During the training, the minimum spanning trees have been computed with *NetworkX* [24]. We used the *C++* implementation of Benchimol et al. (2012) [5] for the branch-and-bound solver. The interface between the *python* and the *C++* code has been done with native functions from both languages. The solver used to compute the optimal solution is *Concorde* [3]. The implementation and the datasets used are released on Github[1].

4.1.3 Training

The training phase corresponds to the execution of Algorithm 1. A specific model is trained for the five configurations. The training sets for Random and Clustered have 100 instances sampled from the given generating scheme. For the Hard dataset, 40 instances uniformly

[1] `https://github.com/corail-research/learning-hk-bound`

sampled from the 50 available instances are taken. For each instance, 10 subgraphs are generated (parameter k) and are added to the training set. They correspond to partially solved instances that could be found inside the branch-and-bound tree. We highlight that we do not need to label the training instances with known multipliers as the learning is unsupervised. Training time is limited to 4 hours on a AMD Rome 7502 2.50GHz processor with 64GB RAM. No GPU has been used. Models are trained with a single run and we observed the convergence of the loss function on a validation set of 20 instances. The Adam optimizer with a learning rate of 10^{-3} has been used.

4.1.4 Hyperparameters

The branch-and-bound has been configured with the standard settings and most of the hyperparameters follow the recommended values. No hyperparameter tuning has been carried out due to our limited resources. We think that our results can be improved with a more thorough calibration (e.g., slightly changing the number of layers), but as our goal was not to build the most efficient neural network but rather to show the promise of using them to get valid multipliers, we have not carried out this operation. A notable exception is the threshold k on the maximum number of nodes which has an important impact on the performances. We tested values in $\{1, 5, 10, 20, 50\}$ and selected $k = 10$ as it provided the best results and trade-off between accuracy and computation time.

4.1.5 Evaluation Metrics

Five metrics are reported in the results: (1) the execution time to prove the optimality of a solution, (2) the number of instances solved to optimality, (3) the percentage of edges filtered before branching through the propagation, (4) the optimality gap for unsolved instances, and (5) the primal-dual integral. This last metric measures the convergence of the dual bound and the primal bound over the whole solving time [10]. At each iteration, we record the best primal and dual bounds during the whole solving process. Then, the primal-dual integral value consists in taking the area between the two resulting curve obtained. Intuitively, the smaller the better. Each instance is solved only once per experiment as no randomness is involved in the execution. We clarify that our objective is not to compete with Concorde, which is capable of optimally solving all the instances we consider. Instead, our goal is to enhance the filtering of the weighted circuit constraint. This constraint can then be used to more complex variants of the Traveling Salesman Problem, such as the TSP with time windows.

4.2 Empirical Results

This section presents the results obtained from our experimental analysis. These findings are compiled to provide a quantitative evaluation of the performance and effectiveness of our approach under different scenarios. Each experiment is designed to test a specific hypothesis. Finally, a discussion about current limitations are proposed.

4.2.1 Main Results: Quality of the Learned Bounds

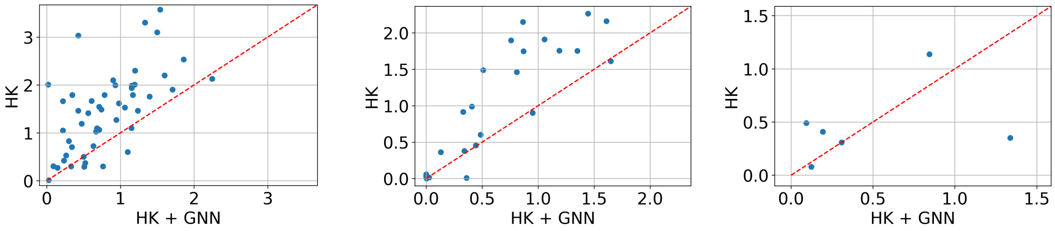
Table 3 summarizes the results for HK and HK+GNN on the five datasets. Values are averaged for each configuration. First, we can observe that our approach gives consistently better results on all the metrics compared to the baseline. As expected, it provides better filtering on the edges. This is reflected by a higher number of solved instances, a reduced execution time, and a reduced optimality gap for unsolved instances. The primal-dual integral confirms that tighter dual bounds are obtained during the search. Second, we notice that the improvements on the Hard dataset are more modest. This can be explained by the fact that they are designed to be challenging. It is consequently more difficult to get improvements on these ones. We note that the time savings obtained with our methods could not provide to a potential faster calculation of the multipliers. The reason is that the cost of computing the multipliers is higher with our method as we combine both the prediction and the iterative HK process (HK+GNN). The speed-up is consequently due to an improved filtering.

Table 3 Comparison of our approach (HK + GNN) with the standard branch-and-bound of Benchimol et al. (HK). The primal bound is 2% above the optimal solution cost computed with Concorde. The statistics considered are: the execution time up to a timeout of 1,800 seconds (*Time*), the number of instances solved to optimality with proof (*# solved*), the primal-dual integral (*PDI*), the percentage of edges filtered (*Filt.*) and the optimality gap for unsolved instances (*Gap*). The relative improvement compared to the baseline is also depicted.

Configuration	Branch-and-bound with standard Held-Karp (HK)					Branch-and-bound with our approach (HK+GNN)				
	Time (sec.)	# solved	PDI	Filt. (%)	Gap (%)	Time (sec.)	# solved	PDI	Filt. (%)	Gap (%)
Random100	559	41/50	1127k	75.9	0.88	497 (- 11%)	46/50 (+ 5)	965k (- 14%)	77.7 (+ 2%)	0.48 (- 45%)
Random200	1800	0/50	4.71m	67.8	1.82	1800	0/50 (+ 0)	4.26m (- 10%)	70.6 (+ 4%)	0.59 (- 68%)
Clustered100	643	38/50	497k	17.7	0.19	590 (- 8%)	40/50 (+ 2)	470k (- 5%)	20.3 (+ 15%)	0.08 (- 58%)
Clustered200	1800	0/50	922k	9.9	0.68	1800	0/50 (+ 0)	690k (- 25%)	12.6 (+27%)	0.38 (- 44%)
Hard	1800	0/6	9.59M	6.4	0.32	1800	0/6 (+ 0)	9.36M (- 2%)	6.5 (+1%)	0.31 (- 3%)

4.2.2 Analysis: Focus on Individual Instances

Figure 4 provides an analysis of the optimality gap for the three hardest configurations by means of scatter plots. Each dot corresponds to a specific instance. When a dot is upper than the diagonal, it means that our approach provided better results than the baseline. Unlike the previous experiments, it provides insights about the robustness of the method. For the majority of the instances, our approach gave better or similar results, except for one instance in the Hard dataset.



(a) Results on Random200. **(b)** Results on Clustered200. **(c)** Results on Hard.

Figure 4 Scatter plots comparing the optimality gap (%) for HK and HK + GNN on the three hardest configurations.

4.2.3 Analysis: Addressing the Optimality Proof

This next experiment evaluates the ability of proving the optimality of a solution *only once this solution has been found.* Concretely, instead of taking a reasonable upper bound of 2%, we assume that the optimal solution has been found and we use it as a perfect upper bound. The idea is to mimic results achieved by a good primal heuristic and to close the search by proving the optimality of the solution provided with this information. Results are summarized in Table 4. In such a situation, the improvements with the baseline are still positive, especially for the largest and hardest configurations. This shows the potential of learning dual bounds to accelerate optimality proofs and the synergies with approaches operating in the primal side.

Table 4 Comparison of our approach (HK + GNN) with the standard branch-and-bound of Benchimol et al. (HK) for optimality proof. Compared to Table 3, the primal bounds are now the cost of the optimal solution.

Configuration	Branch-and-bound with standard Held-Karp (HK)					Branch-and-bound with our approach (HK+GNN)				
	Time (sec.)	# solved	PDI	Filt. (%)	Gap (%)	Time (sec.)	# solved	PDI	Filt. (%)	Gap (%)
Random100	209	48/50	591k	92.1	0.05	203 (- 3%)	48/50 (+ 0)	544k (- 7%)	92.2 (+ 0%)	0.05 (- 0%)
Random200	1800	0/50	4.44m	85.6	1.16	1800	0/50 (+ 0)	4.11m (- 7%)	89.1 (+ 4%)	0.43 (- 62%)
Clustered100	112	48/50	103k	25.4	0	110 (- 1%)	49/50 (+ 1)	100k (- 3%)	25.5 (+ 0%)	0 (- 0%)
Clustered200	1800	0/50	713k	14.5	0.43	1800	0/50 (+ 0)	644k (-9%)	16.8 (+15%)	0.30 (- 30%)
Hard	1800	0/6	7.59M	17.8	0.26	1800	0/6 (+ 0)	7.02M (- 7%)	18.0 (+ 1%)	0.19 (- 24%)

4.2.4 Analysis: Generalization Ability

This last experiment analyzes how the models are able to generalize to new configurations, either for a higher number of cities or with another generation scheme. Concretely, we considered four configurations (Random100, Random200, Clustered100 and AllDataset, the latter being trained on the instance of all datasets) and evaluated them on Clustered200. The idea of this experiment is to compare the generalization ability of models trained on specific datasets (Random100, Random200 and Cluster100). Results on AllDataset provide an idea of the best performances that the specific models can achieve. Results are presented in Table 5. Although the performance of the specific model is not reached, we observe that the models trained on the other distributions are still able to outperform the standard model. Training a model on all datasets (AllDataset) allows to improve upon out-of-distribution models but does not achieve the performance of the specialized model. This confirms the intuitive benefit to know beforehand the distribution of the instances to solve.

Table 5 Analysis of the generalization. The different models are used to solve Clustered200 graphs.

Model	Branch-and-bound with HK+GNN		
	PDI	Filt. (%)	Gap (%)
Clustered200	690k	12.6	0.38
HK without GNN	922k	9.9	0.68
Clustered100	817k	11.1	0.54
Random100	845k	10.4	0.61
Random200	784k	11.3	0.49
AllDataset	722k	12.0	0.45

4.2.5 Discussion: Application to Non-Euclidean Instances

Previous experiments have shown promising results for various configurations of metric TSPs. However, it is important to note that the performance on other types of TSP instances, such as asymmetric or non-Euclidean TSPs, which are known to be more challenging, has not been evaluated in this paper. Consequently, it remains uncertain whether the observed performance will extend to these more complex instances. Exploring this further constitutes an interesting avenue for future research. Such an analysis can be facilitated using the instances available in TSPLib [45], providing a robust framework for testing under more diverse conditions.

4.2.6 Discussion: Considering the Training Time

As is common practice with machine learning tools, our framework assumes that we can train a model offline before its deployment to solve new instances, ideally following a similar distribution. In such cases, the training time can be disregarded, as it will be amortized over a large number of future instances. However, this assumption does not hold in all realistic scenarios where training time cannot be separated from solving time. In such situations, while our approach remains applicable, the time required to train the model must be considered and can be prohibitive. To address this, an initial analysis could involve monitoring how the learned bounds improve with training time. This would provide a more detailed understanding and help identify when training can be stopped to proceed directly to the solving phase. Intuitively, this approach may offer slightly less effective filtering compared to HK+GNN, but it is aimed to result in a reduced total execution time.

5 Conclusion

Learning-based methods have been extensively considered to provide approximate solutions to combinatorial optimization problems, such as the travelling salesman. However, learning to obtain dual bounds has been less considered in the literature and is much more challenging as there is no trivial way to ensure that the obtained bounds are valid. This paper introduces an unsupervised learning approach, based on graph neural networks and the Held-Karp Lagrangian relaxation, to tackle this challenge. The core idea is to predict accurate Lagrangian multipliers and employ them as a warm start for generating Held-Karp relaxation bounds. These bounds are subsequently used to enhance the filtering level of the weighted circuit global constraint and improve the performances of a branch-and-bound algorithm. The empirical results on different configurations of the TSP showed that the learning component can significantly improve the algorithm. We believe that the methodology can be integrated into existing CP solvers. To do so, one will need to refactor the weighted circuit global constraint implemented in the related solver to add the learned bounds. Although centered on the TSP, we note that weighted circuit global constraint could be used for other, and more challenging, TSP variants including time windows or time-dependent costs. Analyzing how these variants can be handled efficiently is part of our future work. Also, we believe that the methodology can be easily reused for other propagators using Lagrangian relaxation, such as for the *AtMostNValue* [9] or for the general framework of CP-based Lagrangian relaxation [48].

References

1 Ahmed Abbas and Paul Swoboda. DOGE-train: Discrete optimization on GPU with end-to-end training. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(18):20623–20631, 2024.

2 Brandon Amos and J. Zico Kolter. OptNet: Differentiable optimization as a layer in neural networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 136–145. JMLR.org, 2017.

3 David L. Applegate, Robert E. Bixby, Vašek Chvatál, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006. URL: `http://www.jstor.org/stable/j.ctt7s8xg`.

4 Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in CHIP. *Mathematical and computer Modelling*, 20(12):97–123, 1994.

5 Pascal Benchimol, Willem-Jan van Hoeve, Jean-Charles Régin, Louis-Martin Rousseau, and Michel Rueher. Improved filtering for weighted circuit constraints. *Constraints*, 17:205–233, 2012.

6 Pascal Benchimol, Jean-Charles Régin, Louis-Martin Rousseau, Michel Rueher, and Willem-Jan Van Hoeve. Improving the Held and Karp approach with constraint programming. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 40–44. Springer, 2010.

7 Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

8 David Bergman, Andre A Cire, and Willem-Jan van Hoeve. Improved constraint propagation via lagrangian decomposition. In *International Conference on Principles and Practice of Constraint Programming*, pages 30–38. Springer, 2015.

9 Frédéric Berthiaume and Claude-Guy Quimper. Local alterations of the lagrange multipliers for enhancing the filtering of the atmostnvalue constraint. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 68–83. Springer, 2024.

10 Timo Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6):611–614, 2013.

11 Quentin Cappart, David Bergman, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Augustin Parjadis. Improving variable orderings of approximate decision diagrams using reinforcement learning. *INFORMS Journal on Computing*, 34(5):2552–2570, 2022.

12 Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Velickovic. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023. URL: `http://jmlr.org/papers/v24/21-0449.html`.

13 Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre A Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3677–3687, 2021.

14 Martin C Cooper, Simon De Givry, Martı Sánchez, Thomas Schiex, Matthias Zytnicki, and Tomas Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8):449–478, 2010.

15 Yanchen Deng, Shufeng Kong, Caihua Liu, and Bo An. Deep attentive belief propagation: Integrating reasoning and learning for solving constraint optimization problems. *Advances in Neural Information Processing Systems*, 35:25436–25449, 2022.

16 Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning heuristics for the TSP by policy gradient. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 170–181. Springer, 2018.

17 Vijay Prakash Dwivedi, Chaitanya K Joshi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *Journal of Machine Learning Research*, 24(43):1–48, 2023.

18 Adam N Elmachtoub and Paul Grigas. Smart "predict, then optimize". *Management Science*, 68(1):9–26, 2022.

19 Aaron Ferber, Bryan Wilder, Bistra Dilkina, and Milind Tambe. MIPaaL: Mixed integer program as a layer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1504–1511, 2020.

20 Aaron M Ferber, Taoan Huang, Daochen Zha, Martin Schubert, Benoit Steiner, Bistra Dilkina, and Yuandong Tian. SurCo: Learning linear surrogates for combinatorial nonlinear optimization problems. In *International Conference on Machine Learning*, pages 10034–10052. PMLR, 2023.

21 Matteo Fischetti and Paolo Toth. An additive bounding procedure for combinatorial optimization problems. *Operations Research*, 37(2):319–328, 1989.

22 Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

23 Minh Hoàng Hà, Claude-Guy Quimper, and Louis-Martin Rousseau. General bounding mechanism for constraint programs. In *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*, pages 158–172. Springer, 2015.

24 Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Lab. Los Alamos, NM (United States), 2008.

25 Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970. URL: http://www.jstor.org/stable/169411.

26 Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 18(1):6–25, 1971. doi:10.1007/BF01584070.

27 Stefan Hougardy and Xianghui Zhong. Hard to solve instances of the euclidean traveling salesman problem. *Mathematical Programming Computation*, 13:51–74, 2021.

28 Vinasetan Ratheil Houndji, Pierre Schaus, Mahouton Norbert Hounkonnou, and Laurence Wolsey. The weighted arborescence constraint. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 185–201. Springer, 2017.

29 Chaitanya K Joshi, Quentin Cappart, Louis-Martin Rousseau, and Thomas Laurent. Learning the travelling salesperson problem requires rethinking generalization. *Constraints*, 27(1-2):70–98, 2022.

30 Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6351–6361, 2017.

31 Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1), February 2016. doi:10.1609/aaai.v30i1.10080.

32 Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

33 Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017. URL: https://openreview.net/forum?id=SJU4ayYgl.

34 Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

35 James Kotary, Ferdinando Fioretto, Pascal Van Hentenryck, and Bryan Wilder. End-to-end constrained optimization learning: A survey. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 4475–4482. International Joint Conferences on Artificial Intelligence Organization, August 2021. Survey Track. `doi:10.24963/ijcai.2021/610`.

36 Jena-Lonis Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial intelligence*, 10(1):29–127, 1978.

37 Yang Li, Jinpei Guo, Runzhong Wang, and Junchi Yan. From distribution learning in training to gradient search in testing for combinatorial optimization. *Advances in Neural Information Processing Systems*, 36, 2024.

38 Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *TOP*, 25(2):207–236, 2017.

39 Jayanta Mandi, Emir Demirovic, Peter J Stuckey, and Tias Guns. Smart predict-and-optimize for hard combinatorial optimization problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1603–1610, 2020.

40 Jayanta Mandi, James Kotary, Senne Berden, Maxime Mulamba, Victor Bucarey, Tias Guns, and Ferdinando Fioretto. Decision-focused learning: Foundations, state of the art, benchmark and future opportunities. *arXiv preprint arXiv:2307.13565*, 2023.

41 Daniel Merchán, Jatin Arora, Julian Pachon, Karthik Konduri, Matthias Winkenbach, Steven Parks, and Joseph Noszek. 2021 Amazon last mile routing research challenge: Data set. *Transportation Science*, 2022.

42 Mathias Niepert, Pasquale Minervini, and Luca Franceschi. Implicit MLE: backpropagating through discrete exponential family distributions. *Advances in Neural Information Processing Systems*, 34:14567–14579, 2021.

43 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

44 Marin Vlastelica Pogančić, Anselm Paulus, Vit Musil, Georg Martius, and Michal Rolinek. Differentiation of blackbox combinatorial solvers. In *International Conference on Learning Representations*, 2019.

45 Gerhard Reinelt. TSPLIB — A traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

46 Utsav Sadana, Abhilash Chenreddy, Erick Delage, Alexandre Forel, Emma Frejinger, and Thibaut Vidal. A survey of contextual optimization methods for decision-making under uncertainty. *European Journal of Operational Research*, 2024.

47 Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

48 Meinolf Sellmann. Theoretical foundations of CP-based lagrangian relaxation. In *International Conference on Principles and Practice of Constraint Programming*, pages 634–647. Springer, 2004.

49 Daniel Selsam, Matthew Lamm, B Benedikt, Percy Liang, Leonardo de Moura, and David L Dill. Learning a SAT solver from single-bit supervision. In *International Conference on Learning Representations*, 2018.

50 Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. URL: `https://openreview.net/forum?id=rJXMpikCZ`.

51 Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

52 Po-Wei Wang, Priya Donti, Bryan Wilder, and Zico Kolter. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *International Conference on Machine Learning*, pages 6545–6554. PMLR, 2019.

53 Ziming Wang, Jun Chen, and Haopeng Chen. EGAT: Edge-featured graph attention network. In Igor Farkaš, Paolo Masulli, Sebastian Otte, and Stefan Wermter, editors, *Artificial Neural Networks and Machine Learning – ICANN 2021*, pages 253–264, Cham, 2021. Springer International Publishing.

54 Tomas Werner. A linear programming approach to max-sum problem: A review. *IEEE transactions on pattern analysis and machine intelligence*, 29(7):1165–1179, 2007.

55 Bryan Wilder. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, 2019.

56 Liang Xin, Wen Song, Zhiguang Cao, and Jie Zhang. NeuroLKH: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem. *Advances in Neural Information Processing Systems*, 34:7472–7483, 2021.

57 Kaan Yilmaz and Neil Yorke-Smith. A study of learning search approximation in mixed integer branch and bound: Node selection in SCIP. *AI*, 2(2):150–178, April 2021. `doi:10.3390/ai2020010`.

Constraint Programming Model for Assembly Line Balancing and Scheduling with Walking Workers and Parallel Stations

Xavier Pucel[1]
ONERA, ONERA DTIS, Toulouse, Université de Toulouse, France

Stéphanie Roussel
ONERA, ONERA DTIS, Toulouse, Université de Toulouse, France

Abstract

In the context of aircraft assembly lines, increasing the production rate and decreasing the operating costs are two important, and sometimes contradictory, objectives. In small assembly lines, sharing production resources across workstations is a simple and efficient way to reduce operating costs. Therefore, workers are not assigned to a unique workstation but can walk between them. On the other side, paralleling workstations is an efficient way to increase the production rate. However, the combination of both strategies create complex conditions for tasks to access the production resources. This paper addresses the problem of allocating tasks to workstations and scheduling them in an assembly line where workers can freely walk across workstations, and where workstations can be organized in parallel. We model this novel problem with Constraint Programming. We evaluate it on real world industrial use cases coming from aircraft manufacturers, as well as synthetic use cases adapted from the literature.

2012 ACM Subject Classification Applied computing → Computer-aided manufacturing

Keywords and phrases Constraint Programming, Assembly Line, Balancing and Scheduling, Parallel Workstations, Walking Workers

Digital Object Identifier 10.4230/LIPIcs.CP.2024.23

Supplementary Material *Dataset (Assembly Line Design)*: `https://doi.org/10.57745/EWXS9O` [29]

Funding This work was funded by the European Union's NextGenerationEU program, and the national France Relance program. It was coordinated by the Directorate General of Civil Aviation (DGAC) in France.

1 Introduction

The production of complex artefacts such as aircraft is commonly performed by the use of assembly lines. In an assembly line, several workstations are dedicated to a subset of assembly tasks, and the artefacts being built travel from one workstation to another until they are complete. As each workstation is always occupied by one instance of the artefact, all instances visit the workstations in the same order, and travel at the same time, after a fixed time interval called the *cycle time*. Since the cycle time is the duration between the completion of two instances of the artefact, its value is of critical importance for business purposes, and a lot of research aims at improving (i.e. minimizing) its value for all sorts of production systems.

A natural way to improve the production rate, without modifying the artifact design, is simply to add more workstations. Since the same amount of work is divided in a larger number of workstations, each workstation has less tasks and complete them faster, thus

[1] Corresponding author

decreasing the cycle time. The main drawback of adding workstations is that it occupies more space in the factory and may not always be possible. Furthermore, even when it is possible, the cost of creating a new workstation can be another obstacle. While in large assembly lines each workstation has its dedicated equipment and operators, this is not necessarily the case in smaller assembly lines, where it is possible for operators to travel, and equipment to be transferred, across workstations. In this context, adding a new workstation can be much cheaper if expensive equipment does not need to be duplicated.

Another way to increase the production rate is to act on the artefact flow, and in particular to extend a production stage with parallel workstations. The requirement of having each task confined to a single workstation then translates to weaker temporal constraints, and it even allows the cycle time to be smaller than the duration of some tasks. It also has the drawback of occupying more space in the factory since more routes must be cleared for the artifacts to travel. Moreover, parallel workstations with shared resources can create complex situations in terms of conflicting access by tasks to resources. These conditions are precisely the highlight of this paper.

In this paper, inspired by an industrial assembly line for small aircraft, we address the problem of increasing the production rate, and use it to evaluate different assembly line structures (number of stages, number of parallel workstations), with a predefined amount of resources. More precisely, our assembly line model has the following features:

- Minimizing the assembly duration.
- The number and layout of workstations is an input of the problem. The layout specifies the number of serial stages, and the number of parallel duplicate workstations in each stage.
- Each workstation has different zones (e.g. inside or outside of the aircraft, front or rear, etc.), and each task occupies one or more zones. Two tasks that occupy the same zone cannot be performed simultaneously.
- Workers can walk freely across workstations. The travel time between workstations is not accounted for, as it is considered negligible with respect to the work time.
- Each worker has one specific skill, and each task requires a given number of workers with each skill (including zero). At each instant a worker can only perform one task, which means tasks compete with one another for accessing workers. As we have walking workers, this competition occurs not only inside workstations, but also across workstations.
- As parallel workstations allow for tasks longer than the cycle time, such a long task can compete with itself for accessing workers. More precisely, two instances of the task performed on two parallel workstations, but shifted by only one cycle, overlap, and thus require twice the amount of workers. This aspect is original, and is explained and illustrated in detail.
- In the problem output, in addition to the smallest possible cycle time, each assembly task is assigned to a unique workstation (or rather a stage in the case of parallel workstations), and is given start and end dates. Moreover, tasks are not preemptive.

We can summarize the contribution of this work as follows:

- we address a novel extension of the Assembly Line Balancing (ALB) problem in which we schedule all tasks, some resources can walk between stations and stations can be parallel;
- we propose a Constraint Programming (CP) model for this problem ;
- we present results associated with data coming from an aircraft manufacturer, which show the applicability of the approach;

- we experiment on public benchmarks that we have adapted to our hypotheses: some corresponding to aircraft assembly lines and the others based on public Resource-Constrained Project Scheduling Problem (RCPSP) instances.

This paper is organized as follows. First the state of the art is reviewed in Section 2, around the ALB and RCPSP variants, and constraint programming in the aeronautics industry. Second, a more detailed account of the assembly line is provided in Section 3, and in particular how parallel workstations behave, and how it impacts the competition for resources between tasks. The CP model is then described in Section 4 along with an illustrative example. Finally, Section 5 relates our experiments on both academic benchmarks and industrial examples. We describe perspectives of this work in Section 6.

2 Related Work

The work presented in this paper can be compared with two well known optimisation problems: the Assembly Line Balancing (ALB) problem and the Resource-Constrained Project Scheduling Problem (RCPSP). ALB essentially deals with assigning tasks to workstations [11]. In particular, the Simple Assembly Line Balancing Problem version 2 (SALB2) aims at minimizing the cycle time given a list of tasks, a list of workstations and precedence constraints between tasks. However, in a classical ALB problem, tasks are only assigned to workstations, but are not scheduled, i.e. they are not assigned a start and end date.

A family of variants of ALB addresses assembly lines where some workstations perform the same tasks in parallel. More precisely, the assembly line is composed of a series of *stages*, each stage contains one or several parallel workstations. Each product goes through each stage in order, but only on one workstations at each stage. Using parallel workstations helps decreasing the cycle time, but requires more resources. Parallel ALB approaches minimize either the cycle time or total cost associated to workstations, and assign tasks to workstations, but do not schedule them [12, 3, 4, 16].

In a classical RCPSP, given a list of tasks, precedence constraints between tasks, a list of resources and a consumption relation between tasks and resources, the problem is to assign a start date and an end date to each task so as to minimize the total work time [18]. The main difference with ALB is that the work is not divided among synchronized workstations. However, by reasoning on start and end dates, it supports a precise management of resources. Our approach can be seen as an extension of RCPSP with workstations, work zones in each workstation, and cumulative resources shared across workstations. This is how we could evaluate our approach on the PSPlib benchmark [19].

The RCPSP enjoys a vast amount of variants [15]. Among them is the Multi-Skill RCPSP, in which workers have one or several skills that are required for executing tasks. Our approach does not feature this aspect. Each worker has a unique skill, and a set of n workers with a particular skill can be directly modeled as a renewable resource with a capacity equal to n.

There are many extensions of ALB that incorporate RCPSP aspects. Resource Constrained ALB (RCALB) accounts for the resources deployed on each workstation and for how tasks use resources [2, 13, 7], much like RCPSP cumulative resources, or [1] that supports multi-mode RCPSP constraints. In RCALB, resources are assigned to a single workstation, which differs from our approach where resources are shared across workstations.

The variant of Multi-Manned ALB resembles RCALB, except in the case of Walking Workers. These approaches have many common points with ours and are detailed in a dedicated section 2.1. ALB and RCPSP have been combined in other ways described in [8].

Multi-Model ALB models assembly lines where several types of product are manufactured. In this variant, many approaches not only assign tasks to workstations but also schedule them. [24] has a list of tasks with precedence constraints in order to assemble a set of products. Tasks are scheduled on workstations accounting for precedence, equipment, surface and buffers, but completely ignores possible future tasks. In [25] the problem is modified to consider cyclic schedules, in which the list of tasks is expected to be repeated, and the concept of cycle time appears again, event though the assembly line is not pulsated. [21] also computes cyclic schedules for multi-model assembly lines, however the products advance on a treadmill and workstations are not separated and organized into stages.

Some variants of the ALB problem support walking workers. In [32] workers can move from one station to another, and the time to travel between stations is accounted for under a form similar to the Traveling Salesman Problem. However, tasks are not scheduled inside each station. [10] addresses mixed model assembly lines and assigns workers to workstations. However, the assignment of tasks to workstations is predefined as input.

2.1 Multi-manned ALB with Walking Workers

An important family of variants of ALBP concerns Multi-manned ALB (MMALBP). Some of these variants model the workers needed to perform tasks in a way similar to how tasks use resources in a RCPSP, much like this paper does. As a consequence, many approaches in this family not only assign tasks to workstations, but also schedule their start and end dates in order to ensure that the tasks assigned to each worker are feasible, both in volume and precedence. In addition, some of these variants have walking workers that travel across workstations. Many of these variants bear significant similarities with our work.

[14] proposes Mixed Integer Linear Programming (MILP) models for MMALB. Each worker is assigned to a station, and each task is assigned to both a workstation and a worker, and is scheduled inside its station. Workers cannot travel to another workstation, and working zones and parallel workstations are not modelled.

[30] addresses MMALBP with walking workers. Each worker can travel to a predefined set of stations (possibly all), and each station can host several workers at once. Each task is assigned to a station and a worker, and is scheduled inside its station. It only differs with this work in that working zones, and more importantly parallel workstations, are not accounted for.

[23] describes MMALB with moving workers. Each task has a set of compatible workers, which models worker skills, and a set of mutually exclusive tasks, which could be used to model working zones. Each workstation is divided into sides, and each worker is assigned to at most two sides of a single workstation. Each task requires one or several sides, and is assigned to a single station, as well as one worker per required side. Tasks are not scheduled by date, but a partial ordering is computed during optimization. Workers only travel across sides of a single workstation; furthermore parallel workstations are not modelled in this paper.

[22] addresses MMALB in production lines where the product does not advance in pulses from one station to another, but rather advances continuously as on a treadmill. Tasks are assigned to workers and scheduled inside an envelope called “cycle time” so as to occupy a bounded region of the treadmill. However, there are no global date boundaries like in pulsated assembly lines.

[9] defines variable workplaces ALB, in which each workstation has a list of so-called “mounting positions”, that are gathered into workplaces in a possibly different way at each workstation. Tasks are assigned to workplaces that must contain the required mounting

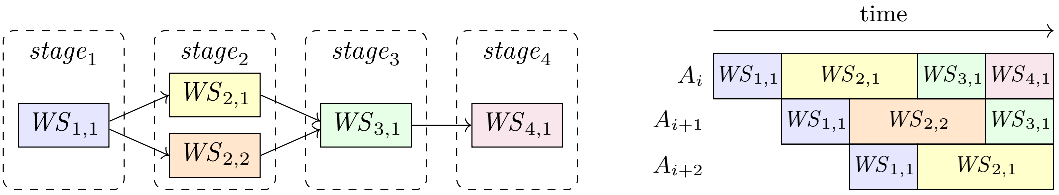


Figure 1 An example of an assembly line with 4 stages and two parallel workstations in the second stage. The left side depicts the physical layout of the assembly line. The right side depicts the temporal schedule of several consecutive aircraft. Note that aircraft A_i and A_{i+2} go through workstation $WS_{2,1}$ while A_{i+1} goes through workstation $WS_{2,2}$. This allows each aircraft to spend 2 cycles in stage 2 without conflict.

positions. Two workplaces cannot overlap on the same workstation. In that work, tasks are scheduled inside workstations, and it accounts for extra-long tasks (longer than one cycle time) by splitting them into smaller tasks.

2.2 ALB and RCPSP Applied to Aircraft Manufacturing

The application of operation research techniques to aircraft manufacturing is not novel. In [31] the problem of scheduling assembly tasks is formulated as a RCPSP problem. Similarly, [5] addresses the problem of assigning operators and models the assembly line as a RCPSP. [17] plans the resources for aircraft turnaround operations as an RCPSP problem. However, the concept of workstation is absent from those studies.

In a more closely related approach, [26] both allocates tasks to workstations and schedules their executions dates. The model accounts for renewable resources and work zones and scales up to a large number of tasks. [27] addresses multi-criteria optimization for the preliminary design of assembly lines. Tasks are both assigned to workstations and scheduled inside them, and work zones and resources such as machines are accounted for. It introduces specific types of constraints such as neutralized zones, and skill exclusion. A simplified version of their use-case was used in the 2023 XCSP3 competition [6]. While they do not account for parallel workstations, in this paper we modify their use-case in the evaluation section.

3 Aircraft Assembly Line Description

The production systems involved in aircraft construction are complex as they are usually composed of several factories that can produce one part of the aircraft or assemble several parts together. Each factory has several key areas: storage, offices, preparation areas, assembly lines, locker rooms, etc. In this work, we focus on the final assembly line of a manufacturer of small aircraft. This final assembly line is in charge of assembling the fuselage, the wings, the propeller and the engine, and of installing electrical and hydraulic systems in the aircraft.

As done classically in the aeronautics industry, the assembly line we consider here is composed of several workstations, and is pulsated. In a sequential assembly line, each aircraft stays a given duration in each workstation before going to the next one or going out of the line after the final workstation. This duration, called *cycle time*, and noted Ct, is imposed on the manufacturer: it is calculated so as to guarantee a certain number of aircraft delivered per month or per year[2]. A set of assembly tasks on the aircraft is performed on each workstation. Tasks are not pre-emptive, i.e. no task can be interrupted, and they cannot overlap several workstations. In fact, no assembly task can be performed while the aircraft moves from one workstation to the next.

In order to be executed, tasks can require equipment and operators with particular skills. Each of them is only available in limited quantity, and we represent it through an abstract cumulative renewable resource. All tools and operators are shared by all the workstations of the assembly line. If a task requires a shared resource in a given workstation, it impacts the resource's available capacity on all other workstations during the task execution. For instance, if a resource with capacity 1 is required for the first task of the first workstation, and if this task has a duration equal to 2, then the resource is not available during the two first time units of each cycle in the entire assembly line.

The aircraft is divided into several zones in which tasks are performed. Two tasks that occupy the same zone cannot be performed simultaneously on the same workstation. Zones are represented as renewable resources of capacity 1; however, unlike equipment and operators, each workstation has its own work zones. This means that two tasks that occupy the same zone, need to be scheduled either one after another on a single workstation, or on different workstations. Note that it would be possible to consider zones with capacity greater than 1.

The assembly line considered in this paper features duplicated workstations that operate in parallel. It is organised as a series of *stages*, each stage being composed of one or several identical workstations that operate in parallel. This allows the aircraft to stay longer in these duplicated workstations. Stages with one workstation behave much like in sequential assembly lines. However, in a stage with n parallel workstations, each aircraft stays n times the cycle time on its workstation before heading to the next stage. At each cycle, one aircraft goes out of each stage and into the next one. For instance, in the assembly line illustrated in Figure 1, the second stage contains two workstations. When entering the second stage, each aircraft A_i goes either on workstation $WS_{2,1}$ or workstation $WS_{2,2}$ and stays there for a duration equal to $2 \cdot Ct$ in the example. As $WS_{2,1}$ and $WS_{2,2}$ are shifted of one cycle time, there is still an aircraft entering $WS_{3,1}$ at each cycle.

Parallel duplicate workstations allow the aircraft to stay in a stage for a duration greater than the cycle time. This allows for tasks longer than the cycle time to be realized. In the example of Figure 1, stages 1, 3 and 4 last one cycle time, but stage 2 lasts two cycles, so it can be assigned a task of a duration up to $2 \cdot Ct$.

Note that while a stage is physically, or spatially, decomposed into workstations, it would be misleading to divide the time period an aircraft spends on a stage into time periods called "workstations". For example in Figure 1, while stage 2 physically has 2 workstations, and while aircraft A_i stays two cycles in workstation $WS_{2,1}$, it would be incorrect to refer to the first half of this time period stay as "the first workstation of stage 2", since aircraft A_i will never travel to workstation $WS_{2,2}$. Instead, we say that stage 2 is temporally decomposed into two *substages*, and stages 1, 3 and 4 are composed of one substage.

To summarize, all substages have the same duration of 1 cycle. The assembly line has as many substages as it has workstations. At each instant, each workstation is occupied by exactly one aircraft, and *each aircraft is in a different substage of its construction.* Each aircraft goes through all construction substages in the same order, however it does not necessarily travel through all workstations.

[2] More precisely, the long term production rate, or *takt*, is an objective imposed on the manufacturer. Our work evaluates whether a factory design can reach an operational *cycle time* consistent with the takt under realistic conditions.

Task	Duration	Zone 1	Zone 2	Resource 1	Resource 2
t_1	2	×		2	1
t_2	8	×		1	
t_3	6		×		
t_4	2		×		1
t_5	2	×	×		
t_6	4	×		1	1
t_7	4		×		1
				$capa_1 = 3$	$capa_2 = 2$

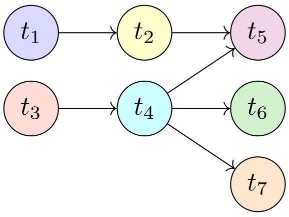
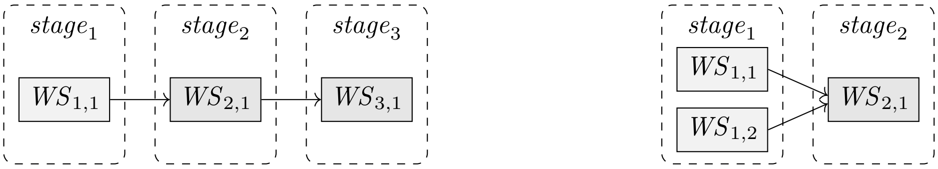


Figure 2 Illustration of the simple assembly line problem detailed in Example 2.



(a) First layout, denoted L_1.

(b) Second layout, denoted L_2.

Figure 3 Illustration of two possible layouts for the problem described in Example 2.

Parallel workstations are a useful tool to increase the production rate. They make it possible to schedule tasks that last more than the cycle time, or conversely to increase the production rate so that the cycle time is smaller than the longest assembly task, which is impossible on sequential assembly lines. The drawbacks of parallel workstations are an increased surface occupation, increased tools and operators requirements, and an increased number of aircraft under construction, which can be a significant financial constraint.

When combined with walking workers, parallel workstations can create counter-intuitive situations, where a task seems to use several times its amount of workers. This happens on stages with duplicated workstations, that are assigned a task longer than the cycle time. Assume in Figure 1 that a task of length $1.5 \cdot Ct$ is executed at the start of stage 2, and in theory only requires 1 worker. When aircraft A_i arrives on workstation $WS_{2,1}$ the task starts and one worker starts working on it, for the next $1.5 \cdot Ct$. One Ct later, aircraft A_{i+1} arrives on workstation $WS_{2,2}$, but the worker is still busy on the aircraft A_i. As a consequence, a task that would only require one worker in a sequential layout, will in fact need two workers in the layout of Figure 1 and with a cycle time shorter than its duration. This aspect is illustrated later in Example 2 and Figure 5.

Our goal is to study the impact of paralleling workstations on the production rate. To that end, we model the assembly line under several layout assumptions and minimize the cycle time under constant resource capacity assumptions.

4 Assembly Line Model

This section introduces definitions for our multi-manned assembly line balancing problem with walking workers and parallel stations. The elements constituting such a problem and an associated solution are described first. Then, a constraint programming model in the OPL ([20]) language is detailed, explained and illustrated.

4.1 Formal Definitions

▶ **Definition 1** (MALBWP). *A Multi-manned Assembly Line Balancing problem with Walking workers and Parallel stations is a tuple* $(\mathcal{T}, nS, \mathcal{Z}, \mathcal{R}, Prec, Occ, Cons)$ *where:*

- $\mathcal{T}$ *is a set of assembly tasks. Each task* $t \in \mathcal{T}$ *is associated with its duration* dur_t.
- nS *is the number of stages.* $\mathcal{S}$ *denotes the set* $[1..nS]$ *of assembly stages, where each stage can have one or more parallel workstations. For each stage* $s \in \mathcal{S}$*, its number of parallel workstations (and thus substages) is noted* w_s*. The total number of substages is noted* $W = \sum_{s \in \mathcal{S}} w_s$.
- $\mathcal{Z}$ *is a set of work zones representing the different areas of the aircraft, in which workers can perform different tasks.*
- $\mathcal{R}$ *is a set of resources or tools that are needed to perform certain tasks. Each resource* $r \in \mathcal{R}$ *is associated with its capacity* $capa_r$.
- $Prec \subseteq \mathcal{T} \times \mathcal{T}$ *is the precedence relation between tasks. For each pair of tasks* $(t_1, t_2) \in Prec$*,* t_2 *cannot start until* t_1 *is finished.*
- $Occ \subseteq \mathcal{T} \times \mathcal{Z}$ *is the occupation relation between tasks and zones.* $(t, z) \in Occ$ *means that task* t *occupies zone* z*. Each task can use any number of zones.*
- $Cons \subseteq \mathcal{T} \times \mathbb{N} \times \mathcal{R}$ *is the resource usage relation between tasks and resources.* $(t, n, r) \in Cons$ *means that task* t *uses* n *units of resource* r*. Each task can use any number of resources. For each resource* r*, we denote* $Cons_r$ *the set* $\{(t, n) | (t, n, r) \in Cons\}$*, i.e. the weighted set of tasks that consumes* r.

This model features two different types of resources. Zones have capacity 1, are specific to each workstation, and are present in all workstations. Resources have a finite capacity (possibly 1), however they are shared across all workstations. Note that this type of problem could easily be extended to workstation-specific resources with finite capacity as in [26].

▶ **Example 2.** Figure 2 describes a small multi-manned assembly line balancing problem with walking workers and parallel stations. The problem contains 7 tasks, 2 zones and 2 resources. The zone occupation relation, the resource consumption relation and the resource capacities are detailed in the table on the left, and the graph induced by the precedence relation is depicted on the right. Resource 1 has capacity 3, and resource 2 has capacity 2.

Figure 3 illustrates two possible layouts for the assembly line with 3 workstations, i.e. $W = 3$. In layout L_1 (Figure 3a) the stage set $S = \{1, 2, 3\}$ contains 3 stages of 1 workstation each, i.e. $w_1 = w_2 = w_3 = 1$. In layout L_2 (Figure 3b), there are two stages $S = \{1, 2\}$, and the first one is composed of two workstations, i.e. $w_1 = 2$ and $w_2 = 1$.

▶ **Definition 3** (Solution). *A solution to a given MALBWP is a tuple* $(Ct, start)$ *defined as follows.*

- Ct *is the pulse rate of the assembly line.*
- *The duration of stage* $s \in \mathcal{S}$ *equals* $w_s.Ct$*, the first stage starts at time* 0 *and each other stage starts when its predecessor stage ends. The start date of stage* s *is denoted* stageStart_s *and its end date is denoted* stageEnd_s.
- $start$ *associates a start date to each task* $t \in \mathcal{T}$*, noted* $start_t$*. The end date of each task, noted* end_t *is the sum of its start date and its duration.*
- *Each task is performed inside a unique stage, i.e. for each task* $t \in \mathcal{T}$*, there exists a unique stage* $s \in \mathcal{S}$ *such that* $\text{stageStart}_s \leq start_t \leq end_t \leq \text{stageEnd}_s$.
- *Two tasks that occupy the same zone do not overlap temporally. Formally, if* $(t_1, z_1) \in Occ$ *and* $(t_2, z_2) \in Occ$*, we have either* $t_1 = t_2$*,* $z_1 \neq z_2$*,* $end_{t_1} \leq start_{t_2}$*, or* $end_{t_2} \leq start_{t_1}$.

- *At each instant, resources are not used beyond their capacity* across all workstations. *Formally, we denote* $\mathit{consumption}(r,\tau)$ *the amount of units of resource* r *used at instant* τ*, and require that at each instant* $\tau \in [0, Ct]$ *and for each resource* $r \in \mathcal{R}$*, we have* $\mathit{consumption}(r,\tau) \leq \mathit{capa}_r$ *where:*

$$\mathit{consumption}(r,\tau) = \sum_{(t,n)\in \mathit{Cons}_r} \sum_{k=1}^{W} \begin{cases} n & \mathit{if\ start}_t \leq \tau + k.Ct \leq \mathit{end}_t \\ 0 & \mathit{otherwise} \end{cases} \tag{1}$$

The last constraint on the consumption of resources across workstations is, to our knowledge, original. At each instant, there is one aircraft in each assembly substage, thus each resource is used simultaneously by each aircraft. Hence, in Equation (1), in the second summation, k ranges across all substages (W is the total number of substages). Moreover, in a stage with multiple workstations, a task may last longer than one Ct. In this case, this task is performed simultaneously on multiple workstations, and contributes to the resource usage multiple times. This is captured by the $\tau + k.Ct$ term of Equation (1), and illustrated in the example that follows.

▶ **Example 4.** Figures 4 and 5 depict two solutions for the problem of Example 2 respectively associated with layout L_1 and layout L_2.

With layout L_1 (Figure 4), the best possible cycle time is 8. Zone occupation and resource usage can be visualized by the intervals depicted in the figure. Since each aircraft instance has its own work zones, each zone is duplicated in each stage and substage. On the opposite, resources are shared between all assembly stages, and are used simultaneously. Thus, the aircraft in stage 1 consumes 2 units of resource 1, at the same time as the aircraft in stage 2 consumes 1 unit of it. The same holds for all stages, resources and tasks. This is illustrated on the right hand side of Figure 4.

In Figure 5, the stage set $S = \{1, 2\}$ contains two stages, however the first stage contains two parallel workstations and therefore two substages. This means we have $w_1 = 2$, and $w_2 = 1$, which allows task t_2 to start at date 2 in substage 1.1 and end at date 8 in substage 1.2 (as tasks cannot span across stages, but can span across substages). In this setting, the smallest cycle time is 6.

A feature specific to our approach is how task t_2 consumes resource 1 "several times" in Figure 5. At each instant, 3 aircraft are being assembled, one in each assembly stage. Let us consider the moment when 3 time units have elapsed in the current cycle and that aircraft A_i, A_{i-1} are in stage 1, and aircraft A_{i-2} in stage 2. A_i has already spent 3 time units in stage 1, meaning that 1 time unit of task t_2 has already been executed for it. Aircraft A_{i-1} is also in stage 1 but has entered it $Ct + 3 = 9$ time units ago and is therefore in substage 1.2. Task t_2 is not yet finished for this aircraft, as 7 time units have already been executed. Consequently, two instances of the task t_2 are simultaneously executed on two different aircraft instances, thus using twice the amount of resource 1, as illustrated on the right. Note that at the same time, task t_6 is executed on aircraft A_{i-1} in stage 2, consuming an additional unit of resource 1.

4.2 Constraint Programming Representation

The MALBWP problem can be represented in CP provided that an upper bound on the pulse time maxCt is added to the problem inputs. The resulting temporal horizon considered, denoted H, is equal to $\sum_{s\in\mathcal{S}} w_s \cdot \mathit{maxCt}$.

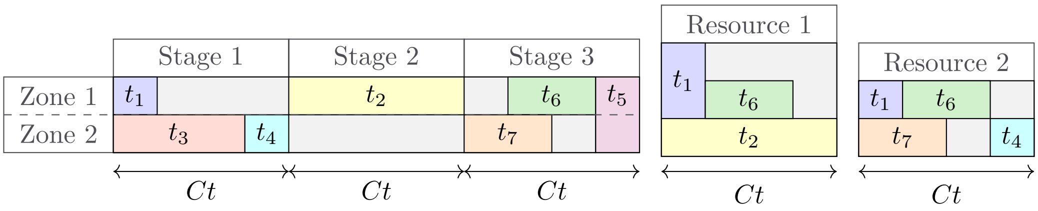


Figure 4 Solution with $Ct = 8$ for the toy problem with layout L_1.

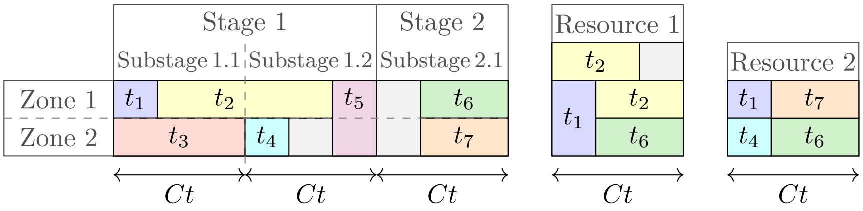


Figure 5 Solution with $Ct = 6$ for the toy problem with layout L_2.

Our representation makes use of the OPL constraint programming language [33], as it makes heavy use of its concept of interval variables and associated constraints, in particular for optional intervals. An interval variable has a range $[min, max]$ that specifies its earliest start date and latest end date, and it can be optional, meaning in that case that it is not necessarily present in the produced schedule. The OPL functions startOf, endOf, and lengthOf return respectively the start date, end date and length of a given interval.

Our constraint model uses the following decision variables. All interval variables have the range $[0, H]$ unless explicitly mentioned.

- The rate is modeled by an integer variable **Ct** in the range $[0, maxCt]$.
- For each task $t \in \mathcal{T}$, the mandatory interval variable $\mathbf{itv}_t$ represents the execution of task t. Its duration is fixed to dur_t.
- For each task $t \in \mathcal{T}$ and each stage $s \in \mathcal{S}$, the optional interval variable $\mathbf{itv}_{t,s}$. If present, this interval has the fixed duration dur_t, and represents the execution of task t in stage s. If absent, it means task t is executed in another stage.
- For each task $t \in \mathcal{T}$, each stage $s \in \mathcal{S}$ and each substage $w \in [1..w_s]$, the optional interval variable $\mathbf{itv}_{t,s,w}$ represents the execution of task t during the w-th substage of stage s. If absent, it means that task t is executed on another stage or substage. For example in Figure 5, task t_1 is executed during the first substage of the first stage. Hence $\mathbf{itv}_{t_1,1,1}$ is present, and both $\mathbf{itv}_{t_1,1,2}$ and $\mathbf{itv}_{t_1,2,1}$ are absent. On the other hand, task t_2 is executed in both substages of stage 1, so $\mathbf{itv}_{t_2,1,1}$ and $\mathbf{itv}_{t_2,1,2}$ are both present and $\mathbf{itv}_{t_2,2,1}$ is absent.
- For each task $t \in \mathcal{T}$, each stage $s \in \mathcal{S}$ and each substage $w \in [1..w_s]$, the optional interval variable $\mathbf{itvRsc}_{t,s,w}$ has a range equal to $[0, maxCt]$. It represents the time during which the execution of time t on the w-th substage of station s uses its resources. If task t is executed on another stage, or not during the w-th substage, this interval is absent. As for the previous set of interval variables, $\mathbf{itvRsc}_{t_1,1,1}$, $\mathbf{itvRsc}_{t_2,1,1}$ and $\mathbf{itvRsc}_{t_2,1,2}$ are present, whereas $\mathbf{itvRsc}_{t_1,2,1}$ and $\mathbf{itvRsc}_{t_2,2,1}$ are absent.

The constraints of our encoding are as follows.

Assembly Line Expressions

We start by defining some expressions for the start and end dates for each stage, and for each pulse cycle inside each stage. Expressions are neither constraints nor decision variables, but are useful for efficiently expressing recurrent patterns in constraints.

$$\forall s \in \mathcal{S}, \qquad \text{stageStart}_s = \sum_{i=1}^{s-1} w_i \cdot \mathbf{Ct} \tag{2}$$

$$\forall s \in \mathcal{S}, \qquad \text{stageEnd}_s = \sum_{i=1}^{s} w_i \cdot \mathbf{Ct} \tag{3}$$

$$\forall s \in \mathcal{S}, \forall w \in [1..w_s], \qquad \text{subStageStart}_{s,w} = \text{stageStart}_s + (w-1) \cdot \mathbf{Ct} \tag{4}$$

$$\forall s \in \mathcal{S}, \forall w \in [1..w_s], \qquad \text{subStageEnd}_{s,w} = \text{stageStart}_s + w \cdot \mathbf{Ct} \tag{5}$$

Equations (2) and (3) define expressions for the start and end dates of each stage. These dates directly depend on the value of **Ct**. Similarly, Equations (4) and (5) define expressions for the start and end date of every substage.

Note that $\text{stageStart}_{\mathit{1}} = 0$, which means the first stage starts at time 0. Moreover, $\text{stageStart}_{s+\mathit{1}} = \text{stageEnd}_s$ for all stages except for the last one, which means each stage starts right when its predecessor ends. Finally, a stage with w_s workstations has a duration of $w_s \cdot Ct$, and each substage has a duration of exactly one Ct.

Stage and Zone Constraints

The stage constraints express that each task can only take place inside one stage. Furthermore, two tasks that occupy the same zone cannot be executed at the same time in the same stage. This set of constraints features the alternative OPL keyword, that accepts one interval a and one set of optional intervals B as arguments, and has the following semantics. If a is absent, then all intervals in B are absent. If a is present, then exactly one interval is B is present, and it has the same start and end dates as a.

$$\forall (t,t') \in \mathit{Prec}, \qquad \mathsf{endBeforeStart}(\mathbf{itv}_t, \mathbf{itv}_{t'}) \tag{6}$$

$$\forall z \in \mathcal{Z} \qquad \mathsf{noOverlap}\big(\{\mathbf{itv}_t | (z,t) \in \mathit{Occ}\}\big) \tag{7}$$

$$\forall t \in \mathcal{T}, \qquad \mathsf{alternative}\Big(\mathbf{itv}_t, \big\{\mathbf{itv}_{t,s} | s \in [1..nS]\big\}\Big) \tag{8}$$

$$\forall t \in \mathcal{T}, \forall s \in \mathcal{S}, \qquad \text{stageStart}_s \leq \mathsf{startOf}(\mathbf{itv}_{t,s}, H) \tag{9}$$

$$\forall t \in \mathcal{T}, \forall s \in \mathcal{S}, \qquad \mathsf{endOf}(\mathbf{itv}_{t,s}, 0) \leq \text{stageEnd}_s \tag{10}$$

Constraints (6) expresses that two tasks with a precedence constraint must be scheduled in the proper order. Constraints (7) forbid two tasks that occupy the same zone to be executed simultaneously in the same stage. Constraints (8) force interval $\mathbf{itv}_t$ to equal exactly one of the $\mathbf{itv}_{t,s}$. Constraints (9) and (10) ensure that each interval $\mathbf{itv}_{t,s}$, if present, takes place during stage s. The second arguments to the startOf and endOf functions (respectively H and 0) are default values in case the interval is absent. In conjunction to Constraints (8), this enforces every task to be executed inside one stage.

Substage Constraints

We consider here constraints associated with the execution of tasks in substages.

$$\forall t \in \mathcal{T}, \forall s \in [1..nS], \qquad \mathsf{span}\Big(\mathbf{itv}_{t,s}, \big\{\mathbf{itv}_{t,s,w} | w \in [1..w_s]\big\}\Big) \tag{11}$$

$$\forall t \in \mathcal{T}, \forall s \in \mathcal{S}, \forall w \in w_s, \qquad \text{subStageStart}_{s,w} \leq \mathsf{startOf}(\mathbf{itv}_{t,s,w}, H) \tag{12}$$

$$\forall t \in \mathcal{T}, \forall s \in \mathcal{S}, \forall w \in w_s, \qquad \mathsf{endOf}(\mathbf{itv}_{t,s,w}, 0) \leq \text{subStageEnd}_{s,w} \tag{13}$$

$$\forall t \in \mathcal{T}, \forall s \in \mathcal{S}, \qquad \mathsf{lenghtOf}(\mathbf{itv}_{t,s}) = \sum_{w \in [1..w_s]} \mathsf{lengthOf}(\mathbf{itv}_{t,s,w}) \tag{14}$$

Constraints (11) uses the span OPL constraint, that accepts an interval a and a set of optional intervals B as arguments, and has the following semantics. If a is absent, then all intervals in B are absent. If a is present, then at least one interval in B is present and the start (resp. end) date of a equals the earliest start date (resp. latest end date) of all the present intervals in B.

Constraints (12) and (13) ensure that each interval $\mathbf{itv}_{t,s,w}$, if present, lies within the w-th substage. As a consequence, for each task t and stage s, two intervals $\mathbf{itv}_{t,s,w}$ and $\mathbf{itv}_{t,s,w'}$ cannot overlap, unless $w = w'$.

Constraints (14) ensure that the cumulated length of all present $\mathbf{itv}_{t,s,w}$ intervals equals the length of their spanning $\mathbf{itv}_{t,s}$ interval. Since they cannot overlap because of Constraints (12) and (13), this ensures that the intervals $\mathbf{itv}_{t,s,w}$ entirely cover the interval $\mathbf{itv}_{t,s}$ without any gaps, for each task t and stage s.

Resource Constraints

As illustrated in Example 2, resource usage is a pattern that is repeated every cycle. In order to model resource usage across workstations with interval variables, we rewrite the consumption of resources of Equation (1) as a consumption elements set. Formally, for each resource r, we consider the set $\mathcal{C}_r$ of consumption elements $\{(\sigma, \tau, n)\}$ where σ is the start date of the consumption, τ is its end date and n the number of units it consumes. $\mathcal{C}_r$ is built following Algorithm 1, where % is the integer remainder.

Algorithm 1 Computation of $\mathcal{C}_r$ for each resource r.

```
 1: function $computeCr(r)$
 2:     $\mathcal{C}_r \leftarrow \emptyset$
 3:     for $(t, n) \in Cons_r$ do
 4:         if $(dur_t \leq Ct) \wedge (start_t \,\%\, Ct + dur_t \leq Ct)$ then
 5:             $\mathcal{C}_r \leftarrow \mathcal{C}_r \cup \{(start_t \,\%\, Ct, end_t \,\%\, Ct, n)\}$
 6:         else
 7:             $before \leftarrow Ct - start_t \,\%\, Ct$
 8:             $after \leftarrow end_t \,\%\, Ct$
 9:             $q \leftarrow \lfloor \frac{dur_t - before - after}{Ct} \rfloor$
10:             $\mathcal{C}_r \leftarrow \mathcal{C}_r \cup \{(start_t \,\%\, Ct, Ct, n), (0, end_t \,\%\, Ct, n), (0, Ct, n \cdot q)\}$
        return $\mathcal{C}_r$
```

$\mathcal{C}_r$ is initialized to the emptyset. Then, for each tuple $(t, n) \in Cons_r$, we add one or several elements to $\mathcal{C}_r$. If the task is contained in a unique substage (Line 4), then we shift the task so that it is contained in the interval $[0, Ct]$. Otherwise, we consider the duration of the task in its first substage (Line 7) and in its last substage (Line 8). Then, the quotient

q (Line 9) denotes the number of substages over which the task completely spans. We finally add consumptions corresponding to the first substage, to the last and all the included substages, and shift them so that they belong to the interval $[0, Ct]$.

The OPL encoding uses the $\mathbf{itvRsc}_{t,s,w}$ intervals to represent the tuples from the $\mathcal{C}_r$ set for each resource. The integer remainder operator can be eluded thanks to the $\mathbf{itv}_{t,s,w}$ intervals.

$$\forall t \in \mathcal{T}, \forall s \in \mathcal{S}, \forall w \in w_s, \qquad \mathsf{endOf}(\mathbf{itvRsc}_{t,s,w}, 0) \leq \mathbf{Ct} \tag{15}$$

$$\forall t \in \mathcal{T}, \forall s \in \mathcal{S}, \forall w \in w_s, \qquad \mathsf{presenceOf}(\mathbf{itvRsc}_{t,s,w}) = \mathsf{presenceOf}(\mathbf{itv}_{t,s,w}) \tag{16}$$

$$\forall t \in \mathcal{T}, \forall s \in \mathcal{S}, \forall w \in w_s, \qquad \mathsf{startAtStart}(\mathbf{itvRsc}_{t,s,w}, \mathbf{itv}_{t,s,w}, \mathrm{subStageStart}_{s,w}) \tag{17}$$

$$\forall t \in \mathcal{T}, \forall s \in \mathcal{S}, \forall w \in w_s, \qquad \mathsf{endAtEnd}(\mathbf{itvRsc}_{t,s,w}, \mathbf{itv}_{t,s,w}, \mathrm{subStageStart}_{s,w}) \tag{18}$$

$$\forall r \in \mathcal{R}, \qquad \sum_{(t,n) \in Cons_r} \mathsf{pulse}(\mathbf{itvRsc}_{t,s,w}, n) \leq capa_r \tag{19}$$

Constraints (15) ensure that resource usage intervals fall into the $[0, \mathbf{Ct}]$ time frame, since they represent the use of each resource at each substages. Constraints (16) impose that the only resource usage intervals that are present are those that correspond to a present substage interval for this task. Constraints (17) and (18) use the startAtStart and endAtEnd OPL constraints with delay. When both intervals are present they are respective shorthands for $\mathsf{startOf}(\mathbf{itvRsc}_{t,s,w}) + \mathrm{subStageStart}_{s,w} = \mathsf{startOf}(\mathbf{itv}_{t,s,w})$, and $\mathsf{endOf}(\mathbf{itvRsc}_{t,s,w}) + \mathrm{subStageStart}_{s,w} = \mathsf{endOf}(\mathbf{itv}_{t,s,w})$, and do nothing if at least one interval is absent.

The final constraints (19) use the pulse OPL function, that represents a cumulative function. The pulse primitive accepts an interval a and an integer value h, and describes the function with value h in interval a (if present), and 0 elsewhere. Constraints (19) ensure that the resources consumed by all tasks at some time during the range $[0, Ct]$ do not exceed the resource capacity.

Figure 6 illustrates the relationship between all interval variables used in the CP model. Absent intervals are grayed, mandatory intervals have a thick border. A dotted line between two intervals indicates that they are connected by the constraints indicated on the right. This figure represents a $[3, 1]$ layout, i.e. a layout with 3 workstations in the first stage and one in the second stage, and a single task t that uses n units of a resource r with capacity $3n$. Interval $\mathbf{itv}_t$ ranges across the whole planning horizon. Each interval $\mathbf{itv}_{t,s}$ ranges across stage s. Each interval $\mathbf{itv}_{t,s,w}$ ranges across substage s, w. Intervals $\mathbf{itvRsc}_{t,s,w}$ all range across $[0, \mathbf{Ct}]$.

5 Experimentation

This section presents experimental results for the constraint model presented in this paper. Experiments were all run using IBM CP Optimizer 20.1.0 through the Java API on a 20-core Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz with 62GB RAM. We first present results obtained on PSPLib-based benchmarks, then results obtained on two real assembly line datasets.

5.1 Results for PSPLib-based Benchmarks

Our model can be seen as an extension of a classical RCPSP problem $(\mathcal{A}, \mathcal{R}es, \mathcal{C}ons, \mathcal{P}rec)$, where $\mathcal{A}$ is the set of activities, $\mathcal{R}es$ the set of resources, $\mathcal{C}ons$ the consumption relationship and $\mathcal{P}rec$ the precedence relationship. Given a set of stages $\{1..nS\}$ and their associated

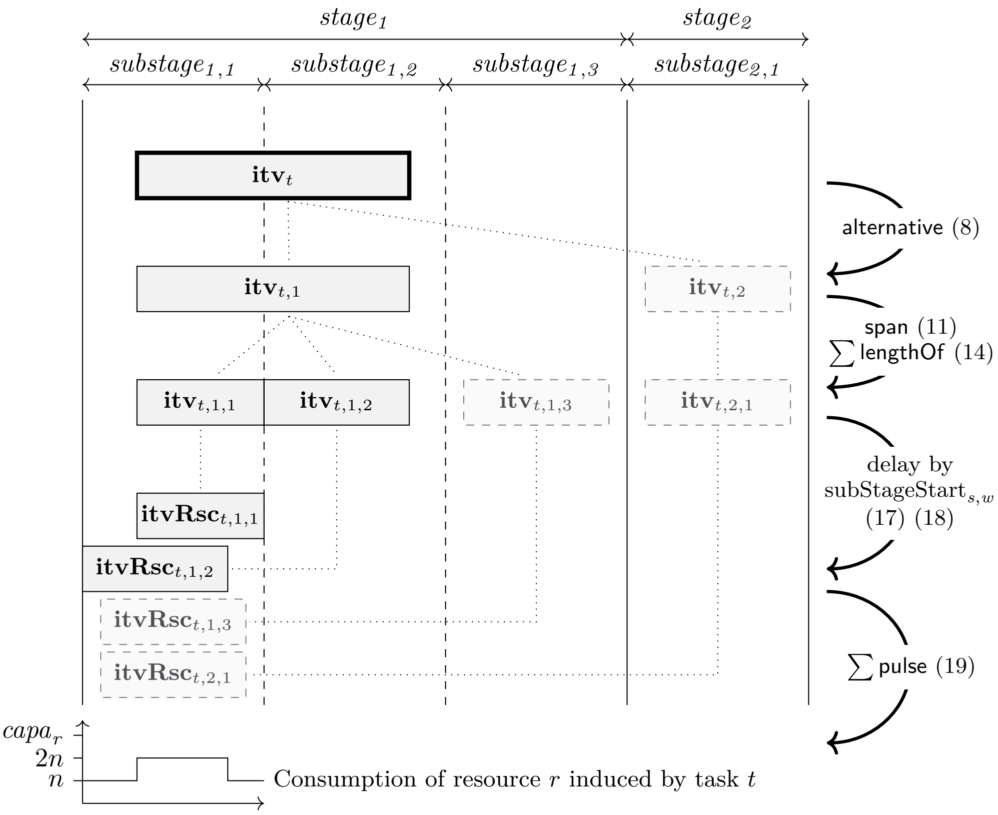


Figure 6 Illustration of the interval variables hierarchy in the CP model.

number of workstations, and a RCPSP problem, we consider the associated MMALBWP $(\mathcal{A}, nS, \emptyset, \mathcal{R}es, \mathcal{P}rec, \emptyset, \mathcal{C}ons)$ in which the set of zones is empty. The resulting dataset will be made public upon acceptance of the paper.

We adapted problems from the PSPLib [19], converting them to MMALBWP by adding a stage size specification, and used a time limit of 5 minutes. The results on the 150 first problems of the `j30rcp` benchmark, detailed in appendix A, demonstrate several results summarized in Table 1a, and in graph in Table 1b that counts the number of instances where some layouts *strictly* improve other layouts. Below are additional considerations.

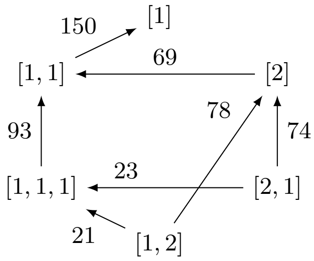
- With one workstation, the cycle time equals the RCPSP makespan.
- More workstations always means a better or equal cycle time (in sequence or in parallel).
- Adding a second workstation, in sequence (layout $[1, 1]$) or in parallel (layout $[2]$), always strictly improves the cycle time of layout $[1]$.
- Adding a third workstation further improves the cycle time in 62% of sequential instances (93/150, $[1, 1, 1]$ compared to $[1, 1]$), and in 51% of parallel instances ($(74 + 78)/300$, $[2, 1]$ and $[1, 2]$ compared to $[2]$).
- At constant number of workstations, merging two stages yields a better cycle time in 25% of instances ($(69 + 23 + 21)/450$, $[2]$ compared to $[1, 1]$ plus $[2, 1]$ and $[1, 2]$ compared to $[1, 1, 1]$). This lower number is explained by the fact that resources become a limiting factor.
- Adding stages makes the problem more difficult for the solver, especially with parallel workstations.

Table 1 Statistics of the solutions of the first 150 instances of the `j30rcp` PSPLIB benchmark.

(a) Results summary.

Layout	#Solved	#Optimal	Avg. Time (s)
$[1]$	150	150	2.5
$[1, 1]$	150	122	72.3
$[2]$	134	99	124.1
$[1, 1, 1]$	150	84	151.1
$[2, 1]$	149	59	193.9
$[1, 2]$	150	64	184.5

(b) Number of instances in which a layout yields a strictly better cycle time than another.



5.2 Industrial Assembly Line

This work was initially motivated by the final assembly line of a manufacturer of small aircraft, who already uses a parallel workstation.

The assembly line model contains 51 tasks, 93 precedence constraints, 12 work zones, and 23 resources. Tasks use between 1 and 4 resource units. We have tested two factory layouts: the first one has 5 sequential stages of one workstation each ($[1, 1, 1, 1, 1]$); the second layout duplicates the first workstation in parallel ($[2, 1, 1, 1]$). We solved the model with a 30 minutes timeout. The solver finds its best solution in a few seconds, but fails to prove that the solution is optimal in the rest of its computation time. The results, and in general the entire modeling work, helped analyse and consolidate the different factory configurations. Note that due to confidentiality issues, we do not provide more information about this dataset nor the obtained results.

5.3 Assembly Line Design Use Case

We tested our approach on the assembly line preliminary design problem originally dataset [28] presented in [27] and adapted for the 2023 XCSP3 competition [6]. In the latter, the assembly line is composed of 4 stages with one workstation each. The objective is to minimize the number of operators in the line. We adapt these benchmarks by fixing the number of available operators, ignoring neutralization constraints, and optimizing the cycle time when considering several factory layouts. We have considered 7 layouts for each of the three instances, resulting in a dataset of 21 instances available online [29].

These instances are larger and much more challenging than the ones presented before. We gave a 10mn time limit to the solver, it only managed to find a solution for 15 instances out of 21, and it still failed to prove the optimality of all instances, as reported in Table 2. Furthermore, in instance 3, the cycle time found for stage sizes $[1, 3]$ (i.e. two stages of respectively 3 and 1 workstations) is higher than the one found for stage sizes $[1, 2, 1]$, which indicates that the solver failed to find the same solution within its time limit.

6 Conclusion

This paper presents a Constraint Programming model for a multi-manned assembly line balancing problem with walking workers and parallel stations. This model can be seen as an extension of multi-manned ALP with walking workers, and of RCPSP with parallel workstations. It provides an efficient way to evaluate how the number of workstations and

Table 2 Production rates associated to each dataset and various layouts. Empty cells indicate that the 10 minutes timeout elapsed before a solution was found.

		Layout						
Instance	Nb. tasks	$[1,1,1,1]$	$[2,1,1]$	$[1,2,1]$	$[1,1,2]$	$[3,1]$	$[1,3]$	$[4]$
Instance 1	178	1260	1260	1260	1260	1024	900	855
Instance 2	178	1260	–	1260	1260	–	954	–
Instance 3	628	1200	–	900	1172	–	1160	–

their flow can help design a factory, by evaluating the production rates that it can attain. We detailed a constraint programming model in the OPL language that makes extensive use of interval variables, and validated the model by reproducing classical RCPSP benchmark results. We also used it to study assembly line implementations for industrial use cases.

There are several directions for pursuing this work. Better heuristics would probably improve the solver performance on the PSPLIB-based instances. Moreover, it would be interesting to address both workstation layout and resource sizing in a multi-objective approach. This would pave the way for the design of assembly lines with different configurations for low rate with low resource consumption, and high rates with high resource consumption.

References

1 Hacı Mehmet Alakaş. General resource-constrained assembly line balancing problem: conjunction normal form based constraint programming models. *Soft Computing*, 25(8):6101–6111, 2021.

2 Hacı Mehmet Alakaş, Mehmet Pınarbaşı, and Mustafa Yüzükırmızı. Constraint programming model for resource-constrained assembly line balancing problem. *Soft Computing*, 24:5367–5375, 2020.

3 Eduardo Álvarez-Miranda, Sebastián Chace, and Jordi Pereira. Assembly line balancing with parallel workstations. *International Journal of Production Research*, 59(21):6486–6506, 2021.

4 Felipe FB Araújo, Alysson M Costa, and Cristóbal Miralles. Two extensions for the alwabp: Parallel stations and collaborative approach. *International Journal of Production Economics*, 140(1):483–495, 2012.

5 Dmitry Arkhipov, Olga Battaïa, Julien Cegarra, and Alexander Lazarev. Operator assignment problem in aircraft assembly lines: a new planning approach taking into account economic and ergonomic constraints. *Procedia CIRP*, 76:63–66, 2018.

6 Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2023 XCSP3 competition. *CoRR*, abs/2312.05877, 2023. doi:10.48550/arXiv.2312.05877.

7 Zhongkai Bao, Lu Chen, and Kejun Qiu. An aircraft final assembly line balancing problem considering resource constraints and parallel task scheduling. *Computers & Industrial Engineering*, 182:109436, 2023.

8 Olga Battaïa and Alexandre Dolgui. Hybridizations in line balancing problems: A comprehensive review on new trends and formulations. *International Journal of Production Economics*, 250:108673, 2022.

9 Christian Becker and Armin Scholl. Balancing assembly lines with variable parallel workplaces: Problem definition and effective solution procedure. *European Journal of Operational Research*, 199(2):359–374, 2009.

10 Alexander Biele and Lars Mönch. Hybrid approaches to optimize mixed-model assembly lines in low-volume manufacturing. *Journal of Heuristics*, 24(1):49–81, 2018.

11 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.

12 Joseph Bukchin and Jacob Rubinovitz. A weighted approach for assembly line design with station paralleling and equipment selection. *IIE transactions*, 35(1):73–85, 2003.

13 Yin-Yann Chen, Chen-Yang Cheng, and Jia-Ying Li. Resource-constrained assembly line balancing problems with multi-manned workstations. *Journal of Manufacturing Systems*, 48:107–119, 2018.

14 Zeynel Abidin Çil and Damla Kizilay. Constraint programming model for multi-manned assembly line balancing problem. *Computers & Operations Research*, 124:105069, 2020.

15 Hongyan Ding, Cunbo Zhuang, and Jianhua Liu. Extensions of the resource-constrained project scheduling problem. *Automation in Construction*, 153:104958, 2023.

16 Yunus Ege, Meral Azizoglu, and Nur E Ozdemirel. Assembly line balancing with station paralleling. *Computers & Industrial Engineering*, 57(4):1218–1225, 2009.

17 Yagmur S Gök, Daniel Guimarans, Peter J Stuckey, Maurizio Tomasella, and Cemalettin Ozturk. Robust resource planning for aircraft ground operations. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21–24, 2020, Proceedings 17*, pages 222–238. Springer, 2020.

18 Sönke Hartmann and Dirk Briskorn. An updated survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of operational research*, 297(1):1–14, 2022.

19 Rainer Kolisch and Sönke Hartmann. Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European journal of operational research*, 174(1):23–37, 2006.

20 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.

21 Thiago Cantos Lopes, Adalberto Sato Michels, Celso Gustavo Stall Sikora, and Leandro Magatão. Balancing and cyclical scheduling of asynchronous mixed-model assembly lines with parallel stations. *Journal of Manufacturing Systems*, 50:193–200, 2019.

22 Thiago Cantos Lopes, Giuliano Vidal Pastre, Adalberto Sato Michels, and Leandro Magatão. Flexible multi-manned assembly line balancing problem: Model, heuristic procedure, and lower bounds for line length minimization. *Omega*, 95:102063, 2020.

23 Bahman Naderi, Ahmed Azab, and Katayoun Borooshan. A realistic multi-manned five-sided mixed-model assembly line balancing and scheduling problem with moving workers and limited workspace. *International Journal of Production Research*, 57(3):643–661, 2019.

24 Cemalettin Öztürk, Semra Tunalı, Brahim Hnich, and Arslan Örnek. Balancing and scheduling of flexible mixed model assembly lines with parallel stations. *The International Journal of Advanced Manufacturing Technology*, 67:2577–2591, 2013.

25 Cemalettin Öztürk, Semra Tunalı, Brahim Hnich, and Arslan Örnek. Cyclic scheduling of flexible mixed model assembly lines with parallel stations. *Journal of Manufacturing Systems*, 36:147–158, 2015.

26 Cédric Pralet, Stéphanie Roussel, Thomas Polacsek, François Bouissière, Claude Cuiller, Pierre-Eric Dereux, Stéphane Kersuzan, and Marc Lelay. A scheduling tool for bridging the gap between aircraft design and aircraft manufacturing. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 28, pages 347–355, 2018.

27 Stéphanie Roussel, Thomas Polacsek, and Anouck Chan. Assembly Line Preliminary Design Optimization for an Aircraft. In *CP 2023 (The 29th International Conference on Principles and Practice of Constraint Programming)*, Toronto, Canada, August 2023. doi:10.4230/LIPIcs.CP.2023.32.

28 Stéphanie Roussel. Dataset for the Assembly Line Preliminary Design Optimization Problem, 2024. doi:10.57745/IQLQ7A.

29 Stéphanie Roussel and Xavier Pucel. Dataset for the Constraint Programming Model for Assembly Line Balancing and Scheduling with Walking Workers and Parallel Stations, 2024. doi:10.57745/EWXS9O.

30 Murat Şahin and Talip Kellegöz. Balancing multi-manned assembly lines with walking workers: problem definition, mathematical formulation, and an electromagnetic field optimisation algorithm. *International Journal of Production Research*, 57(20):6487–6505, 2019.

31 Tamara Borreguero Sanchidrián, Tom Portoleau, Christian Artigues, Alvaro García Sánchez, Miguel Ortega Mier, and Pierre Lopez. Exact and heuristic methods for an aeronautical assembly line time-constrained scheduling problem with multiple modes and a resource leveling objective. working paper or preprint, September 2021.

32 Celso Gustavo Stall Sikora, Thiago Cantos Lopes, and Leandro Magatão. Traveling worker assembly line (re) balancing problem: Model, reduction techniques, and real case studies. *European Journal of Operational Research*, 259(3):949–971, 2017.

33 Pascal Van Hentenryck, Laurent Michel, Laurent Perron, and J-C Régin. Constraint programming in opl. In *International Conference on Principles and Practice of Declarative Programming*, pages 98–116. Springer, 1999.

A PSPLib Benchmarks Results

This appendix presents the results of execution of our solver on the first 150 instances of the PSPLib benchmark `j30rcp`. Table 3 is read as follows: Parameter value p and Instance value i describe the problem found in the file named J30p_i.RCP in the archive named `j30rcp.zip` found in the PSPlib [19] accessible at url `https://www.om-db.wi.tum.de/psplib/getdata_sm.html`. Each column in the "Cycle time" and "Computation time" groups corresponds to a different layout. A cycle time of −1 indicates that the solver failed to find a suitable schedule in the allocated time of 5 minutes. A computation time of 300 seconds (5 minutes) indicates that the solver used all its computation time. This means that the corresponding cycle time (if any) may be sub-optimal.

Table 3 Detailed Results of the 150 first instances of the `j30rcp` benchmark for various number of stages and workstations.

Parameter	Instance	Cycle time						Computation time (seconds)					
		[1]	[1, 1]	[2]	[1, 1, 1]	[2, 1]	[1, 2]	[1]	[1, 1]	[2]	[1, 1, 1]	[2, 1]	[1, 2]
1	1	43	29	29	29	29	29	0.25	1.17	11.75	8.25	77.54	14.15
1	2	47	33	33	33	33	33	0.32	0.72	9.15	3.02	26.13	3.27
1	3	47	26	26	23	23	23	0.06	0.91	11.18	2.21	33.70	3.50
1	4	62	41	41	41	41	41	0.06	1.61	8.42	16.51	30.81	32.36
1	5	39	34	34	34	34	34	0.32	3.82	22.92	33.93	300	119.20
1	6	48	32	32	32	32	32	0.34	0.92	14.08	4.29	38.52	9.59
1	7	60	35	35	35	35	35	0.05	1.11	9.12	4.79	15.42	8.33
1	8	53	33	33	33	33	33	0.06	0.80	5.14	2.79	3.13	2.77
1	9	49	31	31	31	31	31	0.25	1.83	12.68	16.64	115.54	38.57
1	10	45	29	29	29	29	29	0.05	2.37	13.58	4.60	40.59	7.95
2	1	38	26	26	21	21	21	0.05	2.12	13.53	5.94	118.13	102.28
2	2	51	36	36	36	36	36	0.05	1.19	12.30	2.26	23.37	3.69
2	3	43	29	29	29	29	29	0.06	1.00	10.71	2.14	22.25	3.40
2	4	43	23	23	19	19	19	0.05	0.70	9.11	2.33	13.63	2.06
2	5	51	33	33	27	27	27	0.05	0.44	10.23	1.89	2.56	1.93

2	6	47	29	29	22	22	22	0.05	0.95	5.74	1.48	2.46	3.38
2	7	47	29	26	25	25	25	0.06	0.80	10.44	2.98	7.13	4.91
2	8	54	33	32	29	29	29	0.05	1.16	9.35	2.11	21.70	2.43
2	9	54	32	32	30	30	30	0.05	1.26	10.48	10.77	300	300
2	10	43	25	24	22	22	22	0.05	0.85	10.80	1.70	20.23	6.91
3	1	72	39	39	30	29	30	0.06	0.23	8.75	0.81	1.06	2.92
3	2	40	23	20	20	18	19	0.05	0.49	13.45	2.92	1.50	2.59
3	3	57	32	32	24	23	24	0.06	0.37	7.59	0.71	1.26	2.22
3	4	98	62	62	39	39	39	0.06	1.20	2.03	1.27	2.23	1.51
3	5	53	28	28	28	28	28	0.06	1.28	10.12	1.91	23.03	4.60
3	6	54	33	28	24	24	24	0.05	0.81	1.34	5.80	3.42	1.57
3	7	48	24	24	20	18	19	0.05	0.26	1.48	0.96	2.17	2.12
3	8	54	29	27	25	22	22	0.05	0.25	9.22	0.97	1.27	2.27
3	9	65	35	34	31	31	31	0.11	1.58	12.45	2.59	29.00	3.03
3	10	59	30	30	30	30	30	0.06	0.68	9.84	1.69	22.66	3.52
4	1	49	28	25	19	19	18	0.06	0.23	0.48	1.01	1.29	1.34
4	2	60	36	36	28	28	28	0.10	0.39	0.89	0.94	1.91	2.09
4	3	47	28	25	22	20	21	0.06	0.56	1.04	1.31	2.35	1.29
4	4	57	33	32	21	20	21	0.05	0.22	0.80	0.81	1.03	1.36
4	5	59	34	32	24	24	24	0.06	0.61	1.00	2.36	3.02	1.99
4	6	45	26	23	21	21	21	0.06	0.32	0.65	1.32	2.66	8.89
4	7	56	29	28	24	23	23	0.06	0.23	0.79	1.47	1.67	1.74
4	8	55	30	28	21	20	20	0.06	0.26	0.59	0.51	0.47	0.77
4	9	38	22	22	20	20	20	0.07	0.69	0.88	1.36	2.10	2.05
4	10	48	26	25	24	24	24	0.06	0.46	0.73	1.35	2.28	2.34
5	1	53	37	37	34	35	34	0.25	6.59	59.39	95.36	300	269.00
5	2	82	56	56	56	56	56	0.87	6.45	39.28	129.70	300	300
5	3	76	57	57	56	56	56	0.52	6.67	86.05	300	300	300
5	4	63	52	52	52	52	52	1.16	25.59	236.89	233.18	300	300
5	5	76	59	58	58	58	58	0.58	7.45	97.50	197.38	300	300
5	6	64	46	46	44	44	44	0.35	5.98	134.79	47.18	235.48	165.77
5	7	76	72	73	72	72	72	1.07	133.64	300	300	300	300
5	8	67	54	54	51	59	56	0.91	20.42	201.24	105.83	300	300
5	9	49	37	36	35	36	36	0.37	7.11	75.86	184.16	300	300
5	10	70	55	54	52	53	53	0.63	11.10	99.64	126.15	300	300
6	1	59	42	42	42	42	42	0.29	11.41	95.94	300	300	300
6	2	51	36	36	35	34	34	0.13	1.85	13.04	26.98	283.05	74.27
6	3	48	31	31	30	30	30	0.08	1.83	15.05	78.84	300	300
6	4	42	33	33	32	33	32	0.50	11.12	300	300	300	300
6	5	67	51	51	48	50	48	0.26	7.67	74.17	93.55	300	300
6	6	37	26	25	24	24	24	0.05	3.08	17.31	21.34	300	251.14
6	7	46	30	30	29	30	29	0.05	1.95	15.39	16.08	300	227.00
6	8	39	30	30	30	30	31	0.05	3.78	123.51	300	300	300
6	9	51	35	35	35	35	35	0.06	1.61	8.64	7.15	60.12	10.62
6	10	61	44	43	43	43	45	0.37	16.67	201.23	300	300	300
7	1	55	29	28	25	25	25	0.06	1.63	11.28	2.06	6.03	4.25
7	2	42	28	28	27	27	27	0.05	1.86	12.51	6.13	11.71	20.75
7	3	42	28	27	26	26	26	0.06	2.66	6.12	4.49	61.24	11.89

7	4	44	30	29	25	25	25	0.05	1.83	12.79	8.00	72.16	33.25
7	5	44	31	30	30	30	30	0.09	3.07	40.28	144.03	300	300
7	6	35	22	20	19	18	18	0.06	0.92	14.24	4.04	12.15	23.66
7	7	50	33	32	30	29	30	0.06	1.80	11.25	5.07	16.05	19.83
7	8	44	35	34	33	34	33	0.06	2.10	16.16	26.62	300	155.17
7	9	60	33	33	31	31	30	0.05	1.02	13.21	2.76	22.47	3.02
7	10	49	33	31	29	29	29	0.26	1.52	11.14	3.23	30.24	14.02
8	1	44	26	25	23	23	23	0.06	1.15	1.03	2.40	9.16	12.08
8	2	51	30	26	21	21	21	0.07	0.21	0.64	1.49	3.37	2.64
8	3	53	29	27	25	25	25	0.05	0.56	1.07	1.27	2.59	2.74
8	4	48	26	24	22	21	21	0.06	0.43	0.98	1.33	2.06	3.00
8	5	58	32	32	30	30	30	0.06	1.03	1.05	4.50	7.36	17.72
8	6	47	27	26	25	24	24	0.06	0.83	1.21	21.96	49.80	51.37
8	7	41	23	21	18	18	18	0.07	0.36	0.08	1.36	3.93	3.60
8	8	51	30	28	25	25	25	0.06	1.53	1.49	3.28	6.08	6.51
8	9	39	22	20	19	19	19	0.16	0.54	0.77	5.67	173.10	74.25
8	10	67	36	34	25	25	25	0.06	0.09	0.29	1.24	3.89	2.06
9	1	83	75	-1	75	80	75	3.57	300	300	300	300	300
9	2	92	91	-1	89	91	89	50.50	300	300	300	300	300
9	3	68	64	60	56	60	59	1.18	300	300	300	300	300
9	4	71	63	64	62	66	69	1.48	128.39	300	300	300	300
9	5	70	58	58	57	63	61	0.75	32.52	191.66	300	300	300
9	6	59	48	-1	51	50	50	1.16	211.58	300	300	300	300
9	7	63	52	-1	53	55	56	1.74	198.00	300	300	300	300
9	8	91	79	-1	79	82	81	1.46	300	300	300	300	300
9	9	63	52	52	52	61	52	2.58	300	300	300	300	300
9	10	88	79	80	76	76	80	3.53	300	300	300	300	300
10	1	42	31	31	30	31	31	0.06	6.13	47.72	208.58	300	300
10	2	56	43	-1	43	44	44	0.42	35.87	300	300	300	300
10	3	62	48	51	49	48	49	0.38	213.37	300	300	300	300
10	4	58	44	44	44	45	44	0.28	17.68	300	300	300	300
10	5	41	35	34	34	35	34	0.11	300	300	300	300	300
10	6	44	34	34	34	35	34	0.36	28.65	300	300	300	300
10	7	49	34	34	33	34	33	0.07	41.67	300	300	300	300
10	8	54	41	-1	40	41	42	0.38	45.85	300	300	300	300
10	9	49	31	32	31	31	31	0.05	26.49	300	300	300	300
10	10	41	31	31	30	31	31	0.31	15.29	300	300	300	300
11	1	54	44	44	44	44	44	0.08	51.93	300	300	300	300
11	2	56	43	42	42	43	42	0.13	10.68	163.90	300	300	300
11	3	81	43	41	37	38	38	0.06	0.95	15.16	142.67	300	300
11	4	63	42	41	40	41	41	0.05	4.83	55.68	300	300	300
11	5	49	40	40	40	40	40	0.34	79.96	300	300	300	300
11	6	44	30	30	29	30	29	0.06	14.26	246.49	300	300	300
11	7	36	27	26	26	27	27	0.06	10.49	300	300	300	300
11	8	62	44	43	44	43	45	0.07	17.88	281.82	300	300	300
11	9	67	41	41	40	41	41	0.06	3.70	27.53	300	300	300
11	10	38	27	27	26	26	27	0.06	3.00	19.09	174.05	300	300
12	1	47	29	28	26	26	26	0.15	1.76	2.51	73.81	300	300

12	2	46	30	30	30	30	30	0.06	2.15	45.26	300	300	300
12	3	37	23	23	22	22	22	0.06	2.17	14.73	74.48	300	300
12	4	63	35	32	29	29	29	0.08	0.29	1.31	50.03	300	300
12	5	47	24	24	21	21	21	0.07	0.29	0.89	35.54	300	300
12	6	53	31	31	29	30	29	0.06	2.76	10.51	264.60	300	300
12	7	55	30	28	27	27	27	0.07	0.75	2.29	300	300	300
12	8	35	19	18	18	18	18	0.07	0.77	0.72	5.97	300	300
12	9	52	30	29	28	28	29	0.06	1.66	8.45	300	300	300
12	10	57	32	29	26	26	26	0.07	0.24	0.95	60.81	300	300
13	1	58	55	-1	56	56	55	12.99	300	300	300	300	300
13	2	62	61	-1	61	61	60	121.27	300	300	300	300	300
13	3	76	73	-1	73	73	73	16.92	300	300	300	300	300
13	4	72	64	-1	64	67	64	4.85	300	300	300	300	300
13	5	67	65	-1	65	65	65	30.07	300	300	300	300	300
13	6	64	60	64	60	61	60	34.34	300	300	300	300	300
13	7	77	76	-1	76	-1	75	13.47	300	300	300	300	300
13	8	106	102	-1	102	109	97	47.01	300	300	300	300	300
13	9	71	65	-1	65	69	67	1.85	300	300	300	300	300
13	10	64	56	56	56	55	56	3.98	300	300	300	300	300
14	1	50	40	41	41	40	40	0.60	236.77	300	300	300	300
14	2	53	49	-1	49	49	49	1.03	300	300	300	300	300
14	3	58	52	52	50	51	52	0.36	300	300	300	300	300
14	4	50	42	42	41	43	41	0.64	300	300	300	300	300
14	5	52	36	37	37	37	36	0.10	121.81	300	300	300	300
14	6	35	30	30	29	29	29	0.07	300	300	300	300	300
14	7	50	46	45	44	44	46	1.01	300	300	300	300	300
14	8	54	42	42	42	41	42	0.06	300	300	300	300	300
14	9	46	39	40	39	40	41	0.83	258.98	300	300	300	300
14	10	61	43	44	43	44	43	0.29	10.91	300	300	300	300
15	1	46	34	34	35	34	35	0.05	300	300	300	300	300
15	2	47	29	30	29	30	29	0.06	21.71	300	300	300	300
15	3	48	34	34	35	34	34	0.05	300	300	300	300	300
15	4	48	27	24	24	24	24	0.06	0.26	7.58	300	300	300
15	5	58	53	52	52	54	53	0.94	300	300	300	300	300
15	6	67	45	46	46	46	46	0.14	198.44	300	300	300	300
15	7	47	33	33	33	34	33	0.05	23.22	300	300	300	300
15	8	50	39	39	39	39	39	0.06	300	300	300	300	300
15	9	54	35	36	36	36	35	0.06	300	300	300	300	300
15	10	65	40	40	40	40	40	0.06	6.01	112.92	300	300	300

Latency-Aware 2-Opt Monotonic Local Search for Distributed Constraint Optimization

Ben Rachmut

Ben-Gurion University of the Negev, Beersheba, Israel

Roie Zivan

Ben-Gurion University of the Negev, Beersheba, Israel

William Yeoh

Washington University in St. Louis, MO, USA

Abstract

Researchers recently extended Distributed Constraint Optimization Problems (DCOPs) to Communication-Aware DCOPs so that they are applicable in scenarios in which messages can be arbitrarily delayed. Distributed asynchronous local search and inference algorithms designed for CA-DCOPs are less vulnerable to message latency than their counterparts for regular DCOPs. However, unlike local search algorithms for (regular) DCOPs that converge to k-opt solutions (with $k > 1$), that is, they converge to solutions that cannot be improved by a group of k agents), local search CA-DCOP algorithms are limited to 1-opt solutions only.

In this paper, we introduce Latency-Aware Monotonic Distributed Local Search-2 (LAMDLS-2), where agents form pairs and coordinate bilateral assignment replacements. LAMDLS-2 is monotonic, converges to a 2-opt solution, and is also robust to message latency, making it suitable for CA-DCOPs. Our results indicate that LAMDLS-2 converges faster than MGM-2, a benchmark algorithm, to a similar 2-opt solution, in various message latency scenarios.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed Constraint Optimization Problems, Distributed Local Search Algorithms, Latency Awareness, Multi-Agent Optimization

Digital Object Identifier 10.4230/LIPIcs.CP.2024.24

Supplementary Material *Software*: `https://github.com/benrachmut/CADCOP_CP_2024`
archived at `swh:1:dir:4d80c62cc077c5ad44894acf592359cd0ce1580c`

Funding This research is partially supported by US-Israel Binational Science Foundation (BSF) grant #2018081.

1 Introduction

A promising multi-agent approach for addressing distributed applications, where agents aim to achieve mutual optimization goals, is by modeling them as *Distributed Constraint Optimization Problems* (DCOPs) [12, 16, 5]. An illustrative example of such an application is a smart home, where various smart devices must coordinate to create a schedule that optimizes user preferences and satisfies constraints [6, 19]. In this context, decision-makers are represented as "agents" that assign "values" to their respective "variables", and the objective is to optimize a global objective in a decentralized manner.

DCOPs are NP-hard [12] and, thus, considerable research effort has been devoted to developing incomplete algorithms for finding good solutions quickly [23, 10, 24, 3, 4, 20, 8, 14]. Distributed local search algorithms such as *Distributed Stochastic Algorithm* (DSA) [24] and *Maximum Gain Message* (MGM) [10] are two of the most popular incomplete DCOP algorithms.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 24; pp. 24:1–24:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Most state-of-the-art local search DCOP algorithms (including DSA and MGM) are synchronous. However, the general setting in which agents operate is inherently *asynchronous*. Synchronization is achieved through message exchanges in each iteration of the algorithm, in which an agent receives messages sent by its neighbors in the previous iteration, performs computation, and sends messages to all its neighbors [24, 26]. This ensures that at iteration k, an agent has access to all information sent to it during iteration $k-1$. The synchronous design enables the attainment of some desirable properties. For example, MGM agents achieve monotonicity on the quality of the solutions found by modifying their value assignments while ensuring that neighboring agents do not concurrently replace their assignments [10].

There exists a class of local search DCOP algorithms that guarantee that the solutions found are k-opt (i.e., they cannot be improved by a group of k agents) [15]. MGM is a 1-opt algorithm and MGM-2 is an extension that is a 2-opt algorithm. Unfortunately, their synchronous designs take advantage of the overly simplistic communication assumptions in the DCOP model, which do not reflect real-world scenarios. Notably, the assumption that all messages arrive instantaneously or with negligible and bounded delays is impractical, given that real-world networks may suffer from delays due to congestion and limited bandwidth.

To address these limitations, researchers introduced *Distributed Asynchronous Local Optimization* (DALO), an asynchronous k-opt algorithm for solving DCOPs [9]. Unfortunately, its design lacks robustness in scenarios with message delays, restricting its applicability. Specifically, agents try to form groups by asking others to commit to the process they initiate, ensuring an up-to-date local view when computing local optimization. Because neighboring agents attempt to form groups simultaneously, a randomly set local timer is used. Agents can only commit to other groups if a lock request is sent during this timer's duration. However, this design fails when the local timer is not coordinated with the magnitude of message delays, resulting in agents rejecting each other's requests. Additionally, DALO's design does not adequately handle messages not arriving in the order that they were sent. This raises concerns about the algorithm's guaranteed properties under such conditions.

Recent studies [17, 18] explored the performance of local search algorithms for solving DCOPs in the presence of imperfect communication, where messages can be delayed. They demonstrated the significant impact of message latency on the performance of synchronous distributed local search algorithms, especially on property guarantees and convergence rates of MGM. Consequently, a 1-opt *Latency Aware Monotonic Distributed Local Search* (LAMDLS) algorithm was proposed [18]. LAMDLS uses an ordered coloring scheme to prevent neighboring agents from replacing assignments concurrently while preventing agents from waiting for messages as they do in MGM. As a result, LAMDLS demonstrates a quicker convergence rate compared to MGM.

Building on the success of LAMDLS, we advance the research on distributed algorithms that are robust to message delays by proposing LAMDLS-2, which allows agents to form pairs and coordinate their value assignment selection, while maintaining monotonicity and converging to a 2-opt solution. LAMDLS-2 enables sequential change of values among paired agents. Agents utilize a unique pairing selection process and an ordering scheme that allows concurrent value modifications for unconstrained pairs. We further discuss a scheme that will allow to generation of a similar monotonic k-opt algorithm for any $1 \leq k \leq n$ in future studies. We prove the monotonicity of LAMDLS-2 and its convergence to a 2-opt solution. Our empirical results indicate that LAMDLS-2 converges significantly faster, in environments with a variety of latency patterns, compared to MGM-2, an existing 2-opt DCOP algorithm.

2 Background

We present background on Distributed Constraint Optimization Problems (DCOPs), k-opt algorithms, including the 2-opt algorithm MGM-2, Communication-Aware DCOPs (CA-DCOPs), and Latency-Aware Monotonic Distributed Local Search (LAMDLS).

2.1 Distributed Constraint Optimization Problems (DCOPs)

A DCOP is a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$, where $\mathcal{A}$ is a finite set of agents $\{A_1, A_2, \ldots, A_n\}$; $\mathcal{X}$ is a finite set of variables $\{X_1, X_2, \ldots, X_m\}$, where each variable is held by a single agent (an agent may hold more than one variable); $\mathcal{D}$ is a set of domains $\{D_1, D_2, \ldots, D_m\}$, where each domain D_i contains the finite set of values that can be assigned to variable X_i and we denote an assignment of value $d \in D_i$ to X_i by an ordered pair $\langle X_i, d \rangle$; and $\mathcal{R}$ is a set of constraints (relations), where each constraint $R_j \in \mathcal{R}$ defines a non-negative *cost* for every possible value combination of a set of variables and is of the form $R_j : D_{j_1} \times D_{j_2} \times \ldots \times D_{j_k} \rightarrow \mathbb{R}^+ \cup \{0\}$. A *binary constraint* refers to exactly two variables and is of the form $R_{ij} : D_i \times D_j \rightarrow \mathbb{R}^+ \cup \{0\}$.

A *binary DCOP* is a DCOP in which all constraints are binary. Agents are *neighbors* if they are involved in the same constraint. A *partial assignment* (PA) is a set of value assignments to variables, in which each variable appears at most once. *vars(PA)* is the set of all variables that appear in partial assignment PA (i.e., $vars(PA) = \{X_i \mid \exists d \in D_i \wedge \langle X_i, d \rangle \in PA\}$). A constraint $R_j \in \mathcal{R}$ of the form $R_j : D_{j_1} \times D_{j_2} \times \ldots \times D_{j_k} \rightarrow \mathbb{R}^+ \cup \{0\}$ is *applicable* to PA if each of the variables $X_{j_1}, X_{j_2}, \ldots, X_{j_k}$ is included in $vars(PA)$. The *cost of a partial assignment* PA is the sum of all applicable constraints to PA over the value assignments in PA. A *complete assignment* (i.e., *solution*) is a partial assignment that includes all variables ($vars(PA) = \mathcal{X}$). An *optimal solution* is a complete assignment with minimal cost.

For simplicity, we assume that each agent holds exactly one variable (i.e., $n = m$) and we focus on binary DCOPs. These assumptions are common in DCOP literature (e.g., [16, 22]).

2.2 *k*-opt and Region-opt Algorithms

Most local search DCOP algorithms are *synchronous* [24, 10, 26]. In MGM, a step (in which agents decide on value replacements) includes two synchronous iterations. First, agents receive their neighbors' updated value assignments and seek improving alternatives for their assignments. Next, agents share their maximal gain from a value replacement. An agent replaces its assignment if its gain exceeds all its neighbors' reported gains. MGM guarantees that agents compute cost reductions using up-to-date information and prevents simultaneous assignment changes by neighbors. This leads to monotonic global cost improvement. MGM also guarantees convergence to a 1-opt solution.

k-opt generalizes the 1-opt solution concept to any case where k agents cannot improve a solution [10, 15]. An algorithm ensuring this must allow all possible coalitions of k agents to seek improving assignments. A well-known algorithm that guarantees the convergence to a 2-opt solution ($k = 2$) is MGM-2. In MGM-2, agents pair with neighbors to coordinate bilateral assignment replacements. MGM-2's step has five synchronous iterations. In the first three, agents attempt to form pairs, exchange information, and identify the best bilateral gains for these pairs. Unpaired agents select the highest unilateral gain possible. In the remaining two iterations, as in standard MGM, each agent evaluates whether its gain (or the gain of its pair) is larger than the gain of all its neighbors. An agent that is part of a pair, must receive the approval of its partner, that their gain is larger than the gain of the partner's neighbors as well.

A general k-opt algorithm was proposed by Pearce an Tambe [15] and further generalized to region-optimal algorithms by Vinyals *et al.* [21]. A region is defined by groups of agents that are monitored by the same agent. Commonly, these groups are classified according to two parameters: Their size (k) and the distance of the agents from the monitoring agent (t). In each step of the algorithm, monitoring agents select a group from their region, aggregate their information, select an alternative assignment, calculate the corresponding gain, and propagate it to the neighbors of all agents in the group. Groups with a larger gain than the gains reported by their neighbors replace their assignments.

2.3 Communication-Aware DCOPs (CA-DCOPs)

CA-DCOPs [18, 27] extend standard DCOPs by using a *Constrained Communication Graph* (CCG) to model the communication latency between pairs of agents. Thus, they can model any pattern of imperfect communication. Specifically, each edge e in the CCG represents the imperfect communication between a pair of agents and is associated with a latency distribution function.

2.4 Latency-Aware Monotonic Distributed Local Search (LAMDLS)

LAMDLS [17] is monotonic and 1-opt (like MGM). By allowing agents to consider value assignment replacements using a partial order, it effectively mitigates the impact of message latency and facilitates faster convergence. To establish the partial order structure it uses the *Distributed Ordered Color Selection* (DOCS) algorithm. DOCS divides the agents into subsets, where agents in each subset have the same color. Colors are ordered (i.e., there is a mapping from colors to the natural numbers from 1 to NC, where NC is the number of colors). The neighbors of each agent must hold a different color than its own, and the agent must know which neighbors are ordered before it and which after. During the algorithm execution, each agent keeps track of its neighbors' computation steps, updates them with its selection, and performs the k-th iteration when neighbors with a lower color index complete k iterations and those with a higher index complete $k-1$ iterations. LAMDLS demonstrates a faster convergence rate compared to MGM, with the difference becoming more noticeable as the magnitude of message delays increases [18].

3 LAMDLS-2

Latency-Aware Monotonic Distributed Local Search 2 (LAMDLS-2) is a monotonic algorithm that converges to a 2-opt solution. 2-opt algorithms, such as MGM-2, achieve this property by allowing all pairs of agents to make an attempt to improve any assignment that the algorithm traverses, unless it is revised before they get their chance. The main difference in LAMDLS-2 is the method used to generate pairs that will cooperatively suggest an assignment revision. In contrast to MGM-2, where a query response process is used to determine pairs, LAMDLS-2 uses DOCS to find an ordered coloring scheme for determining the pairs. Once DOCS selects an order, the pairs are generated deterministically accordingly, and there are no additional messages required for the pairing process. Thus, message latency has smaller deteriorating effects on this algorithm compared to MGM-2. In order to make sure that all pairs of agents get their chance to improve the current assignment, DOCS is performed iteratively, using random agent indexes. This results in random orderings, which eventually allow all possible pairs to be generated. We present the algorithm in more details below.

Algorithm 1 LAMDLS-2.

```
Input: $N(i)$
 1: $value_i \leftarrow$ selectRandomValue()
 2: $sc_i \leftarrow 1$
 3: for each  $A_j \in N(i) : v^{N(i)}[j] \leftarrow 1$
 4: $docsId_i \leftarrow i$
 5: for each  $A_j \in N(i) : docsIds^{N(i)}[j] \leftarrow j$
 6: $co_i, co^{N(i)} \leftarrow$ DOCS($i$,$docsIds^{N(i)}$)
 7: while stop condition not met:
 8:     pairPhase($sc_i$, $v^{N(i)}$,$co_i$,$co^{N(i)}$,$docsIds^{N(i)}$)
 9:     $docsId_i \leftarrow$ random(0,1)
10:     sendDocsId($N(i)$,$docsId_i$)
11:     $docsIds^{N(i)} \leftarrow$recieveAllDocsIds()
12:     $co_i, co^{N(i)} \leftarrow$DOCS($docsId_i$,$docsIds^{N(i)}$)
```

LAMDLS-2 is composed of two alternating phases: *Ordering* and *Pair Selection*. Algorithm 1 presents the pseudocode performed by an agent A_i. In the ordering phase, agents select ordered colors using the DOCS algorithm (lines 6 and 12). In the pair selection phase, agents select partners and collaboratively adjust assignments using the pairPhase function (line 8). The algorithm's input includes the set $N(i)$ that includes A_i's neighbors.

The algorithm starts with agent A_i randomly selecting $value_i$ for its value assignment (line 1). In addition, A_i maintains a step counter sc_i, which is incremented each time A_i selects a value assignment, and a step counter for each of its neighbors in the set $v^{N(i)}$. Entry $v^{N(i)}[j]$ is updated when a value assignment update from a neighbor A_j is received. Both sc_i and entries in $v^{N(i)}$ are initialized to 1 (lines 2-3).

3.1 Ordering Phase

In the ordering phase, agents use the DOCS algorithm to select ordered colors, as in LAMDLS [18]. Following DOCS, A_i receives its selected color co_i, and the colors $co^{N(i)}$ are selected by its neighbors. In contrast to LAMDLS, where agents use their indexes within the DOCS procedure to select colors, in LAMDLS-2 the agents use random values ($docsId_i$). A_i retains the $docsId$'s of its neighbors in the set $docsIds^{N(i)}$. Once A_i has completed the pair selection phase, before re-starting DOCS, it selects a new value for $docsId_i$ and waits for the $docsId$ values of its neighbors to be updated in $docsIds^{N(i)}$ (lines 9-11). Hence, each time DOCS operates, it uses different values for $docsId$ and $docsIds^{N(i)}$ and, thus, the probability that it would generate distinct values for co_i and $co^{N(i)}$ is very high. In line 6, DOCS is initiated before the pair selection phase. Thus, initial values for the $docsIds$ are according to the agents' indexes. The use of randomized $docsId$ values in DOCS results in diverse and randomized ordered color selections in the different steps of the algorithm.

Algorithm 2 details the execution of the DOCS method by some agent A_i. At the initiation of the algorithm, A_i holds its own $docsId_i$ and the $docsIds$ of its neighbors (in $docsIds^{N(i)}$). When the algorithm terminates A_i holds the color it selected (co_i) and the colors of its neighbors ($co^{N(i)}$). The algorithm begins by initializing the variables co_i and $co^{N(i)}$ (lines 1-2). If the value of $docsId_i$ is the smallest among the values in $docsIds^{N(i)}$, A_i sets the value of co_i to 1 and sends this information to its neighbors. Afterward, A_i remains idle until it receives updated information about the colors selected by its neighbors (line 7). The algorithm terminates when A_i becomes aware of the colors of all its neighbors

Algorithm 2 LAMDLS-2 color selection DOCS.

```
Input: $docsId_i, docsIds^{N(i)}$
Output: $co_i, co^{N(i)}$
 1: $co_i \leftarrow$ None
 2: for each  $A_j \in N(i) : co^{N(i)}[j] \leftarrow None$
 3: if $\min(docsId_i, docsIds^{N(i)})$ then:
 4:    $co_i \leftarrow 1$
 5:    send $(N(i),co_i,value_i)$
 6: while not aware of all colors:
 7:    when color from $A_j$:
 8:       update $(co_j,\ co^{N(i)}[j])$
 9:       update $(value_j)$
10:       if $co_i$ is None and can select color then:
11:          $co_i \leftarrow$ selectMinAvilableColor$(co^{N(i)})$
12:          $value_i \leftarrow$ selectValueUnilaterally$(co^{N(i)})$
13:          send $(N(i),co_i,value_i)$
14: return $co_i, co^{N(i)}$
```

and selects a color for co_i (line 6). Upon receiving updated information about the colors selected by its neighbors, A_i updates $co^{N(i)}$. Then it checks if it can select a color. If a color was not chosen previously and A_i receives the colors of all its neighbors with smaller indices in $docsIds^{N(i)}$, it selects the color with the smallest number that hasn't been chosen by any of its neighbors and sends this color to its neighbors. This process ensures that eventually, the color selected by each agent is different from the colors selected by its neighbors. To accelerate the convergence process of LAMDLS-2, agents can select values while they select their colors (line 12).

3.2 Pair Selection Phase

Like MGM-2, LAMDLS-2 achieves monotonicity and convergence to a 2-opt solution by allowing agents to form pairs and select the best mutual assignment, while their neighbors avoid replacing their assignments at the same time. The main difference from MGM-2 is the use of the ordered color scheme by agents to decide when to suggest pairing with their neighbors, which neighbor they should make suggestions to, and whether to accept such suggestions from their neighbors. Agent A_i selects A_j as its partner and shares all relevant information, including its current assignment, the content of its domain, its neighbors, their assignments, and its constraints. Then, when allowed, A_j proceeds to calculate the bilateral value assignments for both A_i and itself and notifies A_i about its updated value assignment. The phase concludes when the agent makes a selection of its value assignment (denoted by $value_i$). If the pairing process is successful, A_j selects the value assignment for both A_i and A_j. However, if the pairing process fails (i.e., A_i is not paired with any other agent), A_i can unilaterally select its assignment. Following each selection of a value assignment, there is an update of the agent's step counter (sc_i), accompanied by a message sent to its neighbors, which includes $value_i$ and sc_i.

Below, we provide a more detailed description of the Pair Selection phase and present its pseudocode in Algorithm 3. Agent A_i divides its neighbors into two sets, $PC(i)$ and $FC(i)$, based on the input variables co_i and $co^{N(i)}$. $PC(i)$ includes neighbors with color indices smaller than co_i, while $FC(i)$ includes neighbors with larger color indices. This division is

Algorithm 3 LAMDLS-2 Pair Selection Phase.

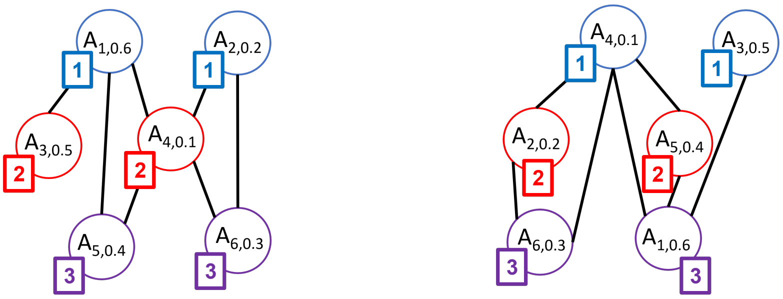
```
Input: $N(i)$,$sc_i$, $v^{N(i)}$,$co_i$,$co^{N(i)}$,$docsIds^{N(i)}$
 1: $varConsist \leftarrow [sc_i,\ v^{N(i)}, co_i, co^{N(i)}]$
 2: $sn, nInfo \leftarrow None$
 3: $sn \leftarrow$offer($varConsist$,$sn$,$docsIds^{N(i)}$)
 4: while phase not completed:
 5:   when receive message from $A_j$:
 6:     if message is of type value then:
 7:       update($values^{N(i)}[j]$,$v^{N(i)}[j]$)
 8:       if message.$sender$ is $sn$ :
 9:         $value_i \leftarrow$ selectValueUnilaterally()
10:       else:
11:         $sn \leftarrow$offer($varConsist$,$sn$,$docsIds^{N(i)}$)
12:     if message is of type reply then:
13:       updateValue(message.getValue(i))
14:     if message is of type offer then:
15:       $nInfo \leftarrow$getOfferInfo(message,$docsIds^{N(i)}$)
16:       reply($varConsist$,$nInfo$)
17: $sc_i \leftarrow sc_i + 1$
18: sendLocalInfo($N(i)$,$value_i$,$sc_i$)
```

used to determine the selected neighbor (sn) that A_i shares its information with. Agents take into consideration co_i, $co^{N(i)}$, sc_i, and $v^{N(i)}$ while deciding when to initiate partnerships and how to respond to partnership requests. LAMDLS-2 agents exchange three types of messages during the pair selection phase:

- **Value** (lines 6-11): Triggers an update of $v^N(i)$, which allows agents to initiate partnerships and reply to them.
- **Reply** (lines 12-13): Contains the value assignment found by the neighbor the agent paired with.
- **Offer** (lines 14-16): Contains the relevant information sent when an agent offers a neighbor to form a pair.

Upon receiving a **value** message, A_i updates its local view (line 7) and then considers two scenarios that may be triggered: Either rejecting or initiating an offer. If the sender of the value message is the agent (sn) to whom A_i has made an offer in the current phase (lines $8-9$), A_i considers the value message as a rejection of its offer. Conversely, if A_i did not initiate an offer during the current phase, a value message reception may prompt an offer initiation due to an update in $v^{N(i)}$, as A_i examines the necessary condition to offer (lines $10-11$).

In the offer function, A_i checks its eligibility to make an offer when the condition $sc_i = sc_j - 1$ is met for every $A_j \in PC(i)$. The offer function is activated under two circumstances. The first occurs when a value is received from the neighbor A_j. This results in an update of sc_j, which might satisfy the condition that will allow A_i to offer. The second is tied to the base case that initiates the phase for agents meeting the condition due to $pc = \emptyset$ (line 3). When the agent decides to make an offer, it selects a neighbor (sn) using a deterministic process. The chosen neighbor must meet the following conditions: Its color index is larger by one from the color index of A_i ($co_i + 1 = co^{N(i)}[sn]$), and the value of $v^{N(i)}[sn]$ equals sc_i. If multiple agents meet these conditions, the neighbor with the smallest



(a) Identity index. **(b)** Random index.

Figure 1 Two different numerical graph color partitions.

value in $docsIds^{N(i)}$ is chosen. If sn is found, A_i sends an **offer** message containing all relevant information for a bilateral value assignment selection. The function returns sn for future examination of whether the offer was accepted or rejected. If no neighbor satisfies the conditions to qualify as sn, A_i unilaterally selects a value assignment and indicates that the phase is completed. After sending an offer message, A_i enters an idle state, awaiting a reply from sn. Upon receiving a **reply** message, A_i is informed of the offer's acceptance. Subsequently, A_i updates its $value_i$ based on the bilateral decision made by sn (line 13).

Upon receiving an **offer** message, A_i stores the shared information and uses the reply function (line 15). A_i has the option to either accept the offer or reject it. A_i can only accept a single offer per step. If A_i accepts the offer, it proceeds to calculate values for itself and its partner using its local information and the information received from its partner and sends a **reply** message back to it. However, if A_i declines the offer, indicating that it has already formed a bilateral value assignment change with a different agent, it sends a message containing its value to inform the sender that the offer was rejected. If A_i receives multiple offers, it selects as a partner the offering agent with the lowest index in $docsIds^{N(i)}$. Let $PO(i)$ denote the set of agents that sent offers to A_i in the current pair selection phase. An offer can be accepted by A_i if the following condition is met: for each agent $A_j \in PC(i) \backslash PO(i)$, $sc_i = sc_j - 1$. Until this condition is met, A_i will remain idle and wait for messages to arrive.

3.3 Demonstration of LAMDLS-2

In the following sub-section, we describe the beginning of a high-level trace of LAMDLS-2, when operating on the constraint graph presented in Figure 1. In this graph, each node represents an agent, and the corresponding colors (selected using DOCS) of the agents are displayed beneath the nodes. Specifically, each node represents an agent $A_{i,docsId}$, where i is the agent's index and $docsId$ is a randomly assigned value that is drawn before the next step.

After agents randomly select values for their assignments, each agent initializes its $docsId$. They also set the entries of $docsIds^{N(i)}$ with the identity indices of their respective neighbors, e.g., A_1: $docsId_1 = 1$ and $docsIds^{N(1)} = [\langle A_3 : 3\rangle, \langle A_4 : 4\rangle, \langle A_5 : 5\rangle]$

First Step

After initiation, agents proceed to execute DOCS. Figure 1 (a) presents the outcome of the color selection process carried out by DOCS. This process utilizes the values of $docsId$ of the agents, therefore the outcome is dependent on their selection. In the example at hand, agents

A_1 and A_2 do not have neighbors with smaller indices, so they select the color 1 (blue) and communicate this information to their neighbors. Among these neighbors, agents A_3 and A_4 do not have other neighbors with smaller indices, so they choose the color 2 (red) and send messages including this information to their neighbors. Finally, agents A_5 and A_6 select the color 3 (purple). This completes the color selection phase.

When the pair selection phase begins, both A_1 and A_2, which selected the color 1, can choose a neighbor and send an offer along with the relevant information. They are eligible because $PC(1) = \emptyset$ and $PC(2) = \emptyset$. A_1 must select a neighbor with the smallest $docsId$ color among its neighbors with color 2. It has two neighbors with color 2, A_3 and A_4), and among them, A_3 has a smaller $docsId$, thus, it sends the offer to A_3. A_2 selects A_4, since it is its only neighbor with color 2.

Upon receiving an offer, A_3 is eligible to respond, given that A_1 is its only neighbor. A_3 selects values for itself and for A_1, updating sc_3 to 2. It then sends a reply to A_1, who adjusts its assignment and updates sc_1 to 2, notifying all its neighbors including A_4.

After receiving an offer from A_2, A_4 must wait for an update from A_1 (which is included in $PC(4)$). Following this update, A_4 selects values for itself and for A_2, increments sc_4 to 2, responds to A_2 and informs its neighbors of the new selected value. Subsequently, A_2 updates its value, sc_2 becomes 2, and it informs its neighbors too.

At this point, agents with colors 1 and 2 have already chosen value assignments. Upon receiving this information, A_5 updates $v^{N(5)} = [\langle A_4 : 2\rangle, \langle A_5 : 2\rangle]$. Thus, when receiving a value message that finalizes the update of $v^{N(5)}$, A_5 is eligible to offer, given that $sc_5 = 1$ and $sc_1 = sc_4 = 2$. While attempting to find a suitable partner, A_5 will pick a value unilaterally since no agent in $co^{N(5)}$ holds color 4 (which is one greater than $co_5 = 3$). Similarly, A_6 will also independently select its value assignment. This finalizes the second phase of the first step.

Second Step

At the beginning of the second step, agents select random $docsId$s and send messages that inform their neighbors of their selection.

Next, agents execute DOCS using the random $docsId$s selected and generate the color selection that is depicted in Figure 1 (b), as described next: Agents A_4 with $docsId_4 = 0.1$ and A_3 with $docsId_3 = 0.5$ do not have neighbors with smaller $docsId$ values, leading them to select color 1 (blue) and communicate this decision to their neighbors. Agents A_2 with $docsId_2 = 0.2$ and A_5 with $docsId_5 = 0.4$ can then select the color 2 (red) and convey it to their neighbors. Eventually, agents A_6 with $docsId_6 = 0.3$ and A_1 with $docsId_1 = 0.6$ select color 3 (purple) and inform their neighbors.

In the pair selection phase, agent A_4 selects A_2 as its partner and forwards an offer (since $docsId_4 < docsId_5$, i.e., $0.2 < 0.4$). Agent A_3 changes its value independently, as its only neighbor A_1 has color 3. Upon receiving a value message from A_4, A_5 can send an offer to A_1. After A_1 receives a value update from A_4, it can respond to A_5. Notably, in the previous step, the pair A_5 and A_1 did not form a partnership. When A_6 receives value messages from A_2 and A_4 ($PC(6) = \{A_2, A_4\}$), it attempts to select a neighbor. Failing to do so (no neighbors in $FC(6)$), it selects a value on its own.

3.4 Theoretical Properties

We now prove that LAMDLS-2 is monotonic and convergence to a 2-opt solution. Our monotonicity proof stems from previous studies that proved the monotonicity of MGM, MGM-2, and LAMDLS [11, 18] based on the fact that, in DCOP algorithms, when a single agent or a pair of agents improve their local state, while their neighbors remain idle, the global cost improves as well. Thus, it remains to show that when an agent or a pair of agents improve their local state in LAMDLS-2, their neighbors are idle until the messages regarding the assignment replacements that were performed by the agent or pair of agents arrive.

▶ **Lemma 1.** *In a DCOP (with symmetric constraints), when an agent A_i is the only agent replaces its assignment, while none of its neighbors ($NC(i)$) replace their assignments, and this replacement results in a local gain, it also results in an improvement of the global cost.*

Proof. Denote the global cost before A_i's assignment replacement by gc and the local gain following A_i's assignment replacement by LR_i. Since the problem is symmetric, the sum of local gains of A_i's neighbors is also equal to LR_i. Since we assumed that $LR_i > 0$, $gc > gc - 2LR_i$. ◀

▶ **Lemma 2.** *When some agent A_i initiates a partnership offer, all agents in $N(i)$ that do not partner with A_i avoid replacing their assignments until A_i completes its assignment replacement.*

Proof. For A_i to be active, sc_i must be equal to k (i.e., it has not been incremented since the color selection phase) and, for each agent $A_{i'} \in PC(i)$, $sc_{i'} = k + 1$. Thus, when A_i sends an offer, all agents in $PC(i)$ have already incremented their step counters. In addition, for each agent $A_{j'} \in FC(i)$ (i.e., $A_i \in PC(j')$), until sc_i is incremented, $A_{j'}$ cannot send an offer or replace its assignment. ◀

▶ **Lemma 3.** *When agent A_i initiates a partnership offer to A_j, agents in $N(j)$ do not replace their assignments until A_j completes its assignment replacement.*

Proof. Agents in $FC(j)$ cannot offer or reply to an offer until sc_j is incremented. On the other hand, for the agents in $PC(j)$, there are two cases:

- $A_{i'} \in PC(j)(i \neq i')$ **did not offer to** A_j. Then, A_j will not reply and replace assignments until $sc_{i'}$ is incremented, which can happen only after $A_{i'}$ replaces its assignment. Thus, it cannot happen concurrently with the assignment replacement of A_j.
- $A_{i'} \in PC(j)(i \neq i')$ **did offer to** A_j. Then, either A_j pairs with it, or it sends a rejection reply only after it completed the assignment replacement. Thus, they do not replace assignments concurrently. ◀

▶ **Proposition 4.** *LAMDLS-2 is monotonic (i.e., each assignment replacement improves the global cost of the complete assignment held by the agents).*

Proof. Follows immediately from Lemma 2 and Lemma 3. While agents replace their value assignments, none of their neighbors can replace their assignments. ◀

▶ **Proposition 5.** *At each pair selection phase, every agent that receives an offer will reply (positively to one of the offering agents and negatively to the rest).*

Proof. We prove by induction, using an order on all agents that can receive an offer (i.e., all agents except for the ones with the color 1; we will assume that the colors are numbered from 1 to NC). When colors are selected, the step counters of all agents are equal (e.g., $sc_i = k$ for all i). Agents of the same color have a different $docsId$. Thus, the order between every two agents that can receive an offer is determined first according to their color (small colors come first). If the colors are equal then the tie is broken using their $docsId$ (smaller comes first).

Recall that the conditions for an agent A_j to reply to an offer are that all agents in $PC(j)$ either offered to A_j or their step counter equals $sc_j + 1$. Assume that A_i is the agent with the smallest $docsId$ among the agents with color 2. It will receive offers from all its neighbors with color 1. Thus, it will be able to select a neighbor to reply positively to its offer, and all its other neighbors will get a negative reply and unilaterally select an assignment.

The agent with the second smallest $docsId$ that received an offer (A_j) with color 2 can have two types of neighbors with color 1: Ones that sent an offer to A_i and ones that sent an offer to A_j. The ones that sent an offer to A_i, after they receive the reply from A_i, will attempt to replace their assignment and increase their step counter. After receiving all indications regarding the increase of the step counters of these agents, A_j can reply to the agents that sent it an offer.

Assume that later on during the algorithm run, A_i is the agent that received an offer, with $sc_i = k$, and with the smallest color index and the smallest $docsId$ among the agents that received an offer and did not yet reply (i.e., if agent $A_{i'}$ received an offer and did not yet reply, then either $co_{i'} > co_i$ or $co_{i'} = co_i \& docsId_{i'} > docsId_i$). Since there are no agents with a color smaller than co_i that received an offer and did not reply, then there is no agent that sent an offer with a color index smaller than $co_i - 1$, which a reply was not sent to it. Thus, the members of $PC(i)$ include two types of agents: Agents that sent an offer to A_i and agents that a reply for the offers they sent was already sent to them. Thus, once all the offers from agents of the first type and the indications on the increase in the step counter of the agents from the second type arrive, A_i will be able to reply to the offers sent to it. ◀

An immediate correlation from Proposition 5 is that the algorithm terminates its phases and does not deadlock. The ordering phase uses the DOCS algorithm and its correctness and termination have been established in previous studies [2, 18]. The pair selection phase must terminate because every agent that receives an offer must reply, and thus, all agents can perform the assignment selection method and increase their step counter.

▶ **Proposition 6.** *LAMDLS-2 converges to a 2-opt solution.*

Proof. According to Proposition 4, LAMDLS-2 is monotonic. Thus, since the problem is finite, it must converge to some solution. To prove that the solution it converges to is 2-opt, we need to establish that following convergence, every pair of neighboring agents will get a chance to form a pair and check all their alternative assignments. For agent A_i to form a pair with agent A_j, one of them (without loss of generality we select A_i) needs to send an offer to the other (A_j), and A_j needs to respond positively. This happens in two conditions: (1) $co_i = co_j - 1$; or (2) for any agent $A_{j'}$ with $co_j = co_{j'}$, $docsId_j < docsId_{j'}$. Since colors and $docsId$s are selected randomly, this situation will eventually occur. ◀

4 Extension to a Region-Optimal Algorithm

Similar to how MGM-2 was extended to k-opt and then to region-opt algorithms, we propose an extension of LAMDLS-2 to LAMDLS-ROpt. In LAMDLS-ROpt, an agent initiating ad-hoc coalition formation takes on a mediator role. Unlike LAMDLS-2, where this agent includes its information in the offer message sent to the selected neighbor, in LAMDLS-ROpt, the mediator sends an offer message to neighboring agents within the coalition it aims to form. This message invites them to join and prompts other specified neighbors to join as well. The information of the agents in the forming coalition is sent back to the monitoring agent, who selects an alternative assignment for the group. The group replaces the assignment if the mediator is ordered before the mediators of neighboring groups according to the ordered color and *docsId* scheme. This process is similar to the region-optimal algorithm RODA [7]. The difference is in its repeated selection of mediators, the selection of members in the groups included in the mediators' regions, and the order in which groups replace assignments, in a designated sequence, according to the ordered color scheme used in LAMDLS and LAMDLS-2. We leave for future work the investigation of the performance of LAMDLS-ROpt in comparison with RODA.

5 Experimental Evaluation

We present a comprehensive study that compares the proposed LAMDLS-2 algorithm to MGM-2, solving a variety of DCOP benchmarks in environments with different patterns of message latency.

5.1 Experimental Design

In our experiments, we use the same asynchronous simulator used by researchers for CA-DCOP algorithms.[1] The experiments were conducted on a Windows Server 2019 Standard operating system, with an Intel Xeon Silver 4210 CPU 2.20GHz.

We follow the approach used in the literature [17, 18] to evaluate the quality of the solutions of the algorithms, as a function of the asynchronous advancement of the algorithm, in terms of non-concurrent logic operations (NCLOs) [25, 13]. The utilization of NCLO ensures implementation independence and avoids double counting of simultaneous actions.

In each experiment, we randomly generated 100 different problem instances with 50 agents and we reported the average solution quality of the algorithms examined. To demonstrate the convergence of the algorithms, we present the sum of costs of the constraints involved in the assignment that would have been selected by each algorithm every $10,000$ NCLOs.

We simulated three types of communication scenarios: (1) Perfect communication; (2) Message latency selected from a uniform distribution $U(0, UB)$, where UB is a parameter indicating the maximum latency; and (3) Message latency selected from a Poisson distribution with $\lambda = |MSG|$ and then scaling it by a factor of m, where $|MSG|$ represents the number of messages that are currently delivered in the system, and m is a scaling factor indicating the magnitude of the latency. This scenario is the evaluation of the impact of bandwidth load. Latency was also measured in terms of NCLOs.

We evaluated our algorithms on three problem types that are commonly used in the DCOP literature:

- **Uniform Random Problems**. These are random constraint graph topologies with densities 0.2 and 0.7. Each variable had a domain of 10 values, and constraint costs were uniformly selected between 1 and 100.
- **Graph Coloring Problems** [24, 4]. Each variable has three values (colors). Equal assignments between two neighbors incurred random costs from $U(10, 100)$, while non-equal assignments had 0 cost. The density was set at 0.05.
- **Scale free Network Problems** [1]. Initially, 10 agents were randomly selected and connected. Additional agents were sequentially added, connecting to 3 other agents with probabilities proportional to the existing agents' edge counts. Similar to the first type, variables had a domain of 10 values, and constraint costs ranged from 1 to 100.

[1] The simulation's code is available at `https://github.com/benrachmut/CADCOP_CP_2024`.

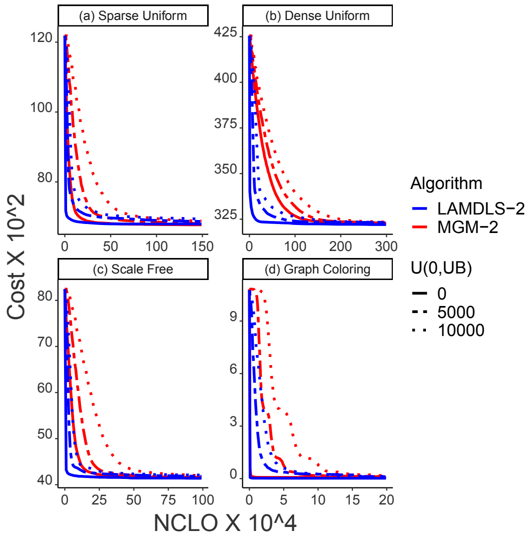


Figure 2 Solution quality as a function of NCLOs. Message delays are sampled from a uniform distribution.

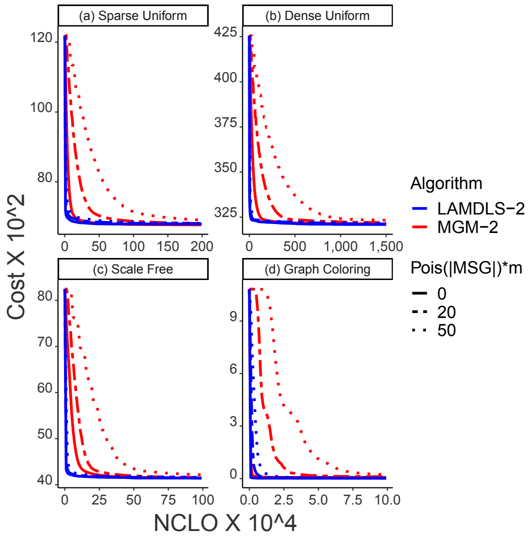


Figure 3 Solution quality as a function of NCLOs. Message delays sampled from a Poisson distribution linked to message volume.

5.2 Experimental Evaluation

Figure 2 presents a comparison between the results of two algorithms: The proposed LAMDLS-2 (represented by the blue curve) and MGM-2 (represented by the red curve). The comparison is performed on different problem types, as shown in each subgraph. The graph illustrates the performance of both algorithms in terms of the average global cost as a function of NCLOs. This enables the demonstration of the solution quality and the convergence speed for each algorithm. Latency is sampled from a uniform distribution, and the line type (solid, dashed, and dotted) corresponds to different magnitudes of latency, where $UB = \{0, 5{,}000, 10{,}000\}$. The results demonstrate that the algorithms converge to

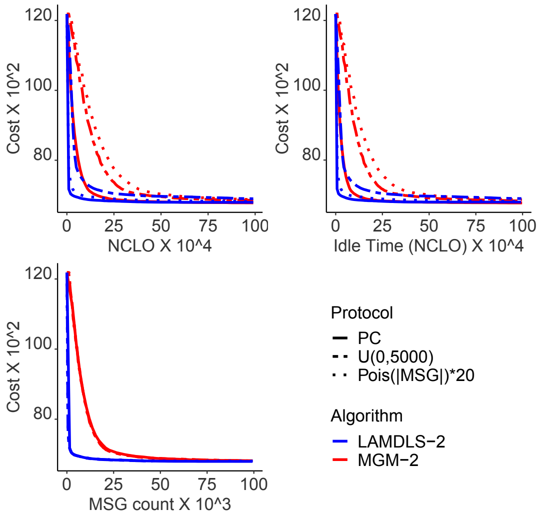


Figure 4 Solution quality as a function of different matrices in environments with different message delays.

solutions with similar quality, independent of message delays. This is expected because, in both algorithms, agents wait for updated information from their neighbors before they perform computation and replace assignments.

LAMDLS-2 demonstrates faster convergence than MGM-2 in scenarios with no message delays, except when solving graph coloring problems, where both algorithms show similar convergence rates. Moreover, LAMDLS-2 is more resilient to message delays than MGM-2. Its convergence rate remains relatively stable even with increasing delay, while MGM-2 experiences a more substantial slowdown in convergence as the latency magnitude increases. The most significant difference in the convergence rate between LAMDLS-2 and MGM-2 is observed in dense uniform problems (Figure 2(b)). Interestingly, LAMDLS-2 with the longest delays $UB = 10{,}000$ converges faster than MGM-2 with no delays. When solving graph coloring problems (Figure 2(d)), although the convergence rates are similar when communication is perfect, LAMDLS-2 exhibits a much faster convergence rate compared to MGM-2 when messages are delayed. These problems are characterized by low density among the examined types, leading to rapid convergence for both algorithms. For sparse uniform problems (Figure 2(a)), the impact of message delays on both LAMDLS-2 and MGM-2 is consistent and proportional. However, LAMDLS-2 maintains its superiority over MGM-2 in terms of convergence speed. When solving scale-free networks (Figure 2(c)), the negative impact on convergence rates is more pronounced for MGM-2 compared to LAMDLS-2 as the latency magnitude increases. Figure 3 presents the results of a similar experiment in which message delays were sampled from a Poisson distribution with the parameter $\lambda = |MSG| \cdot m$, where $m = \{0, 20, 50\}$. In this set of experiments, the resilience of LAMDLS-2 is pronounced regardless of the type of problem being solved. The increase in the latency magnitude did not significantly affect LAMDLS-2's convergence rate, unlike the significant effect it had on MGM-2.

The results in Figures 2 and 3 indicate a faster convergence rate of LAMDLS-2 in comparison with MGM-2. To investigate the reasons for this advantage, we present in Figure 4 the solution costs of the algorithms as a function of two additional elements in the algorithms' execution. These elements are the number of messages exchanged by the

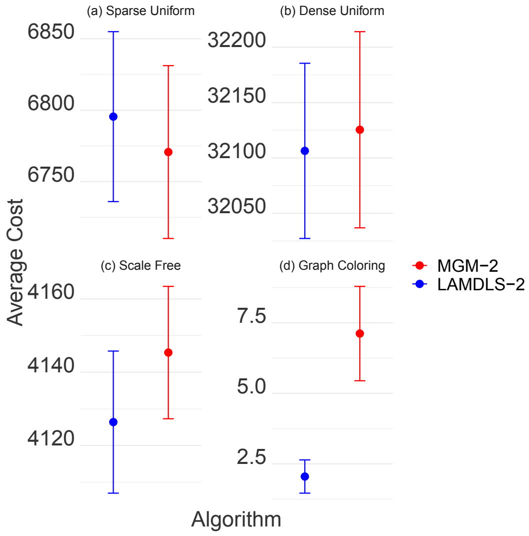


Figure 5 Average costs at convergence with error bars.

agents and the amount of time (in NCLOs) that agents were inactive (i.e., idle). Both algorithms solve sparse uniform problems under various communication scenarios: Perfect communication (PC) represented by the solid line, $U(0{,}5{,}000)$ represented by the dashed line, and $Pois(|MSG|) \cdot 20$ represented by the dotted line. While the three presented subgraphs illustrate the faster convergence rate of LAMDLS-2 compared to MGM-2, each of them highlights a distinct advantage of LAMDLS-2. The faster convergence in terms of message count indicates that LAMDLS-2 makes more economical use of the communication network. The faster convergence in terms of idle time indicates that agents in LAMDLS-2 are more active, and perform more concurrently.

In Figure 5, we present the average costs of both algorithms at convergence with SEM error bars. Overlapping bars across sparse, dense, and scale-free networks suggest no significant difference. Paired t-tests confirm this, with p-values above 0.05 (0.7514 for sparse, 0.8364 for dense, and 0.4839 for scale-free). For graph coloring problems, there is a significant difference (p-value 0.005), indicating diverse algorithmic performance in favor of LAMDLS-2.

6 Conclusions

We introduced Latency-Aware Monotonic Distributed Local Search 2 (LAMDLS-2), a distributed local search algorithm for solving DCOPs, which is monotonic and guarantees convergence to a 2-opt solution. LAMDLS-2 converges faster, compared to MGM-2, a synchronous distributed local search algorithm that converges to 2-opt solutions with similar quality. We demonstrate that the algorithm not only converges faster but also makes more economical use of the communication network and that the agents spend less time idle during the algorithm run. The results indicate that LAMDLS-2 is more suitable for realistic scenarios with message delays. Our approach, which is based on the ordered color scheme, allows the agents to be more active in computing their assignments and spend less effort in coordinating their actions. We also discussed how this approach can be extended to a general k-opt algorithm, which we intend to implement in future work.

References

1 Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

2 Leonid Barenboim and Michael Elkin. Combinatorial algorithms for distributed graph coloring. *Distributed Computing*, 27(2):79–93, 2014.

3 Muhammed Basharu, Ines Arana, and Hatem Ahriz. Solving DisCSPs with penalty driven search. In *Proceedings of AAAI*, pages 47–52, 2005.

4 Alessandro Farinelli, Alex Rogers, Adrian Petcu, and Nicholas R. Jennings. Decentralised coordination of low-power embedded devices using the Max-Sum algorithm. In *Proceedings of AAMAS*, pages 639–646, 2008.

5 Ferdinando Fioretto, Enrico Pontelli, and William Yeoh. Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research*, 61:623–698, 2018.

6 Ferdinando Fioretto, William Yeoh, and Enrico Pontelli. A multiagent system approach to scheduling devices in smart homes. In *Proceedings of AAMAS*, pages 981–989, 2017.

7 Tal Grinshpoun, Tamir Tassa, Vadim Levit, and Roie Zivan. Privacy preserving region optimal algorithms for symmetric and asymmetric DCOPs. *Artificial Intelligence*, 266:27–50, 2019.

8 Khoi D. Hoang, Ferdinando Fioretto, William Yeoh, Enrico Pontelli, and Roie Zivan. A large neighboring search schema for multi-agent optimization. In *Proceedings of CP*, pages 688–706, 2018.

9 Christopher Kiekintveld, Zhengyu Yin, Atul Kumar, and Milind Tambe. Asynchronous algorithms for approximate distributed constraint optimization with quality bounds. In *Proceedings of AAMAS*, pages 133–140, 2010.

10 Rajiv T. Maheswaran, Jonathan Pearce, and Milind Tambe. Distributed algorithms for DCOP: A graphical game-based approach. In *Proceedings of PDCS*, pages 432–439, 2004.

11 Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, and Pradeep Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In *Proceedings of AAMAS*, 2004.

12 Pragnesh J. Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1–2):149–180, 2005.

13 Arnon Netzer, Alon Grubshtein, and Amnon Meisels. Concurrent forward bounding for distributed constraint optimization problems. *Artificial Intelligence*, 193:186–216, 2012.

14 Duc Thien Nguyen, William Yeoh, Hoong Chuin Lau, and Roie Zivan. Distributed Gibbs: A linear-space sampling-based DCOP algorithm. *Journal of Artificial Intelligence Research*, 64:705–748, 2019.

15 Jonathan Pearce and Milind Tambe. Quality guarantees on k-optimal solutions for distributed constraint optimization problems. In *Proceedings of IJCAI*, pages 1446–1451, 2007.

16 Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of IJCAI*, pages 1413–1420, 2005.

17 Ben Rachmut, Roie Zivan, and William Yeoh. Latency-aware local search for distributed constraint optimization. In *Proceedings of AAMAS*, pages 1019–1027, 2021.

18 Ben Rachmut, Roie Zivan, and William Yeoh. Communication-aware local search for distributed constraint optimization. *Journal of Artificial Intelligence Research*, 75:637–675, 2022.

19 Pierre Rust, Gauthier Picard, and Fano Ramparany. Resilient distributed constraint reasoning to autonomously configure and adapt IoT environments. *ACM Transactions on Internet Technology*, 22(4):1–31, 2022.

20 Melanie Smith and Roger Mailler. Getting what you pay for: Is exploration in distributed hill climbing really worth it? In *Proceedings of IAT*, pages 319–326, 2010.

21 Meritxell Vinyals, Eric Anyung Shieh, Jesús Cerquides, Juan A. Rodríguez-Aguilar, Zhengyu Yin, Milind Tambe, and Emma Bowring. Quality guarantees for region optimal DCOP algorithms. In *Proceedings of AAMAS*, pages 133–140, 2011.

22 William Yeoh, Ariel Felner, and Sven Koenig. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research*, 38:85–133, 2010.

23 Makoto Yokoo and Katsutoshi Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of AAMAS*, pages 401–408, 1996.

24 Weixiong Zhang, Guandong Wang, Zhao Xing, and Lars Wittenburg. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence*, 161(1-2):55–87, 2005.

25 Roie Zivan and Amnon Meisels. Message delay and DisCSP search algorithms. *Annals of Mathematics and Artificial Intelligence*, 46:415–439, 2006.

26 Roie Zivan, Steven Okamoto, and Hilla Peled. Explorative anytime local search for distributed constraint optimization. *Artificial Intelligence*, 212:1–26, 2014.

27 Roie Zivan, Ben Rachmut, Omer Perry, and William Yeoh. Effect of asynchronous execution and imperfect communication on max-sum belief propagation. *Autonomous Agents and Multi-Agent Systems*, 37(2), 2023.

Combining Constraint Programming Reasoning with Large Language Model Predictions

Florian Régin
Université Côte d'Azur, I3S, CNRS, Sophia Antipolis, France

Elisabetta De Maria
Université Côte d'Azur, I3S, CNRS, Sophia Antipolis, France

Alexandre Bonlarron
Université Côte d'Azur, Inria, Sophia Antipolis, France
Université Côte d'Azur, I3S, CNRS, Sophia Antipolis, France

Abstract

Constraint Programming (CP) and Machine Learning (ML) face challenges in text generation due to CP's struggle with implementing "meaning" and ML's difficulty with structural constraints. This paper proposes a solution by combining both approaches and embedding a Large Language Model (LLM) in CP. The LLM handles word generation and meaning, while CP manages structural constraints. This approach builds on GenCP, an improved version of On-the-fly Constraint Programming Search (OTFS) using LLM-generated domains. Compared to Beam Search (BS), a standard NLP method, this combined approach (GenCP with LLM) is faster and produces better results, ensuring all constraints are satisfied. This fusion of CP and ML presents new possibilities for enhancing text generation under constraints.

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

Keywords and phrases Solver and Tools, ML-augmented CP, Constrained Text Generation, ML alongside CO

Digital Object Identifier 10.4230/LIPIcs.CP.2024.25

Acknowledgements We thank Jack Massey for his help in reproducing the benchmarks used as baseline in Section 4.1.1.

1 Introduction

How can we perceive Constraint Programming beyond its traditional role in solving combinatorial optimization problems? Once Eugene Freuder wrote *Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it* [13].

Nevertheless, some real-world problems are still beyond the reach of the current CP paradigm. This is particularly true when real-world problems involve vague notions such as "meaning" and "melody" for text and music. These are not easy to model in CP with the classical toolbox, mainly because these notions are hard to define formally. For instance, it is unclear how to formalize an objective function or a constraint to get closer to a meaningful sentence, a melodious song or a captivating painting. On the other hand, recent results in Machine Learning (ML), such as transformer-based models [39], have demonstrated the power of these techniques to capture a significant part of these vague concepts through data-driven statistical learning (e.g., Large Language Model (LLM) like the GPT series [8], stable-diffusion [33], ChatMusician [41]). In the article, we demonstrate that ML, and in particular LLM, can help CP to model and solve problems where such vague concepts can be found.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 25; pp. 25:1–25:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In recent years, there has been a growing interest in text generation under constraints thanks to the rise of transformer-based models, like OpenAI ChatGPT ([8]) and Meta LLaMa ([37]). Nevertheless, even fine-tuned prompted LLMs fail to generate several constrained outputs (see the tasks introduced in [40]). The goal of this paper is to present a new method for the task of text generation under constraints. This interest has a strong chance of continuing to grow insofar as many brands wish to integrate these technologies, in particular with their customers, and want to have control and guarantees on the behavior of these conversational agents. Hence, it may impact several critical marketing aspects (e.g., brand representation, legal issues, data privacy, . . .). Therefore, CP has the potential to become a strong safeguard of this kind of generative model.

For the task of text generation under constraints, ML techniques face limitations when they have to manage structural constraints, such as limits on the number of words or characters (e.g. Text Summarization, Text Simplification, Text style transfer, Question Answering, Storytelling, Poetry or Lyrics Generation, Subtitle) [15]. CP succeeds on these types of constraints, making the combination of CP and ML a natural fit for the task of text generation under constraints.

This paper proposes such a combination, to tackle a class of problems where neither CP and ML succeeds on their own (Fig. 1).

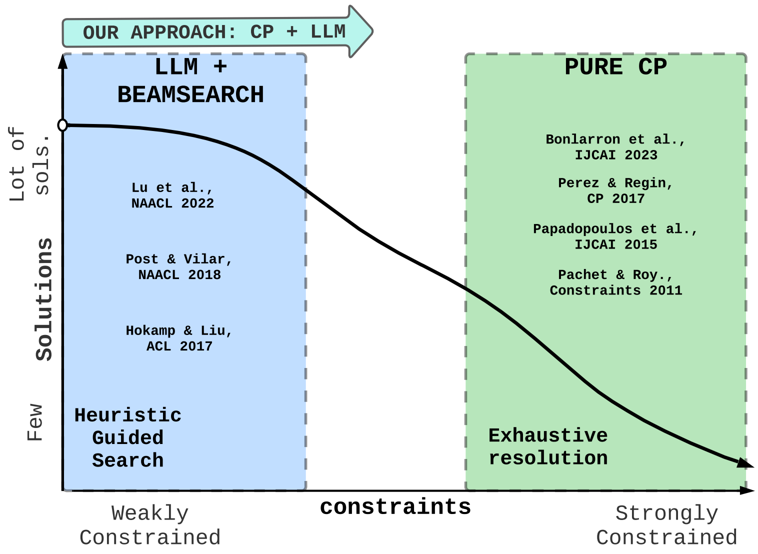


Figure 1 Our approach aspires to explore the in-between area. In the blue (left-hand side) region, LLM guided searches solve weakly constrained problems [27, 32, 17] and in the green (right-hand side) region, CP-based generation tackles strongly constrained problems [36, 5, 6, 4, 31, 30, 29].

Combining Combinatorial Optimization (CO) and ML is a very active research area [2], however there is no easy way to integrate the ML "expertise" into CP as a constraint of the model [1, 26] and *vice versa* [18]. Furthermore, there are many incentives to strengthen the interactions between ML and CO [21, 22, 35]. Usually, the main motivation comes from the performance perspective, where the idea is to improve a solver's performance with ML (e.g., finding branching heuristics thanks to Reinforcement Learning [9] or finding better bounds with Clustering [28]). This paper tackles it from the modeling point of view. Modeling is a crucial part of CO works. In the end, the model must account for the underlying solver that runs it. More in detail, here, the paper focuses on the interaction between CP and ML, more precisely through an ML-augmented CP approach [23].

In the context of text generation under constraints, the domain of a variable represents a word. The base idea of the paper consists in letting ML manage the domain of variables and CP manage the constraints and the number of variables. In this manner, the sentence formed by variables has high chances to have a meaning and all the constraints will be satisfied. In traditional CP, the domains can not be managed by ML because they have to be set beforehand. However, it is possible to rely on On-the-fly Constraint Programming Search (OTFS) [34], a CP based method where variables, domains and constraints are generated during the search for a solution.

The main contribution of this paper is to propose a new version of OTFS, called GenCP, where the generative function of the domain of variables is modified to allow CP variable domains to be computed by an LLM embedded in it, during the search for a solution. More in detail, ML is used during process solving but it is also used as an explicit part of the problem definition (i.e., domains are predicted by the LLM and can replace entirely static variable domains definition of a CSP.). Thus it bridges CP and ML through solving and modelling.

The potential of the approach is showcased for the problem of text generation under constraints, against one the most used techniques in the Natural Language Processing (NLP) field: Beam Search (BS). Both methods (BS and GenCP) are compared on constrained sentences generation tasks extracted from benchmarks recently introduced [40]. The approach highlights how CP can provide guarantees when combined with LLM predictions.

The paper is organized as follows: Sec. 2 serves as background, Sec. 3 shows how to extend OTFS to GenCP and how to implement an interaction between GenCP and LLM. Sec. 4 presents the experimental results in which the new approach is demonstrated on the task of text generation under constraints. Finally, Sec. 5 delves into further discussion, offering additional insights into this work and providing perspectives for future research endeavors.

2 Background

This section introduces the necessary background on LLM and CP.

2.1 LLM Predictions Strategies

2.1.1 Decoding Strategies Combined with LLMs

Large Language Models (LLMs), such as the GPT series, generate text by predicting the next token (word or character) given the history of previously generated words. Decoding in LLMs refers to the strategy used to select the next words to be generated.

2.1.2 Greedy Decoding

The simplest decoding strategy is greedy decoding. Here, the LLM selects the words with the highest probability at each time step. Although simple and efficient, this approach does not guarantee the best overall sequence, as it does not consider the effect of the current selection on future tokens.

2.1.3 Beam Search

Beam Search (BS) [25, 32, 17] is a refined version of greedy decoding. A beam is a candidate sequence of words. Instead of selecting the single best token at each time step, it usually keeps track of the k most likely sequences (beams) at each step.

Although BS usually achieves better results than greedy decoding, it assumes that high-ranking token sequences consist of high-ranking tokens, which may only sometimes be the case. For a more stochastic and diverse output, top-k sampling and top-p sampling (also known as nucleus sampling) are used. In top-k sampling, the model selects from the top k highest probability predictions, while in top-p sampling, it dynamically selects the number of top predictions to cover p percent of the total probability mass.

2.1.4 Perplexity

Perplexity is an entropy metric derived from Shannon's information theory [7]. Since an LLM computes the probability of text, then it can compute text perplexity. It can be expressed as the geometric mean of the inverse conditional likelihood of the sequence [20]. Let S_n be the sequence of a succession of words of size n: $S_n = w_1w_2..w_n$. The perplexity (PPL) of S_n is computed as follows:

$$PPL(S_n) = \sqrt[n]{\frac{1}{P(w_1w_2w_3...w_n)}},$$

where probability $P(\cdot)$ is given by the LLM. PPL can be interpreted as the "how likely a text is generated by a given model" [15]. Usually, it is used to evaluate the LLM itself by checking that good samples are recognized as such (i.e., low PPL values).

In NLP, the evaluation of text is still an open problem, and human evaluation remains the gold standard. Numerous metrics have been developed to address this issue. Among them, PPL remains an objective criterion associated with text produced by a given model. PPL is also much more convenient to use than pure probability. Its range is $[1; +\infty[$. The lower, the better.

2.2 Constraint Programming

Constraint Programming (CP) is a paradigm for solving combinatorial problems that draws on a wide range of techniques from artificial intelligence and operations research. In CP a problem can be defined as a Constraint Satisfaction Problem (CSP). A CSP is a triplet: $\langle X, D, C\rangle$, where:

- $X = \{X_1, X_2, ..., X_n\}$ is the set of variables of the problem.
- $D = \{D_{X_1}, D_{X_2}, ..., D_{X_n}\}$ is the set of domains, where each domain D_{X_i} corresponds to the set of possible values for the variable X_i.
- $C = \{c_1, c_2, ..., c_m\}$ is the set of constraints of the problem. A constraints represent a property of the problem.

A solution is an assignment of all the variables to a value present in their respective domains, such that all the constraints are satisfied.

2.2.1 Avoiding Static Definition of the CSP

In traditional CP, for the task of text generation under constraints, a variable represents a word. Since the domains of variables have to be set beforehand, they will be of enormous size, containing every word/declination of words for a given language. Furthermore, constraints between succession of words may lead to a combinatorial explosion. Since traditional CP is not well suited, this work focuses on OTFS, a CP based method recently introduced by Régin and De Maria [34]. Instead of having the variables/domains/constraints set before the search, OTFS generates the variables/domains/constraints during the search for a solution, avoiding the problem stated above and being expendable to permit the integration of an LLM. The new version of OTFS is called GenCP.

3 Method: LLM alongside OTFS

The approach of this paper extends OTFS by having an embedded LLM generate the domains of variables. Figure 2 graphically depicts the interplay between those components. The approach also adds a minor improvement in the form of helping functions, to differentiate between implicit constraints (prevent infinite loops, ensure a variable represents a word, etc.) and explicit constraints (constraints of the problem). In the next subsection, the new version of OTFS called GenCP is described.

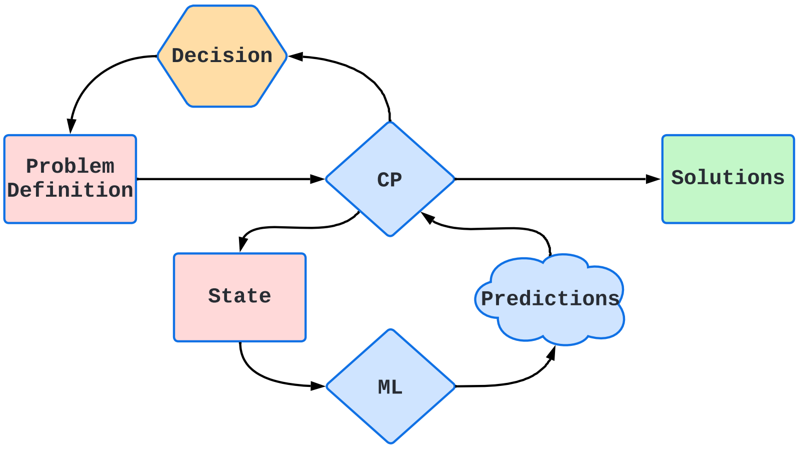


Figure 2 This scheme presents the integration of ML into CP performed by GenCP. It is freely inspired by Sec. 3.2.3 of Bengio et al.'s survey [2], which introduces an architecture for ML alongside Optimization Algorithms. The similarity is highlighted because the master algorithm (here, GenCP) repeatedly queries an ML model (here, an LLM) to obtain a prediction as a subroutine. In the context of this paper, the decision (search or propagation) has an impact on the problem definition (the CSP) because it may generate new variables, domains, or constraints during the solving process. The state is the current assignment of the variables.

3.1 New version of OTFS: GenCP

In traditional CP it is not common to generate new variables/domains/constraints during the search, while OTFS is based on this idea. OTFS begins with an empty or partially defined CSP (the CSP has less variables/domains/constraints than the CSP in traditional CP) and will generate variable/domains/constraints during the search for solutions.

GenCP is a new version of OTFS that makes two changes to the original version: **1)** the function that generates the domain *genD* calls an LLM to generate the domain of the current variable. **2)** Helping functions are added to represent implicit constraints.

Here is GenCP applied to text generation under constraints. An GenCP model can be defined as a pair of sets $\{\mathcal{M}, \mathcal{F}\}$, where:

- $\mathcal{M} = \{X, D, C\}$ represents the model of the problem.
- X represents the variables. The variables represent words.
- D represents the domain of the variables. A domain $d_i \in D$ contains a list of predicted words by an LLM.
- C represents the explicit constraints (constraints of the problem). A constraint $c_i \in C$ represents rules over text (e.g., number of words, number of characters, forbidden words, or symbols).
- $\mathcal{F} = \{G, B, H\}$ is a set of functions.
- G represents the set of generative functions: these functions explain to the solver how to generate variables/domains/constraints.
- B represents the set of Boolean functions: these functions tell the solver when a solution is found.
- H represents the set of helping functions: these functions are used to represent implicit constraints, for example ensuring that when a variable is generated, it helps obtaining a solution (to prevent the solver from attaining an infinite loop of generating variables).

3.1.1 Generative Functions

The set of generative functions $G = \{genV, genD, genC\}$ is such that:

- $genV$ generates a new variable with an empty domain and adds it to X.
- $genD$ calls the LLM with the current sentence formed by the model and sets the domain of the previously generated variable to the output.
- $genC$ generates the constraint(s) relevant to the current variables of the model to C. The constraints generated depend on the problem (e.g., generate a sentence that does not contain the letter “e”).

3.1.2 LLM integration

A variable is generated with an empty domain. To generate the domain of variables, $genD$ calls an LLM using $callLLM(sentence, parameters, k)$, where:

- $sentence$ is the current sentence represented by the variables of the model.
- $parameters$ represents sampling parameters ($top\_k, top\_p...$). For this paper, top_k is used exclusively for both GenCP and BS: the LLM answers k words ranked by probability, highest to lowest.
- k is the number of words asked to the LLM.

Since the parameters and k are not modified after the definition of the model, $callLLM(sentence, parameters, k)$ will be simply referred to as $callLLM(sentence)$.

3.1.3 Helping Functions

Helping functions represent implicit constraints, like avoiding infinite loops. In our current implementation, the set of functions H contains the following functions:

- H_o: it orders the domain of variables depending on the problem.
- $H_{onlyWords}$: it ensures that any word predicted by the LLM is a complete word and not the suffix or prefix of a word and it filters out any symbol or special character.

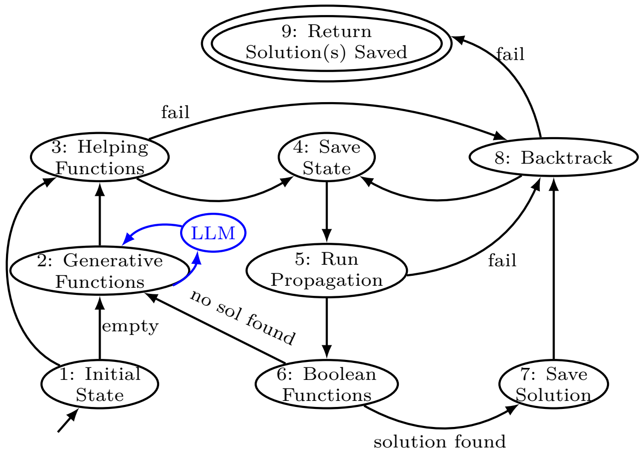


Figure 3 This graph illustrates the main steps in GenCP solving.

Algorithm 1 GenCP($\mathcal{M}, \mathcal{F}$), $\mathcal{M} = \{X, D, C\}$, $\mathcal{F}$ contains the generative and boolean functions.

```
1: $\mathcal{S} = \emptyset$; **if** $\mathcal{M}$ is not empty **then** go to **3.**;
2: generativeFunctions($\mathcal{M}$);
3: helpingFunctions($\mathcal{M}$); **if** $\mathcal{M}$.X.$containsEmptyVariable()$ **then** go to **8.**;
4: saveState($\mathcal{M}$);
5: propagation($\mathcal{M}$); **if** $\mathcal{M}$.X.$containsEmptyVariable()$ **then** go to **8.**;
6: **if** *not booleanFunctions($\mathcal{M}$)* **then** go to **2.**;
7: $\mathcal{S}$.add($\mathcal{M}$);
8: **if** $backtrack(\mathcal{M})$ **then** go to **4.**; **else return** $\mathcal{S}$;
```

3.1.4 Description of the new approach

The main steps of GenCP are depicted in Fig. 3 and Algorithm 1:

1. GenCP begins with an initial state. If the initial state is empty, the generative functions are called (**2.**), otherwise the helping functions are called (**3.**).
2. The generative functions *genV*/*genD*/*genC* are called (*genD* calls the LLM).
3. The helping functions are called to manage implicit constraints, backtracking if necessary (e.g., if the LLM generated an empty domain).
4. The current state of the model $\mathcal{M}$ is saved.
5. The propagation is called, if it fails the model backtracks (8.), else it calls the boolean functions (6.).
6. The Boolean functions are called to check if a solution has been found. If a solution is found, it is saved (7.) and the model backtracks (8.), otherwise the model calls the generative functions (2.).
7. The current sentence formed by the variables is saved as a solution.
8. GenCP backtracks to a previously saved state (4.) of the model and changes the choices made during propagation (5.). If no previous state was saved, then backtracking fails (9.).
   - When backtracking to a previously saved state, the model deletes all variables, their respective domains, and the constraints associated with them, that are not present in the previously saved state.
9. GenCP outputs the solution(s) that were saved or it indicates that no solution was found.

3.1.5 Enforce variability

Variability between two sentences is the number of words that are not equal at each position, for example:

- "The little boy is" and "The little cat is" have a variability of 1.
- "My name is John" and "John is my name" have a variability of 4.

To force a greater variability between solutions (greater than 2), a special backtrack called $backtrackTo(n)$ is used. Let the set of variables $X = \{x_1, \ldots, x_n, x_{n+1} \ldots, x_m\}$. The function $backtrackTo(n)$ deletes the variables x_{n+1} to x_m and causes a backtrack. For example, consider the sentence "I like to swim in the summer.". With $backtrackTo(2)$, "to swim in the summer." is deleted and the value of variable $x2 = $ "$like''$ is changed. The next solution might be "I want to break free.".

3.1.6 Ordering

For some tasks, not following the ordering strategies of the LLM (like top-k and top-p) can lead to better/faster solutions. Two other orderings are considered: PPL valuation and length of a word (depending on the average word length in the given language).

3.2 Modeling Example

Here is a simple example of how the search of GenCP works: for this paper the generative functions only generate variables one at a time but it is important to note that these functions can generate multiple variables, domains and constraints at once. Let us suppose GenCP has to generate a sentence beginning by "The" and containing between 10 and 15 words with exactly 60 characters. The following functions are needed:

- $currentSentence(\mathcal{M})$: outputs the current sentence the variables form.
- $callLLM(sentence)$: described in 3.1.2. Here k is equal to 10 (each time the LLM is called, it will output 10 words).
- $contains(sentence, word)$: outputs yes if the sentence contains the word and no otherwise.
- $nbChar(sentence)$: outputs the number of characters in the sentence.

The obtained model is $\{\mathcal{M}, \mathcal{F}\}$, where:

- $\mathcal{M} = \{X, D, C\}$:
- $X = \{x_1\}$.
- $D = \{d_1 = \{\text{“The”}\}\}$.
- $C = \emptyset$.
- $\mathcal{F} = \{G, B, H\}$.
- $G = \{genV, genD, genC\}$ is a set of functions, each function follows these steps:
  - generate $x_{|X|+1}$ and add it to X with an empty domain $d_{|X|+1}$.
  - $d_{|X|+1} = callLLM(currentSentence(\mathcal{M}))$.
  - $c_{remove_{over60char}((currentSentence(\mathcal{M}), d_{|X|+1})}$.
  - The constraints remove the words that make the current sentence exceed 60 characters from the domain of the current variable.
- $B = \{endNbWords, endNbCharacters, endLLM\}$ is a set of functions, each function behaves as follows:
  - $|X| >= 10 \land |X| <= 15$.
  - $nbChar(currentSentence(\mathcal{M})) == 60$.
  - $contains(callLLM(currentSentence(\mathcal{M})), \text{“.''})$.

- $H = \{H_{ho}\}$:
  - $H_{ho} : order(d_{|X|+1})$.
  - To help attain the goal of 60 characters, the domain of the current variable is ordered such that before the 10th word the solver tries the longer words first and at the 10th word the solver tries the shorter words first.

With the above representation of the problem, GenCP is asked for 4 solutions, $backtrackTo(2)$ is used and the LLM is asked for 10 words maximum per call. The obtained solutions are:

1. The following is an article by the author of the above book.
2. The first time you see the movie version of your book on TV.
3. The New York Times has an article on the new book by Tim Wu.
4. The new year is here and we are ready to make the next step.

3.3 Illustrated Example

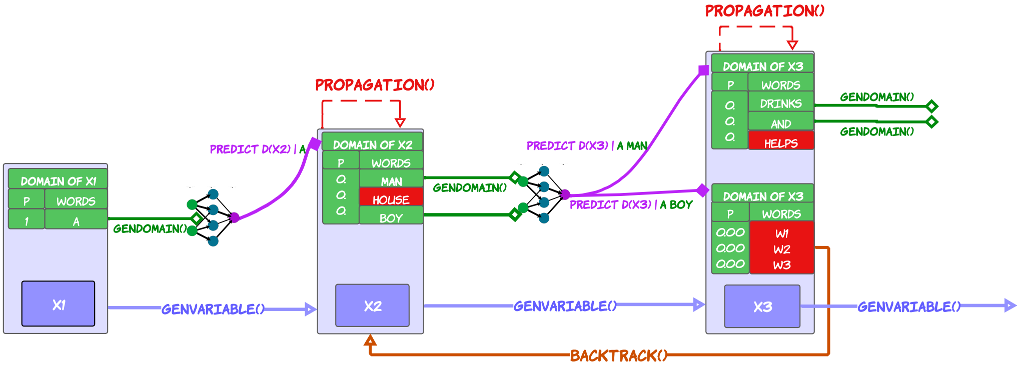


Figure 4 Illustrations of GenCP as a simplified framework with three variables and predictions of 3 values per LLM call, on a simple problem: generate a sentence that does not contain the letter e. For each variable, the predefined constraint c_i "the letter e is forbidden" is generated. A predefined domain with one word is defined for the first variable: A. The current sentence formed by the variable "A" is not a solution ($callLLM(\text{``}A'')$ does not answer a period (".")), so a new empty variable x_2 is generated. GenCP calls the LLM with the sentence "A" to predict the domain of x_2. The LLM model predicts three values: [man, house, boy]. c_2 is generated, causing the domain of x_2 to be filtered accordingly: house is removed. x_2 is then assigned to boy, GenCP generates the variable x_3 and calls the LLM with the sentence "A boy" to predict a new domain. Unfortunately, the domain of x_3 is empty, either because the LLM answered an empty domain or because this domain was entirely filtered during propagation. Hence, GenCP backtracks to x_2 and the value of x_2 is changed to man. In the same fashion as before, GenCP generates the variables x_3, and gives "A man" to the LLM that predicts: [drinks, and, helps]. c_3 is generated, filtering helps because it contains an e. The process continues until the domain of the next predicted variable contains a period (a solution is found) or the solver *fails*.

Fig. 4 illustrates GenCP as a simplified framework with three variables and predictions of 3 values per LLM call, on a simple problem: generate a sentence that does not contain the letter e.

4 Experiments

4.1 Experimental Conditions

4.1.1 Baseline

The experiments presented by Yao et al. are partially reproduced [40]. In particular, the constrained sentence generation tasks described in Tab. 1. Five LLMs were selected: GPT4, GPT4-O, Mistral Next, Claude 3.5, and Gemini. The four LLMs are prompted with the same example command given in [40]. For example, "Please create a sentence of exactly 82 characters." for the Sent-1 task[1]. Tab. 2 gives an overview of the performance of the five LLMs on the four tasks. The satisfaction rate is based on ten trials per task per model. In addition, Tab. 2 also shows that the LLMs perform well on the lexically constrained scenario task-4 with a 90+% satisfaction rate over ten trials. Also, as Yao et al. previously showed in their paper, LLMs struggle to produce constrained sentences involving counting (e.g., words and characters). They provide a nice picture of current LLM satisfaction capabilities by introducing new benchmarks. Unfortunately, the Yao et al. article only provides the benchmarks and some hints on reproducing them. However, it does not give any hints on how to solve the tasks associated with the benchmarks (see the original article for more details [40]).

Table 1 Four tasks on sentence generation used to compare BS and GenCP extract from [40].

name	words count	character count	lexical constraints
sent-1		$= 82$	
sent-2	$= 10$		$X_3 = \text{soft},\ X_7 = \text{soft}, X_{10} = \text{math}$
sent-3	≥ 20	$\forall i, \lvert X_i \rvert \leq 6$	
sent-4			soft, beach, math

Table 2 Number of successes (#s), Number of fails (#f) and satisfaction rate (%sat) for each model (GPT-4, GPT-4.0, Mistral Next, Claude 3.5, Gemini) for each task (sent-1, sent-2, sent-3, sent-4).

name	GPT-4			GPT-4.0			Mistral Next			Claude3.5 Sonnet			Gemini		
	#s	#f	%sat	#s	#f	%sat	#s	#f	%sat	#s	#f	%sat	#s	#f	%sat
sent-1	1	9	10%	0	10	0%	0	10	0%	1	9	10%	0	10	0%
sent-2	0	10	0%	0	10	0%	0	10	0%	0	10	0%	0	10	0%
sent-3	1	9	10%	5	5	50%	0	10	0%	9	1	90%	1	9	10%
sent-4	9	1	90%	9	1	90%	10	0	100%	10	0	100%	1	9	10%

4.1.2 Hardware & Implementation

The experiments were performed on a laptop with Windows 10 Professional, 32 GB RAM, and Intel 16 CPU cores. The approach and the BS are implemented in Java 17 for easier comparisons.

[1] https://chatgpt.com/share/b2834735-f7d8-468a-ba54-7da19dd6723c

4.1.3 LLM choice

LLaMa [37] is responsible for the predictions of words as domains for the variables, mainly because an efficient implementation in C++ was recently released[2]. It allows running a model on a personal laptop and CPU (only) efficiently. Thanks to quantization [16] (model weight compression), the 7B parameters model (in Float16) of 13GB original size, in 4-bit quantization (Q4) drops to 3.9GB of RAM. However, the biggest model of LLama 65B (120GB), even in Q4, needs 38.5 GB of RAM. Thus, the LLaMa v1 model used in the experiments is LLaMa 7B Q4 with low temperature (i.e., ≤ 1, temp $= 0.8$).

When asked for k words, this version of LLaMa will take the same amount of time to ouput 1 word and 1000 words. To minimize the importance of k, *callLLM* outputs more than k words, a beam/variable only keeps k “valid” words. A “valid” word is a word that does not violate a constraint on its own. For example, a word that does not violate the constraint “does not contain the letter e”.

4.1.4 Beam Search Technical Remarks

In the current implementations two halting conditions are defined for BS:

- First solution: when the current beam contains at least one solution, BS is stopped and output the solutions.
- All solutions: when the current beam contains at least one solution but another beam can continue to generate words without violating a constraint (for instance, it does not contain enough characters to satisfy a length constraint), the beam solutions are saved and BS continues with the remaining beams.

4.1.5 Benchmarks Settings

BS and GenCP are compared on some recent benchmarks described in Sec. 4.1.1.

To guarantee GenCP and BS to be judged on the generation of sentences of the same quality, a solution is a sentence that satisfies all the constraints of the current task and, when given this sentence, the LLM predicts a period (“.”). Not to alter BS too much, words are ordered by probability (PPL is not used) and, since BS sentences have low variability, GenCP is used without $backtrackTo(n)$.

BS and GenCP are compared on the following criteria:

- Time in seconds.
- Number of solutions. GenCP was stopped when it found the same number of solutions as BS on a task. 0/1 means that BS found no solution while GenCP found one solution.
- The ratio solutions/outputs as a constraint satisfaction rate.
- For BS only, the number of bad outputs (number of outputs that are not a solution).
- For GenCP only, the number of backtracks.

4.2 Result Analysis

The results show that GenCP can be used to solve efficiently text generation under constraints problems. GenCP is faster than BS and all the outputs are solutions, contrary to BS where some outputs are not solutions.

[2] `https://github.com/ggerganov/llama.cpp`

Although the results suggest that GenCP succeeds in all tasks (see Tab. 3), it becomes particularly interesting when considering size constraints (e.g., sentences with a precise number of words or characters). It obtains sentences that satisfy the constraint with a low PPL score on sent-1 and sent-3 tasks.

GenCP also succeeds in producing sequences obeying lexical constraints in sent-2 and sent-4. However, the PPL and a human evaluation on these sentences show a substantial deterioration in term of quality (i.e., meaningfulness).

Therefore, regarding sent-1 and sent-3 tasks, GenCP is to be preferred, whereas for sent-4 and sent-2 tasks, LLMs prompted alone or joint with BS is still adequate.

Table 3 Comparison of BS and GenCP on the tasks of Tab. 1. Task considered (sent-i), Number of solutions (#sols), Time in seconds (s), Number of bad output (#badoutput), satisfaction rate (%sat) and Number of backtracks (#bk).

Experiments			BS			GenCP		
sent-i	k	#sols	s	#badoutput	%sat	s	%sat	#bk
1	5	1	108	9	10%	103	100%	45
	10	0	182	18	0%	177	100%	84
	20	1	399	58	1%	46	100%	13
	50	1	1123	109	≈ 0%	47	100%	13
2	5	5	34	0	100%	38	100%	38
	10	10	69	0	100%	36	100%	25
	20	20	140	0	100%	58	100%	40
	50	49	354	1	99%	134	100%	100
3	5	0	248	5	0%	36	100%	4
	10	2	510	8	20%	55	100%	6
	20	4	1030	16	20%	164	100%	38
	50	20	2633	30	66%	1174	100%	374
4	5	25	279	3	89%	308	100%	118
	10	30	513	8	78%	311	100%	114
	20	45	1123	14	76%	321	100%	104
	50	89	2928	40	68%	388	100%	27

Table 4 GenCP results for $k = 50$ when given approximately the same amount of time as BS in Tab 3. Number of solutions (#sols), Time in seconds (s), Memory usage in megabytes (MB), Number of backtracks (#bk).

Experiments			GenCP		
sent-i	k	#sols	s	MB	#bk
1	50	2	1123	136	79
2	50	355	222	208	222
3	50	488	2633	378	624
4	50	830	2929	676	680

4.2.1 Beam Search

BS and GenCP are compared in Tab. 3. In all tables, the number of backtracks is denoted by #bk. BS is slower than GenCP and has lower satisfaction rate (number of outputs that are solutions / total number of outputs), denoted by %sat. This is due to multiple facts:

1. Beam Search can not guarantee to find every solution.
2. Beam Search chooses the next word depending on the probability of the LLM.
3. At each step, BS considers k sentences, each sentence asks k words to the LLM, so each step considers k^2 words. BS orders these words decreasingly by probability and only keeps the k first.

Facts 2 and 3 explain why increasing k does not guarantee to find the same/more solutions, it might even cause BS to find less solutions.

Let us suppose $k = 5$, BS found one solution, and at depth 4, the candidate needed to find this solution was ranked 5 out of 25. Let us suppose now k is increased to 6: at each step BS will consider 36 candidates and take the 6 best ones. BS considers 11 more candidates than with $k = 5$; if at depth 4, the candidate needed to find the previous solution is now ranked 7 instead of 5, BS will not consider it and $k = 6$ will not find the solution found with $k = 5$.

4.2.2 GenCP

Table 5 Output sentences of GenCP on the experiments of Tab 1 associated with the task (sent-i), k, *backtrackTo* (*bkTo*), and Perplexity (PPL). In sent-4* a constraint was added so that “soft”, “beach”, “math” have to be separated by at least three words. Sentences with high perplexity were chosen to showcase the importance of low perplexity.

Experiments				GenCP
sent-i	k	$bkTo(n)$	PPL	sentence generated
1	50	NO	8	The following is an article by Dr David Hillon the subject of the role of prayer.
		2	13	The New York Times has an article on the new book by former President George Bush.
		3	6	The following information is taken from the website of the National Park Services.
2	50	NO	189	The following soft skills are required beach resort jobs math.
		2	169	The National soft drink association has beach balls and math.
		3	107	The most soft and comfortable of beach wear is math.
3	50	NO	8	The first time you see the movie The Big Short is like being hit by an ice cube in the face.
		2	5	The world is full of great ideas and the best way to get them out there is by using the power of the web.
		3	5	The first step in the right path is to know what you want and where you are going in life.
4*	50	NO	347	The following is an article by Dr math and science teacher beach high school in soft.
		2	593	The term of the contract is for math and science teachers beach to be able soft.
		3	48	The following data is based on the math and physics of beach waves and the soft sand.

Tab. 4 shows the capability of GenCP to generate more solutions than BS. GenCP is given the same time as BS for the same task and $k = 50$, GenCP obtains more solutions than BS. Note that for sent-1, without *backtrackTo* GenCP only obtains 2 solutions in 1123 seconds, while with *backtrackTo*(6) GenCP obtains 11 solutions in 1123 seconds.

The LLM-enhanced GenCP avoids the drawbacks of BS and proposes an alternative approach to text generation under constraints for the following reasons:

- GenCP can guarantee to find every solution (if any). Increasing k guarantees to find at least the same solutions previously found and potentially finds new solutions. Furthermore, it can offer more solutions than BS.
- All the outputs answered by GenCP are solutions (all the constraints are satisfied).
- GenCP offers more options for improvement, for example to ensure better variability (*backtrackTo* explained in 3.1.5 can be used) or other orderings than probability (3.1.6).

4.2.3 Variability and Perplexity

Tab. 5 demonstrates the importance of enforcing variability and perplexity. When GenCP generated solutions for Tab. 3 and 4, the maximum variability was 4. Tab. 5 shows that with $backtrackTo(2)/backtrackTo(3)$, sentences generated are almost completely different thanks to high variability (10+ for sent-3 for example).

Tab. 5 purposefully contains sentences with high perplexity to illustrate that this leads to a degradation in the sentence quality (i.e., low meaning).

All the sentences generated for sent-4 had the words “soft”, “beach” and “math” next to each other. To showcase the capability of GenCP to improve sentences, sent-4* was created: it is the same as sent-4 except that “soft”, “beach” and “math” must contain at least three words between them.

5 Discussion & Perspectives

5.1 GPU and CPU Interplay

The article shares a proof-of-concept showing that interesting results can be obtained using CPU resources combined with a small quantized LLM in a CP solver. However, LLMs, in general, work best with much larger computational resources and require GPU resources. Even though smaller models (e.g., Mistral 8x7B) sometimes manage to take top places in specific scenarios. The top spots in the LLM Elo rankings feature gigantic models [10]. Given their size, clusters of GPU are quickly mandatory. Hence, it would be interesting to study in more detail how the joint use of resources (for instance, CPU for solver and GPU for LLM) could improve the results of the paper and correspond to more real-world usage in industry.

5.2 Token Management

In this article, GenCP ignores tokens and works at the word level (pre-token). It is possible to handle tokens by adapting the problem modeling. Indeed, it is possible to consider a word as a meta-variable X_1 composed of several decision variables (e.g., X_{1_1}, X_{1_2}, X_{1_3}...). This is useful and straightforward, as it is not clear in advance how the tokenizer will cut the words. For instance, let us consider the following sentence: *The first step in the recruitment of a new hire is to make sure that the job requisition is clear.* Let us look at the assignments of the variables (space separates meta-variables, and semicolon decision variables): *The; first; step; in; the; rec;ruit;ment; of; a; new; h;ire; is; to; make; sure; that; the; job; requ;is;ition; is; clear;.* The word *recruitment* needs three decision variables because it is composed of three tokens (i.e., *rec, ruit* and *ment*). It is easy to manage in GenCP because it can generate as many variables as required. Nevertheless, the evolution of the CSP (generation of variables and domains) is rather technical and, therefore, depends on the tokenizer.

5.3 CSP Modeling

The idea that a CSP can evolve in response to external information is not new (e.g., Dynamic Constraint Network [12]). This dynamic vision of CSPs has been motivated by several real-world problems, particularly in product configuration [19]. GenCP proposes ML integration in modeling by letting LLMs manage operations for CSP domains during the resolution process. The “outside the world” information [3] is given by the LLM. The article shows that LLMs can contribute to CSP modeling for generation tasks. However, how ML/LLMs can be used for CSP modeling in general for any problem remains an open problem [14, 24, 38, 11].

6 Conclusion

This paper showed that combining CP solving of structural constraints and ML understanding of vague notions (like meaning) on the task of text generation under constraints obtains promising results. This paper presents GenCP, a new method that extends OTFS to make the domains manageable by LLM predictions. The results show that GenCP can generate meaningful sentences that ensure various properties like the number of words, number of characters, mandatory keywords, or some forbidden characters. The results also show that GenCP has 100% satisfaction rate and takes less time to output solutions of the same quality than a well-known technique in the field of text generation under constraints: Beam Search. GenCP provides multiple improvements thanks to ordering, enforcing variability and perplexity, allowing thus to obtain overall higher quality solutions than BS.

References

1 Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. Neuron constraints to model complex real-world problems. In *Principles and Practice of Constraint Programming–CP 2011: 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011, Proceedings*, volume 6876, page 115. Springer, 2011.

2 Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021. doi:10.1016/j.ejor.2020.07.063.

3 Christian Bessière. Arc-consistency in dynamic constraint satisfaction problems. In Thomas L. Dean and Kathleen R. McKeown, editors, *Proceedings of the 9th National Conference on Artificial Intelligence, Anaheim, CA, USA, July 14-19, 1991, Volume 1*, pages 221–226. AAAI Press / The MIT Press, 1991. URL: http://www.aaai.org/Library/AAAI/1991/aaai91-035.php.

4 Alexandre Bonlarron, Aurélie Calabrèse, Pierre Kornprobst, and Jean-Charles Régin. Constraints first: a new mdd-based model to generate sentences under constraints. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pages 1893–1901, 2023.

5 Alexandre Bonlarron and Jean-Charles Régin. Intertwining cp and nlp: The generation of unreasonably constrained sentences. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, 2024. To appear.

6 Alexandre Bonlarron and Jean-Charles Régin. Markov constraint as large language model surrogate. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, 2024. To appear.

7 Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, Jennifer C. Lai, and Robert L. Mercer. An estimate of an upper bound for the entropy of English. *Computational Linguistics*, 18(1):31–40, 1992. URL: https://aclanthology.org/J92-1002.

8 Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

9 Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre A. Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(5):3677–3687, May 2021. doi:10.1609/aaai.v35i5.16484.

10 Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. Chatbot arena: An open platform for evaluating llms by human preference, 2024. arXiv:2403.04132.

11 Parag Pravin Dakle, Serdar Kadıoğlu, Karthik Uppuluri, Regina Politi, Preethi Raghavan, SaiKrishna Rallabandi, and Ravisutha Srinivasamurthy. Ner4opt: Named entity recognition for ;optimization modelling from ;natural language. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 20th International Conference, CPAIOR 2023, Nice, France, May 29 –June 1, 2023, Proceedings*, pages 299–319, Berlin, Heidelberg, 2023. Springer-Verlag. doi:10.1007/978-3-031-33271-5_20.

12 Rina Dechter and Avi Dechter. Belief maintenance in dynamic constraint networks. In Howard E. Shrobe, Tom M. Mitchell, and Reid G. Smith, editors, *Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, MN, USA, August 21-26, 1988*, pages 37–42. AAAI Press / The MIT Press, 1988. URL: http://www.aaai.org/Library/AAAI/1988/aaai88-007.php.

13 Eugene C. Freuder. In pursuit of the holy grail. *Constraints*, 2(1):57–61, 1997. doi:10.1023/A:1009749006768.

14 Eugene C. Freuder. Conversational modeling for constraint satisfaction. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(20):22592–22597, March 2024. doi:10.1609/aaai.v38i20.30268.

15 Cristina Garbacea and Qiaozhu Mei. Why is constrained neural language generation particularly challenging? *arXiv preprint arXiv:2206.05395*, 2022.

16 Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *CoRR*, abs/2103.13630, 2021. arXiv:2103.13630.

17 Chris Hokamp and Qun Liu. Lexically constrained decoding for sequence generation using grid beam search. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1535–1546, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi:10.18653/v1/P17-1141.

18 Nan Jiang, Maosen Zhang, Willem-Jan Van Hoeve, and Yexiang Xue. Constraint reasoning embedded structured prediction. *J. Mach. Learn. Res.*, 23(1), January 2022.

19 U Junker, F Rossi, P van Beek, and T Walsh. Handbook of constraint programming. *Chapter Configuration*, 2006.

20 Dan Jurafsky and James H. Martin. *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition.* Pearson Prentice Hall, Upper Saddle River, N.J., 2009. URL: http://www.amazon.com/Speech-Language-Processing-2nd-Edition/dp/0131873210/ref=pd_bxgy_b_img_y.

21 Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1), February 2016. doi:10.1609/aaai.v30i1.10080.

22 Elias B. Khalil, Bistra Dilkina, George L. Nemhauser, Shabbir Ahmed, and Yufen Shao. Learning to run heuristics in tree search. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 659–666, 2017. doi:10.24963/ijcai.2017/92.

23 James Kotary, Ferdinando Fioretto, Pascal van Hentenryck, and Bryan Wilder. End-to-end constrained optimization learning: A survey. In *30th International Joint Conference on Artificial Intelligence, IJCAI 2021*, pages 4475–4482. International Joint Conferences on Artificial Intelligence, 2021.

24 Connor Lawless, Jakob Schoeffer, Lindy Le, Kael Rowan, Shilad Sen, Cristina St. Hill, Jina Suh, and Bahareh Sarrafzadeh. "i want it that way": Enabling interactive decision support using large language models and constraint programming, 2024. arXiv:2312.06908.

25 Yixian Liu, Liwen Zhang, Wenjuan Han, Yue Zhang, and Kewei Tu. Constrained text generation with global guidance - case study on commongen. *CoRR*, abs/2103.07170, 2021. arXiv:2103.07170.

26 Michele Lombardi, Michela Milano, and Andrea Bartolini. Empirical decision model learning. *Artificial Intelligence*, 244:343–367, 2017. Combining Constraint Solving with Mining and Learning. `doi:10.1016/j.artint.2016.01.005`.

27 Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, Noah A. Smith, and Yejin Choi. NeuroLogic A*esque decoding: Constrained text generation with lookahead heuristics. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 780–799, Seattle, United States, July 2022. Association for Computational Linguistics. `doi:10.18653/v1/2022.naacl-main.57`.

28 Mohsen Nafar and Michael Römer. Using clustering to strengthen decision diagram bounds for discrete optimization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(8):8082–8089, March 2024. `doi:10.1609/aaai.v38i8.28647`.

29 François Pachet and Pierre Roy. Markov constraints: Steerable generation of markov sequences. *Constraints*, 16(2):148–172, April 2011. `doi:10.1007/s10601-010-9101-4`.

30 Alexandre Papadopoulos, Pierre Roy, Jean-Charles Régin, and François Pachet. Generating all possible palindromes from ngram corpora. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 2489–2495. AAAI Press, 2015.

31 Guillaume Perez and Jean-Charles Régin. MDDs: Sampling and probability constraints. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 226–242, 2017. `doi:10.1007/978-3-319-66158-2_15`.

32 Matt Post and David Vilar. Fast lexically constrained decoding with dynamic beam allocation for neural machine translation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1314–1324, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. `doi:10.18653/v1/N18-1119`.

33 Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2021. `arXiv:2112.10752`.

34 Florian Régin and Elisabetta De Maria. Using on-the-fly model checking to improve constraint programming for dynamic problems. In *2023 IEEE 35th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 393–398, 2023. `doi:10.1109/ICTAI59109.2023.00063`.

35 Jialin Song, ravi lanka, Yisong Yue, and Bistra Dilkina. A general large neighborhood search framework for solving integer linear programs. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 20012–20023. Curran Associates, Inc., 2020. URL: `https://proceedings.neurips.cc/paper_files/paper/2020/file/e769e03a9d329b2e864b4bf4ff54ff39-Paper.pdf`.

36 Damien Sprockeels and Peter Van Roy. Expressing musical ideas with constraint programming using a model of tonal harmony. In *International Joint Conference on Artificial Intelligence*, 2024. To appear.

37 Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. cite arxiv:2302.13971. URL: `http://arxiv.org/abs/2302.13971`.

38 Dimos Tsouros, Hélène Verhaeghe, Serdar Kadıoğlu, and Tias Guns. Holy grail 2.0: From natural language to constraint models, 2023. `arXiv:2308.01589`.

39 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

40 Shunyu Yao, Howard Chen, Austin W. Hanjie, Runzhe Yang, and Karthik R Narasimhan. COLLIE: Systematic construction of constrained text generation tasks. In *The Twelfth International Conference on Learning Representations*, 2024. URL: `https://openreview.net/forum?id=kxgSlyirUZ`.

41 Ruibin Yuan, Hanfeng Lin, Yi Wang, Zeyue Tian, Shangda Wu, Tianhao Shen, Ge Zhang, Yuhang Wu, Cong Liu, Ziya Zhou, Ziyang Ma, Liumeng Xue, Ziyu Wang, Qin Liu, Tianyu Zheng, Yizhi Li, Yinghao Ma, Yiming Liang, Xiaowei Chi, Ruibo Liu, Zili Wang, Pengfei Li, Jingcheng Wu, Chenghua Lin, Qifeng Liu, Tao Jiang, Wenhao Huang, Wenhu Chen, Emmanouil Benetos, Jie Fu, Gus Xia, Roger Dannenberg, Wei Xue, Shiyin Kang, and Yike Guo. Chatmusician: Understanding and generating music intrinsically with llm, 2024. arXiv:2402.16153.

Structure-Guided Local Improvement for Maximum Satisfiability

André Schidler
Algorithms and Complexity Group, TU Wien, Austria

Stefan Szeider
Algorithms and Complexity Group, TU Wien, Austria

Abstract

The enhanced performance of today's MaxSAT solvers has elevated their appeal for many large-scale applications, notably in software analysis and computer-aided design. Our research delves into refining anytime MaxSAT solving by repeatedly identifying and solving with an exact solver smaller subinstances that are chosen based on the graphical structure of the instance. We investigate various strategies to pinpoint these subinstances. This structure-guided selection of subinstances provides an exact solver with a high potential for improving the current solution. Our exhaustive experimental analyses contrast our methodology as instantiated in our tool MaxSLIM with previous studies and benchmark it against leading-edge MaxSAT solvers.

2012 ACM Subject Classification Hardware → Theorem proving and SAT solving; Theory of computation → Discrete optimization; Theory of computation → Automated reasoning; Theory of computation → Constraint and logic programming; General and reference → Experimentation

Keywords and phrases maximum satisfiability, large neighborhood search (LNS), SAT-based local improvement (SLIM), incomplete MaxSAT, graphical structure, metaheuristic

Digital Object Identifier 10.4230/LIPIcs.CP.2024.26

Supplementary Material
Software (Source Code/Results): `https://doi.org/10.5281/zenodo.12516816`

Funding Austrian Science Fund (FWF), project 10.55776/P36420.

Acknowledgements Part of this work was carried out while taking part in the Dagstuhl Seminar 23261 "SAT Encodings and Beyond," as well as in the extended reunion of the program "Satisfiability: Theory, Practice, and Beyond" in the spring of 2023 at the Simons Institute for the Theory of Computing at UC Berkeley.

1 Introduction

MaxSAT solvers (solvers for the partial weighted maximum satisfiability problem) have proven to be indispensable tools with an expansive range of applications, including problems that arise in software analysis [37], post-silicon fault localization [38], the identification of concurrency bugs and suggestions for fixes [13], and malware detection in smartphone apps [10]. Additional applications and case studies [20, 22] highlight MaxSAT's versatility in computer-aided design and related areas. While the focus in the past was on creating superior exact MaxSAT solvers tailored for identifying optimal solutions, there has been a noticeable shift towards the significance of anytime MaxSAT solvers in recent times. Unlike exact solvers that seek optimal results, anytime solvers prioritize finding commendable solutions in a shorter time frame and, when interrupted, output the best solution found so far. Hickey and Bacchus [12] introduced a technique using Large Neighborhood Search [36] that integrates the capabilities of exact and anytime solvers, harnessing the advantages of both.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 26; pp. 26:1–26:23
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In our paper, we further develop the methodology of combining exact and anytime MaxSAT solvers, leveraging the local characteristics of the MaxSAT instance's graphical structure to direct the search and dissection of the problem. We embody our innovative approach in the tool MaxSLIM, which supports an array of strategies for choosing local subinstances grounded on the incidence graph's graphical structure of the MaxSAT instance. Each strategy starts from a small number of variables and strategically extends the set of variables by tracing out a subgraph in the graphical model until a predefined budget is reached. These selection strategies use a preference metric to select the starting variable and to decide which variable to add next. We consider several different metrics in our framework. MaxSLIM uses a new MaxSAT solver specifically designed for solving local instances, or, alternatively, can use any other MaxSAT solver for this purpose.

Our comprehensive experimental evaluation on the instances from the 2023 MaxSAT Evaluation[1] showcases the efficacy of considering the graphical structure in the MaxSAT instances. The insights derived from a wide-ranging set of experiments conclusively indicate that integrating the graphical structure not only elevates the performance but does so in a significant and robust manner. The consistent and notable performance enhancements highlight the value and effectiveness of this integrated approach.

Related Work

As mentioned above, Hickey and Bacchus [12] proposed an anytime MaxSAT solver based on the Large Neighborhood Search (LNS) metaheuristic [36, 25] called MaxSAT-LNS. Similar to MaxSLIM, MaxSAT-LNS also tries to improve a sub-optimal solution using LNS. In each round of LNS, a weighted random subset of variables is selected and their values fixed. Then, a separate solver is run to find an improved assignment for the remaining variables. Fixing a few assignments can imply many other assignments via unit propagation, making the problem of completing the assignment comparatively easy. The main difference to MaxSLIM is that MaxSAT-LNS selects subinstances without utilizing the graphical structure of the instance.

SAT-Based Local Improvement (SLIM) is a specific type of LNS tailored for the use with (Max)SAT solvers. As such, SLIM is an anytime meta-heuristic that embeds (Max)SAT encodings into heuristic algorithms. In the past, SLIM has been used for a variety of problems, such as graph decomposition problems [11, 15, 16, 26], Bayesian Network structure learning [27, 28, 29], decision tree induction [33, 35], graph coloring [32, 34], and circuit minimization [14, 30, 31]. The common aspect between all these SLIM instantiations is that the initial solution (to be improved by iteratively solving local instances) was too large to be computed directly by a SAT-based solver. Hence initial solutions were computed by other, often greedy heuristic methods.

Other SAT-based LNS approaches exist for timetabling [9] and cell placement [8]. Cell placement follows the metaheuristic *Local Search with SAT Oracle (LSSO)* where the user supplies problem specific neighborhood generators and the remaining algorithm is fixed, apart from hyperparameters that control the search. These LNS instantiations, like most other, define the neighborhood based on global properties of the instance. The search is then performed over the whole instance, while restricting how much the solution can change. In contrast, SLIM instantiations look at the structure of the instance and local properties. The search space for the local instances is then unrestricted, apart from ensuring consistency

[1] `https://maxsat-evaluations.github.io/2023/`

with the global solution. One big advantage of the SLIM approach is that it can be applied to instances that are too large to encode as a whole, as it is only necessary to encode the local instances.

Given the large volume of research, we focus the discussion of anytime MaxSAT solving on the state-of-the-art solvers used in our experiments. SATLike [6] is a dynamic local search algorithm that is used in several anytime solvers. SATLike uses a dynamic weighting scheme that is used to pick variables that are then flipped in the hope of finding a good feasible solution. NuWLS [7] is based on the SATLike algorithm and proposes a different weighing scheme. Its implementation NuWLS-c uses another anytime MaxSAT solver (TT-Open-WBO-Inc) for the initial solution. NuWLS-c won the MaxSAT Evaluations 2022 and 2023. NoSAT MaxSAT [5] is another anytime solver based on SATLike and most recently NuWLS, but in contrast to other anytime solvers, relies solely on local search without invoking a SAT solver. TT-Open-WBO-Inc [24] uses either Open-WBO-Inc's [18] Boolean multilevel optimization (BMO) [17] for weighted instances, or a SAT-based bit-vector optimization algorithm (BVO) [21] for unweighted instances. Instead of calling a SAT solver, TT-Open-WBO-Inc's BVO and BMO algorithms call a local search solver that searches for solutions close to the best known solution. The local solver achieves this by polarity saving and incremental SAT calls [23]. Loandra [4] uses preprocessing and linear search (SAT-UNSAT). The linear search is sped up by preprocessing the instance using core-guided search: the core-guided search runs for a limited time, after which the reformulated instance is passed to the linear search. These solvers follow, broadly speaking, two approaches: local search (NuWLS, NoSAT, TT-Open-WBO-Inc) and simplifying the whole instance (Open-WBO-Inc, Loandra) in order to make finding a good feasible solution easier. In contrast, MaxSLIM looks at much larger neighborhoods than local search and does not modify the instance itself. Instead it tries to repeatedly extract easier subproblems.

2 Preliminaries

A *propositional formula in conjunctive normal form* (*CNF formula*) is a set of clauses, each *clause* is a set of literals, each *literal* is a propositional variable or a negated propositional variable. We consider a CNF formula as the conjunction of its clauses and each clause as a disjunction of its literals. For a literal $\ell \in \{x, \neg x\}$ we define $\mathrm{var}(\ell) = x$, for a clause C we define $\mathrm{var}(C) = \{\, \mathrm{var}(\ell) : \ell \in C \,\}$ and for a CNF formula F we define $\mathrm{var}(F) = \bigcup_{C \in F} \mathrm{var}(C)$. An *assignment* is a mapping $\tau : X \to \{0, 1\}$ defined on a set X of variables; we write $\mathrm{var}(\tau) = X$. We extend τ to literals by setting $\tau(\neg x) = 1 - \tau(x)$. We implicitly use the equivalency $\neg\neg v = v$. For an assignment τ, we put $\mathrm{lit}(\tau) = \{\, x : x \in \mathrm{var}(\tau), \tau(x) = 1 \,\} \cup \{\, \neg x : x \in \mathrm{var}(\tau), \tau(x) = 0 \,\}$. An assignment τ is *total* for a CNF formula F if $\mathrm{var}(\tau) = \mathrm{var}(F)$. All future references to assignments address assignments which may or may not be total, unless explicitly specified. For two assignments τ_1, τ_2 with $\mathrm{var}(\tau_1) \cap \mathrm{var}(\tau_2) = \emptyset$ we define $\tau_1 \cup \tau_2$ to be the assignment with $\mathrm{var}(\tau_1 \cup \tau_2) = \mathrm{var}(\tau_1) \cup \mathrm{var}(\tau_2)$ and $(\tau_1 \cup \tau_2)(x) = \tau_i(x)$ for $x \in \mathrm{var}(\tau_i)$.

An assignment τ *satisfies* a clause C if it sets at least one literal of C to 1. An assignment satisfies a CNF formula if it satisfies all its clauses. For a clause C and an assignment τ, we write $C[\tau] = \{\, \ell \in C : \mathrm{var}(\ell) \notin \mathrm{var}(\tau) \,\}$. For a CNF formula F and an assignment τ, $F[\tau]$ denotes the CNF formula obtained from F by removing all clauses that are satisfied by τ and removing from the remaining clauses all literals that τ sets to 0, that is $F[\tau] = \{\, C[\tau] : C \in F$, τ does not satisfy $C \,\}$. Thus τ satisfies F if and only if $F[\tau] = \emptyset$.

For an assignment τ and a CNF formula F, $\mathrm{UP}_F(\tau)$ denotes the assignment obtained from τ by unit propagation over F. This means, we iteratively extend τ to literals ℓ that are forced because there exists a clause $C \in F$ where $C[\tau] = \{\ell\}$.

An instance of the Maximum Satisfiability problem, or *MaxSAT instance* $\mathcal{F}$ is a triple consisting of two CNF formulas F_h and F_s and a weight function $w : C \in F_s \to \mathbb{N}$. The clauses of F_h are *hard*, the clauses of F_s *soft*. A *solution* to $\mathcal{F}$ is an assignment τ that satisfies F_h. The *cost* of a solution τ, denoted $\mathrm{cost}(\mathcal{F}, \tau)$, is the sum of weights of the soft clauses not satisfied by τ. An *optimal solution* is one with minimum cost over all solutions, thereby maximizing the sum of weights of the satisfied soft clauses, therefore the name MaxSAT. In an *unweighted* instance, every soft clause has weight 1.

We write $\mathrm{var}(\mathcal{F}) = \mathrm{var}(F_h) \cup \mathrm{var}(F_s)$ and $\mathcal{F}_\tau = (F_h[\tau], F_s[\tau], w')$ where w' is defined for $C \in F_s[\tau]$ by $w'(C) = \sum_{C' \in F_s \text{ with } C = C'[\tau]} w(C')$.

We distinguish between two types of MaxSAT solvers, *exact* solvers which provide optimal solutions and *anytime* solvers which aim at providing good solutions within a given time bound and thus are not concerned with the optimality of the solutions. Anytime algorithms can be interrupted at any point of time, upon which they immediately output the best solution found so far and terminate. MaxSAT solvers can be both exact and anytime: a solver's classification expresses if a solver's focus is on proving optimality or finding good solutions, as none of the existing solvers is good at both aspects. A subtype of exact solvers are exact *incremental solvers*. These solvers are run multiple times on almost the same MaxSAT instance. In between each run, the MaxSAT instance can be modified. Further, for each run, the user can specify a temporary variable assignment using *assumptions*.

3 MaxSLIM

In this section, we describe *MaxSLIM*, our variant of SLIM for MaxSAT. MaxSLIM expects a MaxSAT instance – the *global instance* – as input. Then, either a global solution τ is provided, or MaxSLIM computes one using a heuristic. MaxSLIM improves this global solution by repeatedly extracting *local instances*, as discussed in the next section, and solving them using a *local solver*. Whenever the local solver finds a solution with lower cost, we found an improvement for the global solution. The local solver is subject to a *local timeout* and is stopped whenever this timeout elapses.

Alongside our structural approach, we also discuss the details of MaxSAT-LNS's neighborhood definition.

3.1 Local Instances

Given a MaxSAT instance $\mathcal{F} = (F_h, F_s, w)$ and weight function w, we iteratively construct a set $L \subsetneq \mathrm{var}(\mathcal{F})$ of *candidate variables* that induce our local instance. The candidate variables are selected using a strategy, the topic of Section 3.3. Given a total assignment τ of F, we define $\tau|_{\mathrm{var}(F)\setminus L}$ as the restriction of τ to $\mathrm{var}(F) \setminus L$, and let $\tau_{\bar{L}} = \mathrm{UP}_{F_h}(\tau|_{\mathrm{var}(\mathcal{F})\setminus L})$. Hence, $\tau|_{\mathrm{var}(F)\setminus L}$ is the assignment after fixing the value of all non-candidate variables and performing unit propagation and L induces the *local instance* $\mathcal{F}_L = \mathcal{F}[\tau_{\bar{L}}]$.

$\mathcal{F}_L$ is expected to be much smaller than $\mathcal{F}$. We call the variables in $\mathrm{var}(\mathcal{F}_L)$ *free variables*, which are a subset of the candidate variables. Given a solution π for $\mathcal{F}_L$, we obtain a new global solution by completing $\tau_{\bar{L}}$ to a total assignment using π: $\tau \leftarrow \tau_{\bar{L}} \cup \pi$.

The updated τ is indeed a solution for $\mathcal{F}$: all hard clauses not part of $\mathcal{F}_L$ are by definition satisfied by $\tau_{\bar{L}}$ and the hard clauses in $\mathcal{F}_L$ are satisfied by π. A similar argument holds for the soft clauses. The global cost decreases exactly by the value the local cost decreases. Therefore, any improvement for the local instance also improves the global solution.

We specify the *budget* in terms of the number of free variables. Hence, we want to choose L such that $|\mathrm{var}(\mathcal{F}_L)| \leq b$ for some budget b. This ensures that the local instances do not take too long to solve. MaxSAT-LNS uses a similar method: it incrementally fixes the values of some variables, until the number of unassigned variables is below a specific threshold. Our focus on the free variables as opposed to the fixed variables, allows us to follow the instance's structure, as is our next topic.

3.2 Local Instance Selection

The goal of local instance selection (i.e., selection of the candidate variables) is that we would like to reach many unsatisfied soft clauses that can be satisfied by changing the assignment of the free variables. This poses two challenges. The first challenge is identifying the right soft clauses: improvements usually require that some satisfied soft clauses become unsatisfied in exchange for satisfying some previously unsatisfied soft clauses of higher total weight. Once the soft clauses are identified, we know which variables need to become free. The second challenge is identifying the other variables required to free the soft clauses' variables. This gives us the candidate variables for our local instance. This task is hard to perform efficiently, as unit propagation behaviour is very instance-specific and hard to predict.

We address these challenges by considering the *incidence graph* (also known as the clause-variable incidence graph) for our local instance selection. While other graphical models for MaxSAT instances exist (such as the primal graph or the resolution graph), our research shows that the incidence graph is best suited for our method: (i) the incidence graphs contains all the information available in the primal graph and resolution graph, and (ii) the neighborhood of a variable shows which clauses are impacted in case we change the variable's value. The incidence graph $G_{\mathcal{F}}$ is the graph with the set of vertices $V(G_{\mathcal{F}}) = \mathrm{var}(\mathcal{F}) \cup F_h \cup F_s$ and the set of edges $E(G_{\mathcal{F}}) = \{\, \{u, C\} : C \in F_h \cup F_s, u \in \mathit{var}(C)\,\}$. Hence, the incidence graph is a bipartite graph that connects the clauses with the variables they contain, negated or unnegated. We annotate the edges with the polarity of the variable in the clause. Given a global solution τ for $\mathcal{F}$ and a set $L \subseteq \mathrm{var}(\mathcal{F})$, we define the restriction of the incidence graph $G_{\mathcal{F},L}$ to unsatisfied clauses. For this definition, we assume that the assignment to the candidate variables changed and use

$$\tau'(x) = \begin{cases} 1 - \tau(x) & \text{if } x \in L, \\ \tau(x) & \text{otherwise.} \end{cases}$$

Then

$$\begin{aligned} V(G_{\mathcal{F},L}) &= \mathrm{var}(\mathcal{F}) \cup \{\, C \in F_s \cup F_h : C \text{ not satisfied by } \tau' \,\} \text{ and} \\ E(G_{\mathcal{F},L}) &= E(G_{\mathcal{F}}) \cap (V(G_{\mathcal{F},L}) \times V(G_{\mathcal{F},L})). \end{aligned}$$

Local instance selection searches for connected subgraphs of the incidence graph that allow for improvements. We focus on connected subgraphs as after changing an assignment to a variable x, unit propagation can only affect variables within the same connected subgraph as x. The restricted incidence graph focuses this search by considering only those clauses that become unsatisfied after changing x.

MaxSLIM constructs local instances using *strategies* for exploring $G_{\mathcal{F},L}$. Each strategy is a different greedy algorithm that picks variables by maximizing a *metric*. Each metric $s(.)$ defines a *score* for each variable or soft clause x, denoted by $s(x)$.

Algorithm 1 shows the general approach. In each iteration, we initially start from a single unsatisfied soft clause $C \in F_s$. The soft clause is chosen according to the metric and we avoid repeatedly choosing the same soft clause by keeping track of our previous choices in D. Hence, we start with initial set $L_0 = \mathrm{var}(C)$ of candidate variables. We then extend this set to $L_{i+1} = L_i \cup S$ – where S depends on the strategy used – until $|\mathrm{var}(\mathcal{F}_{L_{i+1}})|$ exceeds our

Algorithm 1 MaxSLIM.

Input: A MaxSAT instance $\mathcal{F} = (F_h, F_s, w)$, a metric $s(.)$, a strategy σ for selecting local instances, and a budget b.
Output: A solution τ.

```
$\tau \leftarrow$solve($\mathcal{F}$)                          // $\tau$ can also be passed as a parameter.
$D \leftarrow \emptyset$                                   // $D$ keeps track of visited soft clauses.
while within global timeout do
    $i \leftarrow 0$
    $C_s \leftarrow \arg\max_{C \in F_s \setminus D, C \cap \tau = \emptyset} s(C)$
    $L_0 \leftarrow \mathrm{var}(C_s)$
    $D \leftarrow D \cup \{C_s\}$
    while $\mathrm{var}(L_i) \neq \mathrm{var}(\mathcal{F})$ and $|\mathrm{var}(\mathcal{F}_{L_i})| < b$ do
        Extend $L_i$ to $L_{i+1}$ using the strategy $\sigma$.
        $i \leftarrow i + 1$
    end while
    $\tau_{L_{i+1}} \leftarrow \mathrm{solve}(\mathcal{F}_{L_{i+1}})$
    if $\mathrm{cost}(\mathcal{F}_{L_{i+1}}, \tau_{L_{i+1}}) < \mathrm{cost}(\mathcal{F}_{L_{i+1}}, \tau)$ then
        $\tau \leftarrow \tau|_{\mathrm{var}(\mathcal{F}) \setminus \mathrm{var}(\mathcal{F}_{L_{i+1}})} \cup \tau_{L_{i+1}}$
        Update metric.
        $D \leftarrow \emptyset$
    end if
    if $D = \{ C \in F_s : C \cap \mathrm{lit}(\tau) = \emptyset \}$ then
        $D \leftarrow \emptyset$
    end if
end while
return $\tau$
```

budget or we have added all variables. Then $L \coloneqq L_{i+1}$. Whenever we tried all soft clauses or found an improvement, we reset D. Hence, MaxSLIM runs either until the global timeout is reached or the budget allows solving the whole instance.

3.3 Strategies

Given a global instance $\mathcal{F} = (F_h, F_s, w)$, a global solution τ for $\mathcal{F}$, and a metric $s : \mathrm{var}(\mathcal{F}) \cup F_s \rightarrow \mathbb{R}$ (which can depend on τ), we use one of the following strategies for extending L_i to L_{i+1}, where ties are always broken arbitrarily:

- *Variable Strategy*: Let $N_c = \{ C : u \in L_i, \{u, C\} \in E(G_{\mathcal{F}, L_i}) \}$ and $N_v = \{ u : C \in N_c, \{u, C\} \in E(G_{\mathcal{F}, L_i}) \} \setminus L_i$, i.e., N_v is the set of variables which occur in some clause together with at least one variable in L_i. This strategy sets $L_{i+1} = L_i \cup \{\arg\max_{u \in N_v} s(u)\}$. I.e., this strategy adds as many high-scoring variables to the local instance as possible.
- *k-Adjacency Strategy*: This strategy picks a variable $v = \arg\max_{v \in L_i} s(v)$ and then extends L_i to L_{i+1} by adding the k best variables of distance 2 from v in $G_{\mathcal{F}, L_i}$. The idea behind this strategy is that high-scoring candidate variables are only useful if they become free. Adding variables that occur together in a clause with the high-scoring variables increases the chances of the high scoring candidate variables becoming free.
- *Fast Strategy*: This strategy does not use a metric to avoid sorting and priority queues. Let $v \in L_i$ be an arbitrary vertex and N_v be defined as in the Variable Strategy, then $L_{i+1} = L_i \cup N_v$. Hence, all variables occurring together with any variable in L_i are added.

This strategy tries to maximize the speed with which local instances are constructed. Which strategy is best depends on the structure of the instance, as we will discuss in our experiments. We discuss further strategies and results in Appendix A.

Next, we will discuss the different metrics used by the strategies.

3.4 Metrics

The metrics try to identify variables and soft clauses that have a high probability of contributing to an improvement. For brevity, only some metrics are discussed here and more metrics and results can be found in Appendix B. We heavily use the concept of *units*: variable v is a unit of clause C with respect to an assignment τ if $\{v\} = \mathrm{var}(\mathrm{lit}(\tau) \cap C)$. Hence, if a clause has a unit, changing the unit's assignment will make the clause unsatisfied. We define $\mathrm{unit}(v) = \{\, C \in F_s \cup F_h : \{v\} = \mathrm{var}(\mathrm{lit}(\tau) \cap C) \,\}$. Hence, $\mathrm{unit}(v)$ are exactly those clauses that become unsatisfied if we change the value of v. Particularly for instances with homogenous soft clause weights, many soft clauses end up having the same metric score. For this reason, we use $\arg\min_{v \in C} |\mathrm{unit}(v)|$ – smaller is better – as a tie breaker, whenever the score of two soft clauses is the same.

We consider the following metrics:

- *Unit Metric*: For each variable $v \in \mathrm{var}(\mathcal{F})$ the score is

  $$s(v) = -|\mathrm{unit}(v) \cap F_h| - \sum_{C \in \mathrm{unit}(v) \cap F_s} w(C).$$

  This metric prefers variables where changing the assigned value would unsatisfy as few clauses as possible. For a soft clause C_s the score is $\min_{v \in \mathrm{var}(C_s)} s(v)$.
- *Satisfying Metric*:

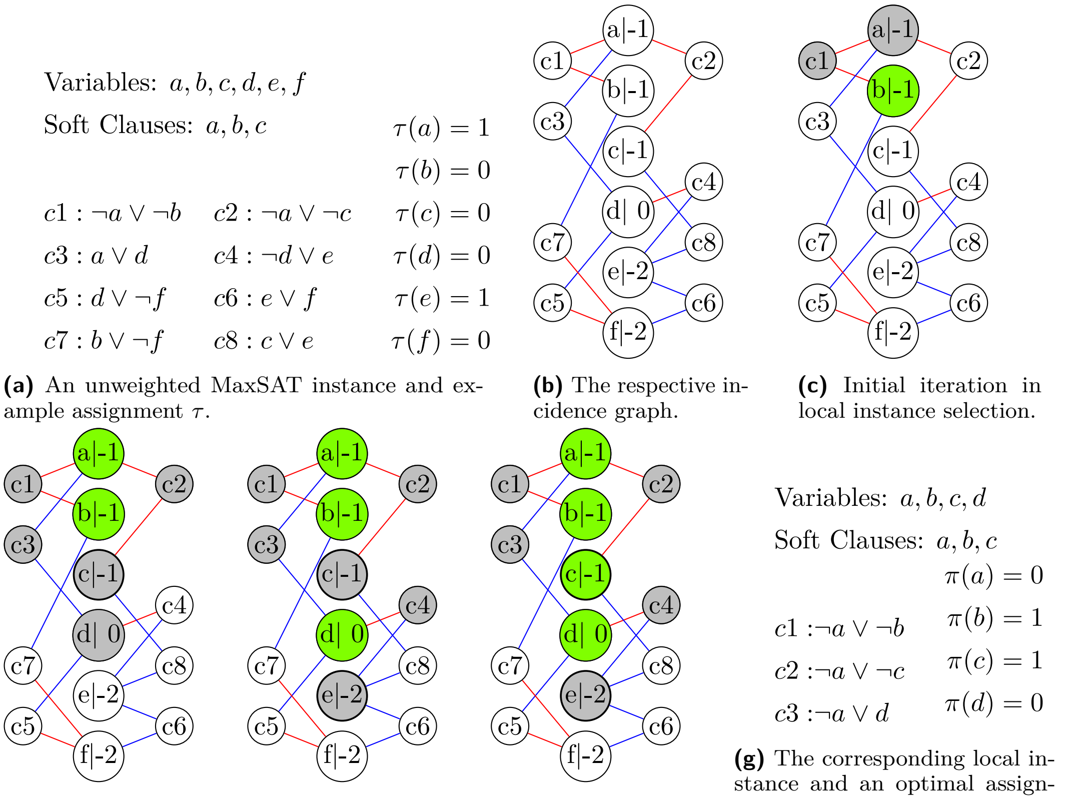
  $$s(v) = \sum_{\substack{C_h \in F_h, C_s \in F_s, \\ v \in \mathrm{var}(F_h), \\ C_h \cap \tau = \{\ell\}, \neg\ell \in C_s}} \begin{cases} 0, & \text{if } \mathrm{lit}(\tau) \cap C_s \neq \emptyset; \\ 0, & \text{if } v = \mathrm{var}(\ell); \\ w(C_s), & \text{otherwise.} \end{cases}$$

  This metric identifies variables in unsatisfied soft clauses that cannot be changed to a different value, as they alone satisfy some hard clause. Giving the other variables in these hard clauses a high score and thereby changing their value can enable MaxSLIM to satisfy more soft clauses. For a soft clause C_s, $s(C_s) = \sum_{v \in \mathrm{var}(C_s)} s(v)$.
- *NuWLS Metric*: This metric uses the initial weighting scheme of NuWLS [7]. For unweighted instances, each soft clause C_s has score $s(C_s) = 1000$, for weighted instances, with w_a being the average weight over all soft clauses, $s(C_s) = w(C_s) \cdot \frac{3000}{w_a}$.
  For the purposes of computing the variable score, let $s(C) = 1$ for all $C \in F_h$. The score for a variable v is defined as

  $$s(v) = \sum_{C_s \in F_s, C_s \cap \mathrm{lit}(\tau) = \emptyset} s(C_s) - \sum_{C \in \mathrm{unit}(v)} s(v).$$

  This metric weighs the clauses that will be satisfied by changing the variable's value against the clauses that will become unsatisfied.

Instead of using the metric scores directly as discussed, we use *weighted random sampling*, similar to MaxSAT-LNS: We use a constant factor c such that $s(x) + c > 0$ for all $x \in F_s \cup \mathrm{var}(\mathcal{F})$ and then, for each $x \in \mathrm{var}(\mathcal{F}) \cup F_s$, we set $s(x) \leftarrow \log u_x \cdot \frac{1}{s(x)+c}$, where u_x is a randomly generated number between 0 and 1. Sampling causes MaxSLIM to explore more diverse local instances, leading to more improvements over time.



Variables: a, b, c, d, e, f

Soft Clauses: a, b, c

		$\tau(a) = 1$
		$\tau(b) = 0$
$c1 : \neg a \vee \neg b$	$c2 : \neg a \vee \neg c$	$\tau(c) = 0$
$c3 : a \vee d$	$c4 : \neg d \vee e$	$\tau(d) = 0$
$c5 : d \vee \neg f$	$c6 : e \vee f$	$\tau(e) = 1$
$c7 : b \vee \neg f$	$c8 : c \vee e$	$\tau(f) = 0$

(a) An unweighted MaxSAT instance and example assignment τ.

(b) The respective incidence graph.

(c) Initial iteration in local instance selection.

(d) Second iteration.

(e) Third iteration.

(f) Last iteration.

Variables: a, b, c, d

Soft Clauses: a, b, c

	$\pi(a) = 0$
$c1 : \neg a \vee \neg b$	$\pi(b) = 1$
$c2 : \neg a \vee \neg c$	$\pi(c) = 1$
$c3 : \neg a \vee d$	$\pi(d) = 0$

(g) The corresponding local instance and an optimal assignment π.

Figure 1 Local instance selection using the Unit Metric, Variable Strategy, and a budget of 4. A red edge indicates that the variable occurs negated in a clause and a blue edge indicates that the variable occurs unnegated.

▶ **Example 1.** Figure 1 shows an example for local instance selection. Figure 1a shows an unweighted MaxSAT instance with three unary soft clauses and a sub-optimal assignment τ. The corresponding incidence graph is shown in Figure 1b. The graph also shows the polarity of the variable using the edge color, blue for positive and red for negative. The example uses the Unit Metric: the weight for the variables is given with the variable name in the graph vertices. a has weight -1 as it alone satisfied $c3$ under τ, while d has weight 0 because it only satisfies $c4$, which is also satisfied by e under τ. Local instance selection starts in Figure 1c using the Variable Strategy, for clarity we do not use weighted random sampling. Initially, b and c could be chosen as they occur in unsatisfied soft clauses and have highest weight. Here, b is chosen arbitrarily as the initial soft clause and variable. The gray vertices have been explored by the strategy and only the variables corresponding to explored vertices are considered as candidate variables in the next iteration. Since a is the only such variable, it is added next in Figure 1d. In the next two iteration, the variables with the highest weights are selected, until we hit our budget of 4. We now have selected four candidate variables that are also free variables and the corresponding local instance is shown in Figure 1g. The improved assignment to the local instance can then be completed to a global assignment using the original assignments to e and f.

3.5 Local Solvers

MaxSLIM can use any MaxSAT solver as the local solver, whether it is an anytime or an exact solver. Next, we will discuss the advantages of different solver types and then discuss our own solver developed for MaxSLIM.

Anytime solvers seem like a straightforward choice, as we only require improved solutions, not necessarily optimal ones. Unfortunately, anytime solvers often struggle to show optimality. Hence, they often run for the full local timeout, where an exact solver can determine within seconds that no improvement is possible. While current implementations of anytime solvers do not offer the option to start from a known solution, the underlying algorithms themselves would support it in principle.

Exact solvers are indeed often faster in showing that no improvement is possible. The main disadvantage when using exact solvers is that they often do not find (good) intermediate solutions, whenever the local run does not finish. There are methods like stratification [2, 17] that find upper bounds for weighted instances during solving, but the upper bounds are often not tight until late in the solving process. Further, exact solvers cannot profit much from an already known solution, apart from hardening [2] which only works for weighted instances. Incremental exact solvers would support encoding the instance only once and reusing learned information across multiple local instances. Unfortunately, our experiments show that the large number of assumptions (up to hundreds of thousands and more) required for expressing the local instances slows down an incremental solver. Further, the large number of assumptions makes it hard for the solver to learn cores or clauses that are useful for another local instance. We could not observe any improvement of using an incremental solver over a non-incremental exact solver.

In MaxSLIM, we use our own exact solver based on the OLL algorithm [1, 19] with some specific adaptations. For brevity, we discuss the main differences to a plain OLL implementation, without giving the details of OLL itself. For this description, we assume that each soft clause consists of only one literal, as this can always be achieved by introducing auxiliary variables.

A simple method that works for all solvers is using the fact that any improved solution has to satisfy at least one additional soft clause. Let $F_{L,s}$ be the set of soft clauses of a local instance F_L and $F_{L,u} = \{\, C \in F_{L,s} : C \cap \mathrm{lit}(\tau) = \emptyset \,\}$. We can now add a single disjunction stating that at least one soft clause in $F_{L,u}$ has to be satisfied. This has at least one of two effects: any solution the local solver finds is different from the current solution, which can lead to improvements in subsequent local runs, and more often, adding this disjunction increases the optimal cost of the local instance. The increase in optimal cost, in turn, makes it easier and faster to determine whether no improvement is possible.

The second change is *upper bound search*, where our solver actively tries to find improved non-optimal solutions. In each OLL iteration, we assume that exactly one additional soft clause in $F_{L,u}$ is satisfied. If the subsequent SAT call returns satisfiable, we have reduced our upper bound and found a better solution. Otherwise, we proceed as usual: we extract a core and increase our lower bound. This way, we may find improved solutions, even if the solver does not find an optimal solution.

4 Experimental Evaluation

In our experimental evaluation, we address several research questions:

Q1: Is there a benefit of a structured approach compared to an unstructured one like MaxSAT-LNS? (Section 4.2)

Q2: Is there a benefit of local improvement compared to just running the initial solver for the entire time? (Section 4.3)

Q3: How does MaxSLIM compare to other anytime solvers? (Section 4.4)

Q4: What strategies/metrics work best? (Section 4.5)

We first introduce our experimental setup and then examine the results concerning these questions.

4.1 Experimental Setup

4.1.1 Cluster

The experiments were run on servers with two AMD EPYC 7402 CPUs, each having 24 cores running at 2.8 GHz, and using Ubuntu 18.04. Each run had 64 GB of memory. We used GCC 11 to compile all the solvers. We use timeouts of 5, 30 and 60 minutes.

4.1.2 Comparison

We compare our implementation of MaxSLIM against MaxSAT-LNS[2], as well as the MaxSAT Evaluation 2023 solvers NuWLS-c (static), TT-Open-WBO-Inc (Glucose for unweighted, IntelSAT for weighted), Loandra, and NoSAT MaxSAT[3]. We utilize the same scoring system as the MaxSAT Evaluation: let c_{best} be the cost of the best known solution for the given instance, and c_{solver} be the cost of the solution the given solver found, the score is calculated with $\frac{c_{\text{best}}+1}{c_{\text{solver}}+1}$. The solvers finding the best solution get a score of 1. The lowest cost among all our experiments provides the baseline. Hence, values in different tables are comparable.

We perform three runs per solver and configuration, using three specific random seeds for reproducibility, whenever the solver supports it. The random seeds themselves have been initially randomly generated. If not stated otherwise, we give the average of the three runs.

We generate an initial solution using NuWLS-c, the winner of the MaxSAT Evaluation 2023, and aim for a comparability between MaxSLIM, MaxSAT-LNS, and NuWLS-c using the following setup. Instead of running NuWLS-c separately for 5, 30, and 60 minutes, we only run it for the 60 minute-runs and extract the current best solution after 1, 5, 30, and 60 minutes. These extracted solutions are then used as the initial solution for MaxSLIM and MaxSAT-LNS: for the 5 minute runs, MaxSLIM and MaxSAT-LNS get NuWLS-c's best solution after one minute as an input; for 30 and 60 minute timeouts we give MaxSLIM and MaxSAT-LNS the NuWLS-c's best solution after 5 minutes as the initial solution. We compensate for this by running MaxSLIM and MaxSAT-LNS for only 4, 25, and 55 minutes instead of 5, 30 and 60 minutes. This setup means that MaxSLIM, MaxSAT-LNS, and NuWLS-c always start from the same solution. Instances where no initial solution could be computed are omitted from the results.

We note that it is common that anytime solvers use other anytime solvers: TT-Open-WBO-Inc uses NuWLS-c, and NuWLS-c uses TT-Open-WBO-Inc [5]. Further, we did not try to create the best anytime solver, but evaluate how well our structured approach works. Interleaving our approach more with the other solvers would yield better results, but makes it hard to identify how much our approach contributes. Possible improvements are using NuWLS for those instances that do not allow local improvements, or optimizing the timeout for the initial solution.

4.1.3 Instances

We used the instances from the 2023 MaxSAT Evaluation's anytime track[4]. The set contains 179 unweighted and 160 weighted instances. Our experiments show that EvalMaxSAT-SCIP, the winner of the unweighted track and close third-best solver in the weighted track, was able to solve 39 of the unweighted instances and 48 of the weighted instances within one hour.

[2] https://github.com/rgh000/MaxSAT_LNS

[3] https://maxsat-evaluations.github.io/2023/descriptions.html

[4] https://maxsat-evaluations.github.io/2023/benchmarks.html

We restrict the results to those instances where we could find an initial solution. This avoids giving NuWLS-c a better score on instances where NuWLS-c does not find an initial solution, but later finds a solution within the timelimit. Hence, we avoid lowering MaxSLIM's and MaxSAT-LNS's score in case of NuWLS-c's poor performance.

4.1.4 Configuration

We solve local instances using our OLL solver limited to a local timeout of 55 seconds. This admits local instance selection and solving the local instance to finish within a minute. As the budget, we initially use $\frac{|\mathrm{var}(\mathcal{F})|}{10}$ many variables, but not more than 25 000. Every five consecutive failures of finding an improvement, we increase the budget by another $\frac{|\mathrm{var}(\mathcal{F})|}{10}$. Whenever the budget reaches the total number of variables, we run the local solver without a timeout on the whole instance. We use the Variable Strategy and either the Unit Metric for unweighted, or the NuWLS Metric for weighted instances.

Table 1 Virtual best average scores for different combinations of solvers. Virtual best scores take for each instance the best solution over the specified solvers.

	Unweighted			Weighted		
Solvers	5 m	30 m	60 m	5 m	30 m	60 m
MaxSLIM	0.886	0.920	9.927	0.833	0.911	0.917
MaxSLIM & MaxLNS	0.896	0.930	0.936	0.849	0.927	0.933
MaxSLIM & NuWLS	0.901	0.937	0.940	0.888	0.928	0.933
All Solvers	0.929	0.970	0.980	0.947	0.976	0.981

4.2 Comparison of SLIM and LNS (Q1)

The comparison between MaxSAT-LNS and MaxSLIM in Tables 2 and 3 shows that MaxSLIM performs overall better than MaxSAT-LNS for all timeouts and scores. This suggests that the structured approach has an advantage over randomly selecting the local instances. In contrast to all the other solvers, MaxSAT-LNS and MaxSLIM both have a large variance between the runs. This suggests that weighted random sampling introduces a significant diversity among the local instances. The virtual best results in Table 1 show that MaxSAT-LNS and MaxSLIM are not very complementary. Interestingly, the difference between the virtual best and MaxSLIM's score is constant over the timeouts, suggesting that some improvements were only found by MaxSAT-LNS.

4.3 Comparison of MaxSLIM and NuWLS-c (Q2)

One crucial question is whether MaxSLIM is better than running the anytime solver used for the initial solution for the entire duration. The results in Table 2 show that MaxSLIM performs better on unweighted instances, where it is better on all metrics for all timeouts.

The results are different for weighted instances as shown in Table 3. Here, MaxSLIM performs worse for the 5-minute runs. The weighted instances contain on average three times more variables and hard clauses, as well as ten times more soft clauses compared to the unweighted instances. This increase in size decreases the performance of the solvers and leads to a poor initial solution. NuWLS-c is then faster at finding improvements than MaxSLIM. This is clearly visible in the much better results on higher timeouts, where MaxSLIM is able

Table 2 Comparison between MaxSLIM, MaxSAT-LNS (MaxLNS), NuWLS-c (NuWLS), TT-Open-WBO-Inc (TT-OpenWI), Loandra, and NoSAT MaxSAT (NoSAT) for unweighted instances. The score is obtained by taking for each instance the best scoring run, the worst scoring run, and the average over all three runs. *Best* shows on how many instances the solver found the best solution.

	MaxLNS	MaxSLIM	NuWLS	TT-OpenWI	Loandra	NoSAT
5-Minutes			164 Instances			
Score Min	0.859	**0.876**	**0.876**	0.872	0.806	0.576
Score Average	0.871	**0.887**	0.885	0.876	0.820	0.589
Score Max	0.887	**0.899**	0.895	0.881	0.835	0.602
Best	32	**42**	33	32	**42**	15
30-Minutes			171 Instances			
Score Min	0.895	0.902	**0.910**	0.896	0.884	0.577
Score Average	0.907	**0.919**	0.917	0.907	0.892	0.591
Score Max	0.928	**0.937**	0.925	0.917	0.900	0.606
Best	36	39	30	28	**41**	13
60-Minutes			171 Instances			
Score Min	0.898	0.907	**0.911**	0.902	0.887	0.587
Score Average	0.912	**0.923**	0.918	0.913	0.900	0.601
Score Max	0.930	**0.940**	0.926	0.925	0.912	0.617
Best	39	**41**	30	29	**41**	14

to find improvements overlooked by NuWLS-c. Even on the 5-minute timeout, MaxSLIM finds many improvements, overlooked by NuWLS-c, as highlighted by the high virtual best score in Table 1.

The gap between the virtual best and the best score behaves similar to the comparison with MaxSAT-LNS. Except for the 5-minute weighted run, the gap remains almost constant over the timeouts, suggesting that some improvements are not found by MaxSLIM. One type of instance where NuWLS-c performs better are instances, where improvements are only possible on (almost) the full instance, since solving these instances with an exact solver is slow.

4.4 Other Solvers (Q3)

Tables 2 and 3 also show the results for the other solvers from the 2023 MaxSAT Evaluation. Striking is the large variance for every solver, showing that luck plays a significant role in the score. MaxSLIM's variance stays comparatively high even for the longer timeouts, indicating that longer timeouts could result in further significant improvements.

In the unweighted case, MaxSLIM is the best solver over the different timeouts, but it clearly distinguishes itself from NuWLS-c only for the best score, as the average score is only better by a small margin.

The weighted case is similar to the comparison with NuWLS-c for the same reasons. Hence, MaxSLIM performs best for the longer timeouts, where the improvements become harder to find.

Table 3 Comparison between MaxSLIM, MaxSAT-LNS (MaxLNS), NuWLS-c (NuWLS), TT-Open-WBO-Inc (TT-OpenWI), Loandra, and NoSAT MaxSAT (NoSAT) for weighted instances. The score is from taking for each instance the best scoring run, the worst scoring run, and the average over all three runs. *Best* shows on how many instances the solver found the best solution.

	MaxLNS	MaxSLIM	NuWLS	TT-OpenWI	Loandra	NoSAT
5-Minutes	154 Instances					
Score Min	0.797	0.813	0.857	**0.862**	0.818	0.307
Score Average	0.818	0.833	0.868	**0.872**	0.843	0.319
Score Max	0.839	0.851	**0.882**	0.881	0.868	0.331
Best	32	29	31	22	**35**	0
30-Minutes	159 Instances					
Score Min	0.877	**0.895**	0.889	0.881	0.882	0.327
Score Average	0.894	**0.911**	0.889	0.888	0.893	0.338
Score Max	0.914	**0.929**	0.912	0.895	0.906	0.348
Best	34	**42**	17	17	33	0
60-Minutes	159 Instances					
Score Min	0.881	**0.903**	0.896	0.889	0.887	0.330
Score Average	0.899	**0.917**	0.906	0.897	0.898	0.341
Score Max	0.918	**0.933**	0.914	0.906	0.910	0.352
Best	35	**41**	19	20	34	0

Hence, MaxSLIM is generally competitive, but needs more local search for better performance on short weighted runs. The virtual best score over all solvers in Table 1 shows that the approaches are quite complementary, as the virtual best for 5-minutes would outperform the single best after one hour.

4.5 Impact of the Configuration (Q4)

An interesting question is how the hyperparameters impact MaxSLIM's performance. We ran the same configuration used in the previous experiments and varied only a single parameter. The virtual best scores over all the configurations discussed in this section are 0.926 for unweighted and 0.867 for weighted instances. Hence, a good dynamic configuration could severely improve MaxSLIM's performance.

Table 4 shows the results of disabling some of MaxSLIM's features.

The upper bounding search does have a noticeable impact. Nonetheless, disabling it yields better results on some instances. The varying performance comes from the fact that the upper bounding search is slower than normal search whenever the local instance cannot be improved.

Weighted random sampling performs in a similar manner, but improves the performance overall more than upper bounding search. There is no clear indication as to when sampling is beneficial and when it is not.

Local search has the biggest impact on the results. Whenever we take the first solution from the initial solver, instead of letting it run the full minute, the performance degrades significantly. This is particularly impactful on weighted instances. MaxSLIM is comparatively slow, but can find improvements not visible to local search. Local search is complementary,

Table 4 Performance of different disabled features of MaxSLIM over a 5-minute timelimit. *Improved* shows how many input solutions were improved. *Better* and *Worse* show on how many instances the configuration did better or worse than the baseline.

	Unweighted				Weighted			
Configuration	Score	Improved	Better	Worse	Score	Improved	Better	Worse
Baseline	0.887	97	-	-	0.833	105	-	-
No Upper Bounding Search	0.878	94	16	32	0.825	101	16	33
No Weighted Sampling	0.875	88	22	40	0.819	103	15	40
No Initial NuWLS Solution	0.817	-	20	94	0.650	-	10	102

as it is very good at finding improvements fast, but often gets stuck when improvements become hard to find. Hence, not running local search severely degrades the performance for short runtimes.

Next, we discuss results on using different strategies and metrics.

4.5.1 Strategies

Table 5 shows the results of comparing different strategies. The results show that the Variable Strategy is overall the best strategy, while the 5-Adjacency finds many solutions missed by the Variable Strategy. The Fast Strategy performs overall worst, but performs better on some very large instances. In general, different strategies perform complementarily.

Table 5 Performance of different strategies. *Best* Score shows on how many instances MaxSLIM found the best solution using the given strategy. *Unique* Best shows on how many instances the best solution could only be found using the given strategy. The timelimit was 5 minutes.

	Unweighted				Weighted			
Strategy	Score	Improved	Best	Unique	Score	Improved	Best	Unique
Variable	**0.887**	**97**	44	5	**0.833**	105	34	14
5-Adjacency	**0.887**	94	**48**	**13**	0.822	**107**	**40**	**18**
Fast	0.873	79	28	5	0.817	99	24	10

In Table 6 are several statistics for each strategy. We can see that the relative time spent on constructing local instances is indeed significantly lower for the fast strategy, and about the same for all other strategies. Interestingly, the number of local instances is much higher for the Variable and 5-Adjacency Strategy. This shows that the fast strategy extracts many local instances that do not lead to an improvement, but require long solving times.

Another interesting observation is that neither the number of local instances, nor the number of improved local instances is a good indicator for performance. According to both results, the Variable Strategy would not be the best strategy.

4.5.2 Metrics

Table 7 shows the results of comparing different metrics. The score between the best and the worst metric generally does not vary much. The Unit Metric is in the absence of weights the best overall metric. Unsurprisingly, for weighted instances, those metrics that use the weights work better.

Table 6 Strategy statistics averaged over all instances: average local instance size relative to the full instance (LI Size), average ratio of free variables to candidate variable per local instance (F/C), average fraction of runtime spent on local instance generation (LI Time Ratio), average fraction of variables changed per improvement (Changes), average ratio of local instances that have been solved optimally (Optimal Ratio), average number of local instances (#LI), average number of local instances that led to an improvement (#Improved).

Strategy	LI Size	F/C	LI Time Ratio	Changes	Optimal Ratio	#LI	#Improved
			Unweighted				
Variable	0.40	0.64	0.10	0.06	0.64	49.11	2.01
5-Adjacency	0.38	0.65	0.12	0.07	0.67	55.49	3.29
Fast	0.41	0.69	0.02	0.07	0.50	32.29	1.69
			Weighted				
Variable	0.23	0.52	0.21	0.12	0.42	33.33	3.27
5-Adjacency	0.28	0.57	0.24	0.10	0.44	29.71	3.55
Fast	0.35	0.60	0.12	0.12	0.35	23.16	1.54

The statistics in Table 8 show that the different metrics perform very similarly. Together with the strategy statistics in Table 6 there are some interesting details regarding the performance differences between weighted and unweighted instances. The local instances for weighted instances are much smaller than for unweighted instances, while constructing them takes significantly longer. The number of variables changed and optimal ratio also show that it is harder to find the improvements for weighted instances. This further explains the poorer performance on weighted instances compared to weighted instances.

Table 7 Performance of different metrics. *Best* Score shows on how many instances MaxSLIM found the best solution using the given metric. *Unique* Best shows on how many instances the best solution could only be found using the given metric. The timelimit was 5 minutes.

	Unweighted				Weighted			
Metric	Score	Improved	Best	Unique	Score	Improved	Best	Unique
Unit	**0.887**	**97**	51	5	0.823	103	**47**	**17**
NuWLS	0.885	**97**	51	8	**0.833**	**105**	44	9
Satisfying	0.883	96	**66**	**13**	0.824	103	36	7

5 Conclusion

In this paper, we have proposed MaxSLIM as a structured approach to anytime MaxSAT solving. It tackles the problem of anytime MaxSAT solving by iteratively extracting and solving smaller subinstances whose selection is guided by the graphical structure of the instance. This combines anytime and exact MaxSAT solvers in a novel way. Our experimental evaluation shows the competitiveness of MaxSLIM as compared to state-of-the-art anytime solvers which have been refined for several years, and other LNS approaches. MaxSLIM's trajectory of improvements over time is particularly attractive for applications with longer runtimes.

Table 8 Metric statistics averaged over all unweighted instances: average local instance size relative to the full instance (LI Size), average ratio of free variables to candidate variable per local instance (F/C), average fraction of runtime spent on local instance generation (LI Time Ratio), average fraction of variables changed per improvement (Changes), average ratio of local instances that have been solved optimally (Optimal Ratio), average number of local instances (#LI), average number of local instances that led to an improvement (#Improved).

Metric	LI Size	F/C	LI Time Ratio	Changes	Optimal Ratio	#LI	#Improved
			Unweighted				
Unit	0.40	0.64	0.10	0.06	0.64	49.11	2.01
NuWLS	0.40	0.63	0.10	0.07	0.63	49.78	2.16
Satisfying	0.42	0.66	0.11	0.06	0.57	47.00	1.78
			Weighted				
Unit	0.27	0.54	0.20	0.10	0.42	31.72	3.59
NuWLS	0.23	0.52	0.21	0.12	0.42	33.33	3.27
Satisying	0.27	0.57	0.22	0.10	0.39	30.87	3.40

Our evaluation uses a default configuration and our results show that choosing the parameters – strategy, metric, timeouts – according to the application can significantly improve MaxSLIM's performance even further.

An interesting avenue for further research is to adapt other anytime solvers to integrate better within MaxSLIM. This would allow us to interleave local improvement phases with additional runs of the initial solver, starting from the best solution found so far. Such an interleaved SLIM approach has shown to be surprisingly powerful for circuit minimization [30, 31]. The large variance when comparing the score of different runs for the same configuration, as well as different configurations, indicates that interleaving MaxSLIM with itself, using different strategies and metrics, may also be beneficial for the result. Although MaxSLIM can be parallelized, results so far show that this is only beneficial on some instances, but further improvements may be possible. As MaxSLIM can benefit from the tuning of its parameters, we expect that further efficiency improvements can be obtained through automated algorithm configuration, possibly even adjusting parameters during the run [3].

References

1 Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in clasp. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *LIPIcs*, pages 211–221. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.ICLP.2012.211.

2 Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving sat-based weighted maxsat solvers. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2012. doi:10.1007/978-3-642-33558-7_9.

3 Carlos Ansótegui, Josep Pon, Meinolf Sellmann, and Kevin Tierney. Reactive dialectic search portfolios for MaxSAT. 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 765–772. AAAI Press, 2017. URL: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14872.

4 Jeremias Berg, Emir Demirovic, and Peter J. Stuckey. Core-boosted linear search for incomplete MaxSAT. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2019. doi:10.1007/978-3-030-19212-9_3.

5 Jeremias Berg, Matti Järvisalo, Ruben Martins, and Andreas Niskanen. Maxsat evaluation 2023: Solver and benchmark descriptions. Technical report, University of Helsinki, 2023.

6 Shaowei Cai and Zhendong Lei. Old techniques in new ways: Clause weighting, unit propagation and hybridization for maximum satisfiability. *Artif. Intell.*, 287:103354, 2020. doi:10.1016/J.ARTINT.2020.103354.

7 Yi Chu, Shaowei Cai, and Chuan Luo. NuWLS: improving local search for (weighted) partial MaxSAT by new weighting techniques. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 3915–3923. AAAI Press, 2023. URL: https://ojs.aaai.org/index.php/AAAI/article/view/25505.

8 Aviad Cohen, Alexander Nadel, and Vadim Ryvchin. Local search with a SAT oracle for combinatorial optimization. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 87–104. Springer, 2021. doi:10.1007/978-3-030-72013-1_5.

9 Emir Demirovic and Nysret Musliu. MaxSAT-based large neighborhood search for high school timetabling. *Comput. Oper. Res.*, 78:172–180, 2017. doi:10.1016/j.cor.2016.08.004.

10 Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/automated-synthesis-semantic-malware-signatures-using-maximum-satisfiability/.

11 Johannes K. Fichte, Neha Lodha, and Stefan Szeider. SAT-based local improvement for finding tree decompositions of small width. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 401–411. Springer Verlag, 2017. doi:10.1007/978-3-319-66263-3_25.

12 Randy Hickey and Fahiem Bacchus. Large neighbourhood search for anytime MaxSAT solving. In Lud De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 1818–1824. International Joint Conferences on Artificial Intelligence Organization, July 2022. doi:10.24963/ijcai.2022/253.

13 Sepideh Khoshnood, Markus Kusano, and Chao Wang. ConcBugAssist: constraint solving for diagnosis and repair of concurrency bugs. In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 165–176. ACM, 2015. doi:10.1145/2771783.2771798.

14 Alexander S. Kulikov, Danila Pechenev, and Nikita Slezkin. SAT-based circuit local improvement. In Stefan Szeider, Robert Ganian, and Alexandra Silva, editors, *47th International Symposium on Mathematical Foundations of Computer Science, MFCS 2022, August 22-26, 2022, Vienna, Austria*, volume 241 of *LIPIcs*, pages 67:1–67:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.MFCS.2022.67.

15 Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. A SAT approach to branchwidth. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 179–195. Springer Verlag, 2016. doi:10.1007/978-3-319-40970-2_12.

16 Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. SAT-encodings for special treewidth and pathwidth. 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 429–445. Springer Verlag, 2017. doi:10.1007/978-3-319-66263-3_27.

17 João Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: algorithms & applications. *Ann. Math. Artif. Intell.*, 62(3-4):317–343, 2011. doi:10.1007/S10472-011-9233-2.

18 Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver,. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, 2014. doi:10.1007/978-3-319-09284-3_33.

19 António Morgado, Carmine Dodaro, and João Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, 2014. doi:10.1007/978-3-319-10428-7_41.

20 António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided maxsat solving: A survey and assessment. *Constraints An Int. J.*, 18(4):478–534, 2013. doi:10.1007/s10601-013-9146-2.

21 Alexander Nadel. Solving MaxSAT with bit-vector optimization. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 54–72. Springer, 2018. doi:10.1007/978-3-319-94144-8_4.

22 Alexander Nadel. Anytime weighted MaxSAT with improved polarity selection and bit-vector optimization. In Clark W. Barrett and Jin Yang, editors, *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, pages 193–202. IEEE, 2019. doi:10.23919/FMCAD.2019.8894273.

23 Alexander Nadel. On optimizing a generic function in SAT. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, pages 205–213. IEEE, 2020. doi:10.34727/2020/ISBN.978-3-85448-042-6_28.

24 Alexander Nadel. Polarity and variable selection heuristics for sat-based anytime maxsat. *J. Satisf. Boolean Model. Comput.*, 12(1):17–22, 2020. doi:10.3233/sat-200126.

25 David Pisinger and Stefan Ropke. Large neighborhood search. In *Handbook of Metaheuristics*, pages 399–419. Springer Verlag, 2010.

26 Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. MaxSAT-based postprocessing for treedepth. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 478–495. Springer, 2020. doi:10.1007/978-3-030-58475-7_28.

27 Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. Learning fast-inference bayesian networks. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 17852–17863, 2021. URL: https://proceedings.neurips.cc/paper/2021/hash/94e70705efae423efda1088614128d0b-Abstract.html.

28 Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. Turbocharging treewidth-bounded bayesian network structure learning. 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 3895–3903. AAAI Press, 2021. doi:10.1609/aaai.v35i5.16508.

29 Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. Learning large Bayesian networks with expert constraints. In James Cussens and Kun Zhang, editors, *Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence*, volume 180 of *Proceedings of Machine Learning Research*, pages 1592–1601. PMLR, 01–05 August 2022. URL: https://proceedings.mlr.press/v180/peruvemba-ramaswamy22a.html.

30 Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. Circuit minimization with QBF-based exact synthesis. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 4087–4094. AAAI Press, 2023. URL: https://ojs.aaai.org/index.php/AAAI/article/view/25524.

31 Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. eSLIM: Circuit minimization with SAT based local improvement. In *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*, 2024. to appear.

32 André Schidler. SAT-based local search for plane subgraph partitions (CG challenge). In Xavier Goaoc and Michael Kerber, editors, *38th International Symposium on Computational Geometry, SoCG 2022, June 7-10, 2022, Berlin, Germany*, volume 224 of *LIPIcs*, pages 74:1–74:8. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SoCG.2022.74.

33 André Schidler and Stefan Szeider. SAT-based decision tree learning for large data sets. In *Proceedings of AAAI'21, the Thirty-Fifth AAAI Conference on Artificial Intelligence*. AAAI Press, 2021. doi:10.1609/aaai.v35i5.16509.

34 André Schidler and Stefan Szeider. SAT-boosted tabu search for coloring massive graphs. *ACM J. Exp. Algorithmics*, 28, July 2023. doi:10.1145/3603112.

35 André Schidler and Stefan Szeider. SAT-based decision tree learning for large data sets. *J. Artif. Intell. Res.*, 80:875–918, 2024.

36 Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael J. Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 1998. doi:10.1007/3-540-49481-2_30.

37 Xujie Si, Xin Zhang, Radu Grigore, and Mayur Naik. Maximum satisfiability in software analysis: Applications and techniques. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 68–94. Springer, 2017. doi:10.1007/978-3-319-63387-9_4.

38 Charlie Shucheng Zhu, Georg Weissenbacher, Divjyot Sethi, and Sharad Malik. Sat-based techniques for determining backbones for post-silicon fault localisation. In Zeljko Zilic and Sandeep K. Shukla, editors, *2011 IEEE International High Level Design Validation and Test Workshop, HLDVT 2011, Napa Valley, CA, USA, November 9-11, 2011*, pages 84–91. IEEE Computer Society, 2011. doi:10.1109/HLDVT.2011.6113981.

A Strategies

We use the same definitions as in Section 3.3: given a global instance $\mathcal{F} = (F_h, F_s, w)$, a global solution τ for $\mathcal{F}$, and a metric $s : \mathrm{var}(\mathcal{F}) \cup F_s \to \mathbb{R}$ (which can depend on τ), we use the following additional strategies for extending L_i to L_{i+1}, where ties are always broken arbitrarily:

- *Adjacency Strategy*: This strategy picks a variable $v = \arg\max_{v \in L_i} s(v)$ and then extends L_i to L_{i+1} by adding all variables of distance 2 from v in $G_{\mathcal{F},L_i}$. The idea behind this strategy is that high scoring candidate variables are only useful if they become free. Adding all variables that occur together in a clause with the high scoring variables, increases the chances of the high scoring candidate variables becoming free.
- *Clause Strategy*: This strategy picks an arbitrary clause $C \in V(G_{\mathcal{F},L_i})$ with $\mathrm{var}(C) \cap L_i \neq \emptyset$ and then sets $L_{i+1} = L_i \cup \{\arg\max_{v \in \mathrm{var}(C) \setminus L_i} s(v)\}$. The idea here is that should we change the assignment to any candidate variable, some clauses may become unsatisfied. This strategy follows the chain of necessary assignment changes to ensure that all clauses are satisfied.
- *Global Strategy*: This strategy emulates MaxSAT-LNS's neighborhood definition and does not follow Algorithm 1: instead of starting from a soft clause, we use $L_0 = \mathrm{var}(\mathcal{F})$ and we stop once we are within our budget. In each iteration $L_{i+1} = L_i \setminus \{\arg\min_{v \in L_i} s(v)\}$. This strategy ensures that the assignment of high scoring variables is preserved.

Different strategies explore the incidence graph differently and lead to different local instances. Further, different strategies have different runtime complexities. The two factors that influence runtime the most are sorting the variables to pick the variable of highest score and how often we need to run unit propagation to calculate the number of free variables. Hence, the Fast Strategy is indeed the fastest, followed by the Global Strategy, as the latter requires sorting all variables only once. We will further discuss the runtimes in the experimental results in the next section.

A.1 Strategy Results

Table 9 Performance of different strategies. *Best* Score shows on how many instances MaxSLIM found the best solution using the given strategy. *Unique* Best shows on how many instances the best solution could only be found using the given strategy. The timelimit was 5 minutes.

	Unweighted				Weighted			
Strategy	Score	Improved	Best	Unique	Score	Improved	Best	Unique
Global	**0.906**	92	**64**	**22**	0.826	104	35	14
Variable	0.887	**97**	44	5	**0.833**	105	34	14
5-Adjacency	0.887	94	48	13	0.822	**107**	**40**	**18**
Clause	0.877	95	36	1	0.816	106	28	8
Adjacency	0.873	92	36	4	0.820	106	29	13
Fast	0.873	79	28	5	0.817	99	24	10

Table 9 shows the results of comparing different strategies. The results show that the Variable Strategy is overall the best strategy, while the Global Strategy performs better for unweighted instances. Nonetheless, every strategy has several instances where it is the only

Table 10 Strategy statistics averaged over all instances: average local instance size relative to the full instance (LI Size), average ratio of free variables to candidate variable per local instance (F/C), average fraction of runtime spent on local instance generation (LI Time Ratio), average fraction of variables changed per improvement (Changes), average ratio of local instances that have been solved optimally (Optimal Ratio), average number of local instances (#LI), average number of local instances that led to an improvement (#Improved).

Strategy	LI Size	F/C	LI Time Ratio	Changes	Optimal Ratio	#LI	#Improved
			Unweighted				
Global	0.40	-	0.02	0.07	0.73	77.79	4.25
Variable	0.40	0.64	0.10	0.06	0.64	49.11	2.01
5-Adjacency	0.38	0.65	0.12	0.07	0.67	55.49	3.29
Clause	0.33	0.61	0.11	0.07	0.65	55.35	2.13
Adjacency	0.36	0.66	0.10	0.07	0.69	50.11	2.65
Fast	0.41	0.69	0.02	0.07	0.50	32.29	1.69
			Weighted				
Global	0.24	-	0.09	0.11	0.50	46.59	4.30
Variable	0.23	0.52	0.21	0.12	0.42	33.33	3.27
5-Adjacency	0.28	0.57	0.24	0.10	0.44	29.71	3.55
Clause	0.24	0.54	0.23	0.06	0.48	34.06	3.33
Adjacency	0.20	0.57	0.24	0.09	0.45	28.09	3.28
Fast	0.35	0.60	0.12	0.12	0.35	23.16	1.54

one that could find the best solution. The difference in score is significantly larger among unweighted instances compared to weighted instances. Hence, it is easier to give a good default strategy for unweighted instances than for weighted instances.

In Table 10 are several statistics for each strategy. We can see that the relative time spent on constructing local instances is low for the Global Strategy and the Fast Strategy, and about the same for all other strategies. Nonetheless, the number of local instances is much higher for the global strategy than for the fast strategy. This shows that the fast strategy extracts many local instances that do not lead to an improvement, but require long solving times. This also explains the good performance of the global strategy on unweighted instances: the high number of local instances shows that MaxSLIM increases the budget faster than with the other strategies. Further, the global strategy is good at finding many small improvements, which has more overall impact on unweighted instances than on weighted instances.

Generally, the statistics are more homogenous for unweighted instances than for weighted instances, which also explains the results in Table 9. Another interesting observation is that neither the number of local instances, nor the number of improved local instances is a good indicator for performance. According to both results, the Variable Strategy would be among the worst strategies.

B Metrics

We use the definitions from Section 3.4: a variable v is a unit of clause C with respect to an assignment τ if $\{v\} = \mathrm{var}(\mathrm{lit}(\tau) \cap C)$ and $\mathrm{unit}(v) = \{\, C \in F_s \cup F_h : \{v\} = \mathrm{var}(\mathrm{lit}(\tau) \cap C) \,\}$.

We define the following additional metrics:

- *Random Metric*: Assigns each variable and soft clause a random score. This causes widespread exploration over consecutive local instances, but does not consider which parts of the instance are more promising.
- *Counting Metric*: The score of a variable is the negative number of times it was selected as a candidate variable. For a clause C, the score $s(C) = \sum_{v \in \mathrm{var}(C)} s(v)$. This metric encourages exploration of all variables over time.
- *LNS Metric*: The metric used by MaxSAT-LNS [12]. For a variable v, we define

$$s(v) = \sum_{\substack{C \in F_s,\\ v \in \mathrm{var}(C)}} \begin{cases} -w(C), & \text{if } v \in \mathrm{var}(C \cap \mathrm{lit}(\tau)); \\ w(C), & \text{if } C \cap \mathrm{lit}(\tau) = \emptyset; \\ 0, & \text{otherwise.} \end{cases}$$

  We extend this scoring to a clause C by setting $s(C) = \sum_{v \in \mathrm{var}(C)} s(v)$.
- *Ratio Metric*: The score for a variable v is the number of clauses that would be satisfied if v's assigment were changed divided by the number of clauses that would become unsatisfied by the change. For a clause C the score is $\min_{v \in \mathrm{var}(C)} s(v)$.

B.1 Metric Results

Table 11 Performance of different metrics. *Best* Score shows on how many instances MaxSLIM found the best solution using the given metric. *Unique* Best shows on how many instances the best solution could only be found using the given metric. The time limit was 5 minutes.

	Unweighted				Weighted			
Metric	Score	Improved	Best	Unique	Score	Improved	Best	Unique
Unit	**0.887**	**97**	51	5	0.823	103	**47**	**17**
NuWLS	0.885	**97**	51	8	**0.833**	**105**	44	9
Satisfying	0.883	96	**66**	**13**	0.824	103	36	7
LNS	0.882	96	56	12	0.823	103	41	14
Counting	0.881	92	45	2	0.822	104	38	7
Random	0.881	91	39	0	0.823	102	32	5
Ratio	0.879	92	46	3	0.822	103	32	6

Table 11 shows the results of comparing different metrics. The score between the best and the worst metric generally does not vary much. Nonetheless, apart from the Random Metric, every metric achieves several unique best scores. The Unit Metric is in the absence of weights the best overall metric. Unsurprisingly, for weighted instances, those metrics that use the weights work better.

Table 12 shows various statistics for these metrics. In general, most statistics do not vary much between metrics, except for the number of improving local instances.

Table 12 Metric statistics averaged over all unweighted instances: average local instance size relative to the full instance (LI Size), average ratio of free variables to candidate variable per local instance (F/C), average fraction of runtime spent on local instance generation (LI Time Ratio), average fraction of variables changed per improvement (Changes), average ratio of local instances that have been solved optimally (Optimal Ratio), average number of local instances (#LI), average number of local instances that led to an improvement (#Improved).

Metric	LI Size	F/C	LI Time Ratio	Changes	Optimal Ratio	#LI	#Improved
			Unweighted				
Unit	0.40	0.64	0.10	0.06	0.64	49.11	2.01
NuWLS	0.40	0.63	0.10	0.07	0.63	49.78	2.16
Satisfying	0.42	0.66	0.11	0.06	0.57	47.00	1.78
LNS	0.43	0.64	0.09	0.06	0.56	46.32	1.79
Counting	0.40	0.63	0.10	0.06	0.64	46.87	1.72
Random	0.40	0.63	0.10	0.06	0.63	46.79	1.76
Ratio	0.40	0.64	0.11	0.06	0.64	49.99	2.07
			Weighted				
Unit	0.27	0.54	0.20	0.10	0.42	31.72	3.59
NuWLS	0.23	0.52	0.21	0.12	0.42	33.33	3.27
Satisying	0.27	0.57	0.22	0.10	0.39	30.87	3.40
LNS	0.28	0.54	0.20	0.09	0.39	31.23	3.46
Counting	0.25	0.53	0.20	0.10	0.40	31.16	3.23
Random	0.25	0.53	0.20	0.08	0.38	30.40	3.15
Ratio	0.25	0.54	0.21	0.08	0.39	30.58	3.52

Efficient Implementation of the Global Cardinality Constraint with Costs

Margaux Schmied
Université Côte d'Azur, CNRS, I3S, Sophia Antipolis, France

Jean-Charles Régin
Université Côte d'Azur, CNRS, I3S, Sophia Antipolis, France

Abstract

The success of Constraint Programming relies partly on the global constraints and implementation of the associated filtering algorithms. Recently, new ideas emerged to improve these implementations in practice, especially regarding the all different constraint.

In this paper, we consider the cardinality constraint with costs. The cardinality constraint is a generalization of the all different constraint that specifies the number of times each value must be taken by a given set of variables in a solution. The version with costs introduces an assignment cost and bounds the total sum of assignment costs. The arc consistency filtering algorithm of this constraint is difficult to use in practice, as it systematically searches for many shortest paths. We propose a new approach that works with upper bounds on shortest paths based on landmarks. This approach can be seen as a preprocessing. It is fast and avoids, in practice, a large number of explicit computations of shortest paths.

2012 ACM Subject Classification Computing methodologies → Planning for deterministic actions; Theory of computation → Constraint and logic programming

Keywords and phrases global constraint, filtering algorithm, cardinality constraints with costs, arc consistency

Digital Object Identifier 10.4230/LIPIcs.CP.2024.27

Funding This work has been supported by the French government, through the 3IA Côte d'Azur Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-19-P3IA-0002.

1 Introduction

In Constraint Programming (CP), a problem is defined on variables and constraints. Each variable is provided with a domain defining the set of its possible values. A constraint expresses a property that must be satisfied by a set of variables. CP uses a specific resolution method for each constraint.

The success of CP relies on the use of high-performance filtering algorithms (also known as propagators). These algorithms remove values from variable domains that are not consistent with the constraint, i.e. that do not belong to a solution of the constraint's underlying sub-problem. The most well-known propagator is that of the all different (alldiff) constraint, which specifies that a set of variables must all take different values. The efficiency in practice of that propagator strongly depends on its implementation. Thus, algorithms proposing practical improvements on Régin's algorithm [15] are still appearing [22, 21].

In this article, we consider another constraint introduced by Régin that is also popular [3, 12, 18, 7, 4]: the cardinality constraint with costs [14]. We propose to try to speed up its filtering algorithm when there is nothing to deduce. This is often the case at the start of the search, particularly as the optimal value is far from known. In addition, at this stage, the gains can be significant since few values have been removed from the domains, and so the complexity of the algorithms is greater. This approach can be particularly interesting with

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 27; pp. 27:1–27:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

aggressive restarting methods and could simplify the use of CP: there is less need to worry about the inference strength of constraints versus their cost. We can worry less about the type of filtering to be used and consider the arc consistency right away.

The global cardinality constraint (gcc) [16] is a generalization of the alldiff constraint. A gcc is specified in terms of a set of variables $X = \{x_1, ..., x_p\}$ which take their values in a subset of $V = \{v_1, ..., v_d\}$. It constrains the number of times a value $v_i \in V$ is assigned to variables in X to belong to an interval $[l_i, u_i]$.

A gcc with costs (costgcc) is a generalization of a gcc in which a cost is associated with each value of each variable. Then, each solution of the underlying gcc is associated with a global cost equal to the sum of the costs associated with the assigned values of the solution. In a costgcc constraint the global cost must be less than a given value, H.

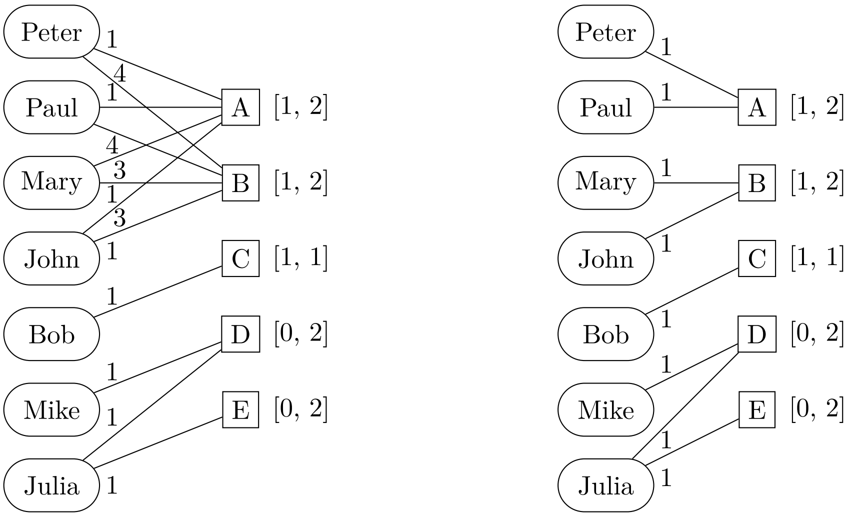


Figure 1 Example of a global cardinality constraint with costs. Source [14]. The sum of assignment costs must be less than or equal to 11. On the left, the original problem and on the right, the same problem after deleting all arcs that cannot belong to a solution.

Cardinality constraints with costs has proved useful in many real-life applications, such as routing, scheduling, rostering, or resource allocation problems. The total costs are often used for expressing preferences, time or cost.

Figure 1 gives an example of a costgcc constraint and the associated filtering algorithm. There are 7 workers represented by the variables $Peter, Paul, Mary, John, Bob, Mike, Julia$ and 5 tasks represented by the values A, B, C, D, E. Each worker has the ability to perform certain tasks and must perform exactly one of them. There is an arc from a worker to a task if the worker can perform the task, its cost corresponding to the time it takes the worker to perform the task. A task has a capacity defining the number of times it must be performed. For example, A must be performed between 1 and 2 times. The objective is to find an assignment whose sum of costs is less than 11. The best possible assignment has a cost of 7, so it is a solution. On the right-hand side of Figure 1, all arcs that cannot belong to a solution have been removed. For example, the arc $(Peter, B)$ can be deleted. If B is assigned to $Peter$, then the maximum capacity of B will be exceeded, so the arc $(Mary, B)$ or $(John, B)$ cannot be part of the solution. If $(John, B)$ is kept, then a value must be assigned to $Mary$, the only possibility is $(Mary, A)$ with a cost of 3. The cost of all assignments is now 12, which is more than 11, so this is not a solution. Similarly, if $(Mary, B)$ is kept, then the only possibility for $John$ is $(John, A)$ with a cost of 3 and the total cost is 12, which is too high.

The filtering algorithm associated with a costgcc constraint [14] can be described as repetitive. First, it computes a maximum flow at minimum cost to determine whether the constraint is consistent (i.e., admits a solution). Then, to find out whether a variable x can be instantiated with a value a, it tries to pass a unit of flow through the arc representing the assignment of a to x so that the total cost of the flow is less than H. This operation involves computing min-cost flow through an arc from a given min-cost flow. This can be done by searching for a shortest path between x and a in the residual graph of the min-cost flow. Furthermore, it has been shown that it is possible to avoid computing a shortest path for each value of each variable and that computing one shortest path per assigned value (which is less than the number of variables) is sufficient [14]. Unfortunately, the algorithm is repeated for each assigned value, which often proves prohibitively expensive.

In this paper, we introduce a new approach avoiding this repetitive aspect as much as possible. Our approach is based on several observations:

- Finding a min-cost flow for each assignment is not necessary. Finding that there exists a flow whose cost is less than H is enough.
- It is not necessary to compute any path exactly because we are only interested in their costs, not the path. Further, the exact value of the cost is not necessary either. An upper bound below a maximum cost is sufficient.
- The use of landmarks (i.e., particular nodes) have proved their worth in speeding up computations of the shortest paths between large elements (millions of nodes) [6]. Let x and y be two nodes of a graph and p be another node called landmark, then we have: $d(x,p) + d(p,y) \geq d(x,y)$ where $d(i,j)$ is the shortest path distance from i to j. Thus, by selecting one or several good landmarks p we can find a good upper bound of $d(x,y)$ for each pair of nodes x, y.
- Calls to the filtering algorithm often do not remove any value. This means that the margin (i.e., slack between H and the min-cost flow value) is often large relative to the data, so using the upper bound should give good results.

On the basis of the above, we propose to introduce preprocessing in order to reduce the effective shortest path computations as proposed by Régin's algorithm. Our approach is to search for landmarks and use them to compute upper bounds on paths to avoid unnecessary explicit shortest path computations. We consider several types of landmarks to integrate the structure of the graph, such as landmarks at the periphery (outline) of the graph or at the center. The advantage of this approach is its low cost because only two shortest paths are required per landmark. We also introduce a way to quickly detect whether a costgcc constraint is arc consistent.

The paper is organized as follows. Section 2 recalls some preliminaries on constraint programming, graph and flow theory. Section 3 describes Régin's algorithm because our method is based on it. Section 4 introduces upper bounds on shortest paths based on landmarks and, in Section 5, the arc consistency algorithm is accordingly adapted. Section 6 details some landmark selection methods. Section 7 gives some experiments on classical problems showing that our approach dramatically reduces the number of computed shortest paths.

2 Preliminaries

The following definitions, theorems and algorithms are based on the following papers and books: [14, 2, 10, 17, 1].

Constraint Programming

A finite constraint network $\mathcal{N}$ is defined as a set of $n \in N$ variables $X = \{x_1, \ldots, x_n\}$, a set of current domains $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible values for variable x_i, and a set $\mathcal{C}$ of constraints between variables. We introduce the particular notation $\mathcal{D}_0 = \{D_0(x_1), \ldots, D_0(x_n)\}$ to represent the set of initial domains of $\mathcal{N}$ on which constraint definitions were stated. A constraint C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ that specifies the allowed combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An element of $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ is called a tuple on $X(C)$ and is denoted by τ. In a tuple τ, the assignment of the i^{th} variable is denoted by τ_i. $var(C, i)$ represents the i^{th} variable of $X(C)$. A value a for a variable x is often denoted by (x, a). Let C be a constraint. A tuple τ on $X(C)$ is valid if $\forall (x, a) \in \tau, a \in D(x)$. C is consistent iff there exists a tuple τ of $T(C)$ which is valid. A value $a \in D(x)$ is consistent with C iff $x \notin X(C)$ or there exists a valid tuple τ of $T(C)$ with $(x, a) \in \tau$.

The costgcc constraint is formally defined as follows.

▶ **Definition 1** ([14]). *A **global cardinality constraint with costs** is a constraint C associated with a cost function on $X(C)$ `cost`, an integer H and in which each value $a_i \in D(X(C))$ is associated with two positive integers l_i and u_i*

$$\begin{aligned} T(C) = \{\ & \tau \textit{ such that } \tau \textit{ is a tuple on } X(C) \\ & \textit{and } \forall a_i \in D(X(C)) : l_i \leq \#(a_i, \tau) \leq u_i \\ & \textit{and } \Sigma_{i=1}^{|X(C)|} cost(\mathrm{var}(C, i), \tau[i]) \leq H\ \} \end{aligned}$$

It is denoted by $costgcc(X, l, u, cost, H)$.

To understand how arc consistency on a costgcc is established, some concepts from graph theory and flow theory are required.

Graph theory

A directed graph or digraph $G = (X, U)$ consists of a node set X and an arc set U, where every arc (x, y) is an ordered pair of distinct nodes. We will denote by $X(G)$ the node set of G and by $U(G)$ the arc set of G. The cost of an arc is a value associated with the arc. When costs are associated with arcs, one should talk about weighted directed graphs.

A path from node x_1 to node x_k in G is a list of nodes $[x_1, ..., x_k]$ such that (x_i, x_{i+1}) is an arc for $i \in [1..k-1]$. The path is called simple if all its nodes are distinct. The cost of a path P, denoted by $cost(P)$, is the sum of the costs of the arcs contained in P. A shortest path from a node s to a node t is a path from s to t whose cost is minimum.

Flow theory

Let G be a digraph where each arc (x, y) is associated with three information: l_{xy} the lower bound capacity, u_{xy} the upper bound capacity and c_{xy} the cost of the arc.

A flow in G is a function f satisfying the following two conditions:

- For any arc (x, y), f_{xy} represents the amount of some commodity that can "flow" through the arc. Such a flow is permitted only in the indicated direction of the arc, i.e., from x to y. For convenience, we assume $f_{xy} = 0$ if $(x, y) \notin U(G)$.
- A conservation law is fulfilled at each node: $\forall y \in X(G) : \sum_x f_{xy} = \sum_z f_{yz}$.

The cost of a flow f is $cost(f) = \sum_{(x,y) \in U(G)} f_{xy} c_{xy}$.

The feasible flow problem consists in computing a flow in G that satisfies the capacity constraint. That is finding f such that $\forall(x,y) \in U(G)$ $l_{xy} \leq f_{xy} \leq u_{xy}$. The minimum cost flow problem consists in finding a feasible flow f such that $cost(f)$ is minimum.

A min cost flow can be computed thanks to the augmenting shortest path algorithm. The main idea of the basic algorithms of flow theory is to proceed by successive improvement of flows that are computed in a graph in which all the lower bounds are zero and the current flow is the zero flow (i.e., the flow value is zero on all arcs).

First, assume that there is no lower capacity. So, consider that all the lower bounds are equal to zero and suppose that you want to increase the flow value for an arc (x,y). In this case, the zero flow is a feasible flow. Let P be a shortest path from y to x different from (y,x), and $val = min(\{u_{xy}\} \cup \{u_{ab}$ s.t. $(a,b) \in P\})$. Then we can define the function f on the arcs of G such that $f_{ab} = val$ if $(a,b) \in P$ or $(a,b) = (x,y)$, and $f_{ab} = 0$ otherwise. This function is a flow in G and $f_{xy} > 0$. Now, from this flow we can define a particular graph without any flow value and all lower bounds equal to zero, the residual graph.

▶ **Definition 2.** *The **residual graph** for a given flow f, denoted by $R(f)$, is the digraph with the same node set as in G and with the arc set defined as follows:*
$\forall(x,y) \in U(G)$:

- *$f_{xy} < u_{xy} \Leftrightarrow (x,y) \in U(R(f))$ and has cost $rc_{xy} = c_{xy}$ and upper bound capacity $r_{xy} = u_{xy} - f_{xy}$.*
- *$f_{xy} > l_{xy} \Leftrightarrow (y,x) \in U(R(f))$ and has cost $rc_{yx} = -c_{xy}$ and upper bound capacity $r_{yx} = f_{xy} - l_{xy}$.*

All the lower bound capacities are equal to 0.

Then, we can select an arc and apply the previous algorithm on this arc in order to increase its flow value. By dealing only with shortest path we can guarantee that the computed flow will have a minimum cost.

Now consider the lower capacities. In this case, we can use the algorithm mentioned by Régin:

Start with the zero flow f^o. This flow satisfies the upper bounds. Set $f = f^o$, and apply the following process while the flow is not feasible:

1) pick an arc (x,y) such that f_{xy} violates the lower bound capacity in G (i.e., $f_{xy} < l_{xy}$).
2) Find P a shortest path from y to x in $R(f) - \{(y,x)\}$.
3) Obtain f' from f by sending flow along P; set $f = f'$ and goto 1)

If, at some point, there is no path for the current flow, then a feasible flow does not exist. Otherwise, the obtained flow is feasible and is a minimum cost flow.

3 Filtering Algorithm

Our work builds on top of the original costgcc filtering (i.e., [14]). Before presenting how we speed up the algorithm for costgcc, let us briefly review the original algorithm.

There is a relation between a costgcc and the search for min-cost flow in a particular graph.

▶ **Definition 3** ([14]). *Given $C = costgcc(X, l, u, cost, H)$. The value graph of C is the bipartite graph $GV(C) = (X(C), D(X(C)), U)$ where $(x,a) \in U$ if $a \in D_x$. The **value network** of C is the directed graph $N(C)$ with l_{xy} the lower bound capacity, u_{xy} the upper bound capacity and c_{xy} the cost on arc from the node x to the node y. $N(C)$ is obtained from the value graph $GV(C)$ by:*

- *Orienting each edge of $GV(C)$ from values to variables.* $\forall x \in X(C) : \forall a \in D(x) : l_{ax} = 0$, $u_{ax} = 1$ *and* $c_{ax} = cost(x,a)$.
- *Adding a node s and an arc from s to each value.* $\forall a \in D(X(C))$: $l_{sa} = l_a$, $u_{sa} = u_a$ *and* $c_{sa} = 0$.
- *Adding a node t and an arc from each variable to t.* $\forall x \in X(C) : l_{xt} = 1$, $u_{xt} = 1$ *and* $c_{xt} = 0$.
- *Adding an arc (t,s) with l_{ts}= u_{ts}=$|X(C)|$ and c_{ts}=0.*

▶ **Property 4** ([14]). *A costgcc C is consistent iff there is a minimum cost flow in the value network of C whose cost is less than or equal to H.*

Figure 2 represents the residual graph of the value network of the costgcc constraint defined in Figure 1. This is the graph computed from a flow resulting of the min cost flow algorithm applied on the value network. In the residual graph, the optimal solution corresponds to the arcs oriented from the variables to the values. The optimal cost value is 7.

For clarity, in the remainder, we consider that $C = costgcc(X,l,u,cost,H)$ is a costgcc constraint and that f is min cost flow in $N(C)$. We also assume that the arc consistency of the underlined gcc of C has been established.

The consistency of a value relates to the existence of a particular path in the residual graph of the min cost flow.

▶ **Property 5** ([14]). *A value a of a variable x is not consistent with C iff the two following properties hold:*

- $f_{ax} = 0$
- $d_{R(f)}(x,a) > H - cost(f) - rc_{ax}$

where $d_{R(f)}(x,a)$ is the shortest path between x and a in the residual graph of f, and rc_{ax} is the residual cost of the arc (a,x).

To establish arc consistency, the previous property could be checked for each value of each variable. However it is possible to reduce the number of computed shortest paths.

▶ **Corollary 6** ([14]). *Given any variable x and b the value of x such that $f_{bx} = 1$. Then, the value a of x is not consistent with C iff the two following properties hold:*

- $f_{ax} = 0$
- $d_{R(f)}(b,a) > H - cost(f) - rc_{ax} - rc_{xb}$

An example of the application of Property 5 is given in Figure 2. The length of the shortest path from $Julia$ to E has a cost of -1 (see blue arcs) and the cost of the arc $(E, Julia)$ is $rc_{EJulia} = 1$. Thus we have $d_{R(f)}(Julia, E) = -1$ and $H - cost(f) - rc_{EJulia} = 11-7-1 = 3$, so we have $-1 \leq 3$. From Property 5 it means that $(E, Julia)$ is consistent. The shortest path from $Peter$ to B is $d_{R(f)}(Peter, B) = 1$ and the cost of the arc $(B, Peter)$ is $rc_{BPeter} = 4$ (see red arcs). Hence, we have $H - cost(f) - rc_{BPeter} = 11 - 7 - 4 = 0$, so $1 > 0$. $(B, Peter)$ is inconsistent, the arc is then removed.

4 Upper Bounds of Shortest Paths

Although Corollary 6 reduces the number of computations required to establish the arc consistency of the constraint, it systematically computes a large number of shortest paths. Precisely, the algorithm involves computing the shortest path between each assigned value and all other values which makes it difficult to use in practice. In addition, the constraint is often arc consistent, rendering any computation useless. The aim of our approach is therefore to reduce the number of operations computed unnecessarily.

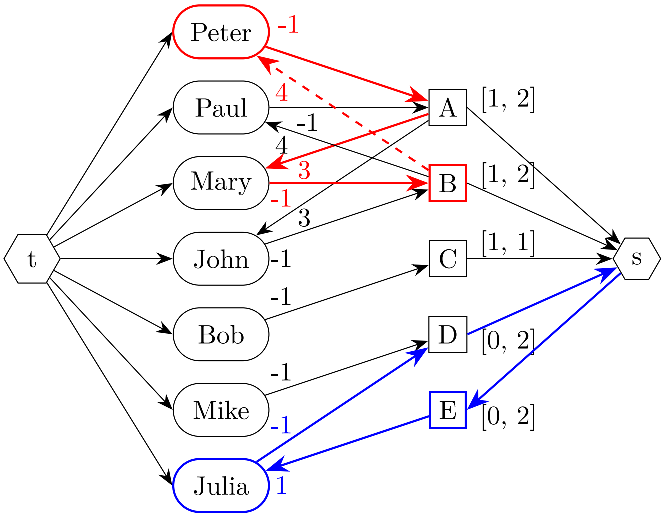


Figure 2 Example of computation of the consistency for the arcs (E, Julia) and (B, Peter). The value B is not consistent with *Peter*. Thus, the dotted arc can be removed from the graph.

We present a much more applied approach, based on the fact that Corollary 6 relies on the existence of a path of length less than a given value. It is not necessary to know the path precisely, or even to know its value. An upper bound is sufficient.

We can therefore immediately establish the following proposition:

▶ **Proposition 7.** *Let $B^+(x, a) \geq d_{R(f)}(x, a)$ be any upper bound on the length of the shortest path from x to a. If*

$$B^+(x, a) \leq H - \mathit{cost}(f) - rc_{ax}$$

then the value a of a variable x is consistent with C.

A good way of obtaining an upper bound on a distance between two points is to use the triangle inequality. Here we are talking about the triangle inequality with respect to the shortest path distances in the graph, not an embedding in Euclidean space or some other metric, which need not be present.

▶ **Property 8.** *Let x, y, and p be three nodes of a graph. According to the triangle inequality computed on shortest paths, we have:*

$$d(x, p) + d(p, y) \geq d(x, y)$$

Here, p is a particular node called landmark.

Upper bounds obtained by the triangular inequality have been shown to be useful for guiding the computation of shortest paths. The ALT algorithm, yielding excellent results in practice for computing shortest paths in a very large graph, is based on this technique [6].

Property 5 and Corollary 6 can be rewritten for landmarks:

▶ **Proposition 9.** *Given any variable x such that $f_{bx} = 1$, a any value of x and p any landmark. If one of the two condition is satisfied*

$$d_{R(f)}(x, p) + d_{R(f)}(p, a) \leq H - \mathit{cost}(f) - rc_{ax}$$
$$d_{R(f)}(b, p) + d_{R(f)}(p, a) \leq H - \mathit{cost}(f) - rc_{ax} - rc_{xb}$$

then the value a of x is consistent with C.

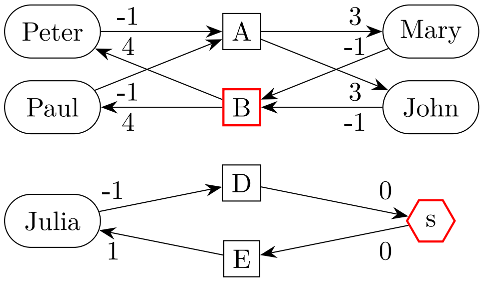


Figure 3 Example of landmark use. Nodes B and s, shown in red, are selected as landmarks.

The residual graph may have several strongly connected components. Each component must be treated separately. Thus, at least one landmark per component must be selected.

Thanks to the use of upper bounds we can go even further. It is possible to compute the consistency of all values of variables of a strongly connected component by checking a single condition.

▶ **Definition 10.** *Consider S a strongly connected component of $R(f)$, p a landmark in S, $x \in S$ a variable, and a a value of x. We define:*

- $d^{max}_{R(f)}(\cdot, p) = \max_{x \in S}(d_{R(f)}(x, p))$
- $d^{max}_{R(f)}(p, \cdot) = \max_{x \in S}(d_{R(f)}(p, x))$
- $rc^{max} = \max_{x \in S, a \in D(x)}(rc_{ax})$

This leads to the following proposition:

▶ **Proposition 11.** *Given S a strongly connected component of $R(f)$ and p a landmark in S. If*

$$d^{max}_{R(f)}(\cdot, p) + d^{max}_{R(f)}(p, \cdot) \leq H - cost(f) - rc^{max}$$

then all the values of all the variables involved in S are consistent with C.

The advantage of this method is that if the condition is satisfied, we can guarantee that all the values of a strongly connected component are consistent by computing only two shortest paths per landmark.

Figure 3 gives an example of a residual graph on which Proposition 9 or 11 can be applied. There are 2 strongly connected components $\{Peter, A, Mary, Paul, B, John\}$ and $\{Julia, D, s, E\}$. At least 2 landmarks are required (one for each component). We select B and s arbitrarily.

Thanks to Proposition 11 we see that the maximum shortest path through s is the path from D to $Julia$ with $d^{max}_{R(f)}(\cdot, S) = d(D, s) = 0$ and $d^{max}_{R(f)}(s, \cdot) = d(s, Julia) = 1$. Furthermore, the longest arc of this strongly connected component is $rc^{max} = rc_{EJulia} = 1$. Thus we have $d(D, s) + d(s, Julia) = 1$ and $H - cost(f) - rc_{EJulia} = 3$, so we have $1 \leq 3$. This confirms that all the values of variables in the strongly connected component of s are consistent with the constraint. If the Proposition 11 can guarantee that all values of variables are consistent in this strongly connected component then we can easily deduce that the Proposition 9 can also do it.

For the first strongly connected component, Proposition 9 and 11 do not guarantee the consistency of the values of the variables. It is therefore necessary to compute exact shortest paths between values and variables.

Algorithm 1 Arc Consistency Algorithm for a Strongly Connected Component.

```
ARCCONSISTENCYWITHLANDMARKS($f, R_f, S, P$);
for $p \in P$ do
    $d(p,\cdot) \leftarrow shortestPath_{R(f)}(p,\cdot)$ // shortest path in $R(f)$ ;
    $d(\cdot,p) \leftarrow shortestPath_{\overline{R}(f)}(p,\cdot)$ // shortest path in $\overline{R}(f)$, the reverse graph of $R(f)$ ;
// Check of Proposition 11 ;
$rc^{max} \leftarrow \max_{x \in S, a \in D(x)}(rc_{ax})$ ;
for $p \in P$ do
    $d^{max}_{R(f)}(\cdot,p) \leftarrow \max_{x \in S}(d_{R(f)}(x,p))$ ;
    $d^{max}_{R(f)}(p,\cdot) \leftarrow \max_{x \in S}(d_{R(f)}(p,x))$ ;
    if $d^{max}_{R(f)}(\cdot,p) + d^{max}_{R(f)}(p,\cdot) \leq H - cost(f) - rc^{max}$ then
        // all values of all variables of $S$ are consistent ;
        return ;
$\Delta \leftarrow \{a$ such that $f_{sa} > 0\}$ ;
for value $b \in \Delta$ do
    for $x$ such that $f_{bx} = 1$ do
        $\delta(b) \leftarrow \{a$ such that $a \in D(x)$ and $a \neq b\}$ ;
        $computePath \leftarrow false$ ;
        // Check of Proposition 9 ;
        for $a \in D(x)$ while not $computePath$ do
            $dpmin \leftarrow \min_{p \in P}(d_{R(f)}(x,p) + d_{R(f)}(p,a))$;
            if $dpmin > H - cost(f) - rc_{ax}$ then
                $computePath \leftarrow true$ ;
        // Check for an explicit shortest path computation ;
        if $computePath$ then
            $d_{R(f)}(b,\cdot) \leftarrow shortestPath(b,\cdot)$ ;
            for $a \in D(x)$ do
                if $d_{R(f)}(b,a) > H - cost(f) - rc_{ax} - rc_{xb}$ then remove $a$ from $D(x)$;
```

5 Improved Filtering Algorithm

We can now describe Algorithm 1, which eliminates values that are inconsistent with the constraint. The algorithm takes as parameters a min cost flow f, its residual graph $R(f)$, a strongly connected component represented by its set of nodes S and P a set of landmarks of S. This algorithm must therefore be called for each strongly connected component. The algorithm begins by checking whether Proposition 11 holds. If true, then the algorithm stops, since this means that all the values of the variables in the connected component S are consistent. Otherwise, it is necessary to check each value potentially inconsistent individually. So, for each of those values Proposition 9 is checked. If it is satisfied, then the value is consistent, otherwise an explicit shortest path is computed to determine whether the value is consistent or not.

When testing Corollary 6, we could refine the algorithm by identifying the nodes for which we need to search for a shortest path from b to them, but this is not interesting in practice as the shortest path algorithm will quickly find that they are at an acceptable distance from b.

Practical improvements

One can compute landmarks only when they are needed. This consideration is effective in practice and a simple modified version of the basic algorithm is possible. This modification proceeds by iteration over the landmarks. Consider V the set of values for which a shortest path must be computed.

The following process is defined: The landmark p is considered. Proposition 11 is checked according to p. If it is satisfied then V is emptied (all values are consistent) otherwise the values V that satisfies Proposition 9 according to p are removed from V, because they are consistent.

This process is repeated while V is not empty and some landmarks remain. In other words, the landmarks are successively considered while the status of some values is not determined.

If there are no more landmarks to compute, then, and only then, shortest paths are explicitly computed for the value in V. In practice, it is frequent to find that all values are consistent without using all the landmarks. This practical improvement means that not all landmarks need to be systematically computed.

Note that the landmark approach subsumes all the practical improvements proposed by Régin.

As far as the shortest path algorithm is concerned, it is interesting to remove the negative costs from the residual graph in order to use Dijkstra's algorithm, as mentioned by Régin. It only requires one shortest path computation [14].

Complexity

Let SP be the complexity of computing a shortest path from one node to all others. Régin's algorithm has a complexity of $\Omega(\delta \times SP)$ in the best case and $O(\delta \times SP)$ in the worst case, where δ is the number of assigned values. With landmarks, the complexity in the best case is in $\Omega(FindP + |P| \times SP)$ where $|P|$ is the number of landmarks and $FindP$ is the complexity of finding the landmarks. This complexity is obtained when Proposition 11 detects that every value is consistent. Note that, this detection can happen on the first landmark and so we can have $|P| = 1$. In the worst case, the complexity is the same as that of Régin's algorithm, provided that $|P|$ is in $O(\delta)$ and $FindP$ is in $O(\delta \times SP)$. As with the ALT method, we consider several landmarks in order to have a better chance of finding landmarks that avoid explicit shortest path computations.

6 Landmark Selection

There are different methods for selecting landmarks.

Random

A landmark is randomly selected. This method is fast to find landmarks, so we used it to compare to other methods.

Outline

The method is based on an approximation of the outlines of a graph.

▶ **Definition 12.** *The* ***outlines of a graph*** *G are defined by one or more pairs of nodes (x, y) with $x, y \in X$ that maximize the minimum distance between x and y among all pairs of nodes in the graph.*

To find the pair of nodes representing the outline, we use a well-known 2-approximation. First, we perform a shortest-path search starting from an arbitrary node x, then select the node y, which is the furthest node from x, as a landmark. Next, the shortest paths from

y are computed, and z the node furthest from y is selected as the second landmark. The outline is therefore (y, z) and the landmarks y and z. The complexity of finding a landmark depends on the complexity of computing the two shortest paths, and is therefore in $O(SP)$.

Center

The method is based on an approximation of the center of a graph.

▶ **Definition 13.** *The* ***center of a graph*** *G is defined by one or more nodes $x \in X$ that minimize the maximum distance from them to any other node in the graph.*

As the definition of the outlines and the center are similar, the selection of landmarks is also similar. We search for the outlines (x, y) with $x, y \in X$ and select as the center the node z that lies halfway between x and y. The landmark is z. The complexity is the same as for the previous method, $O(SP)$.

Outline & center

The method is based on both outlines and center of a graph, that is a pair of outlines and a center are selected as described earlier.

Maximum degree

The method is based on the node's degree. We select as a landmark the node $x \in X$ that maximizes $(deg^+(x) + deg^-(x)) \times min(deg^+(x), deg^-(x))$, where $deg^+(x)$ (resp. $deg^-(x)$) is the number of outgoing arcs of x (resp. incoming arcs to x). We used this formula to choose nodes with a large number of predecessors and successors. We also expect to choose nodes with a good balance between predecessors and successors. To find landmarks we traverse every node once, giving a complexity of $O(|X|)$.

All these methods must be applied for each strongly connected component.

7 Experimentation

The experiments were carried out on a computer with an Intel Core i7-3930K CPU 3,20 GHz processor, 64 GB of memory and running under Windows 10 Pro. All algorithms were implemented in Java (openjdk-17) in an internal CP solver.

The results relate to the solving of four problems, the traveling salesman problem (TSP) [9], the StockingCost problem [8], the flexible job shop scheduling problem (FJSSP) [13, 20] and a problem of assigning child to activities (CHILD) [19]. The TSP data are the instances (77) of the TSPLIB [5] having less than 1,500 cities. Some of them involve more than a million of edges. The StockingCost data are those used in a Houndji's paper [8], this is random data distributed define as 100 instances with 500 periods. Precisely, the StockingCost instances have 500 variables and 500 values. The FJSSP data come from two different sources, given by Pelleau [13] and Weise [20]. There are 370 instances with between 5 and 20 variables linked to a few values (between 5 and 10), and most instances have between 50 and 300 arcs. The CHILD instance contains only real-life data from [19]. There are 623 children and 317 activities. Each child must be assigned to one activity. One activity can be associated with multiple children.

For each instance of each problem, we measure the information relating to the establishment of the arc consistency of the costgcc constraint at the root of the search tree. The mean of the results for each data set are reported in the tables.

The H value of the TSP instances comes from the heuristic of Lin-Kernighan [11]. Most of the time, this value is the optimal value. For the instances StockingCost, FJSSP and CHILD the regular H is the smallest value such that there exists at least one solution and the big H is twice as large as the regular H.

It is important to pay close attention to the relationship between the value of H and the value of the minimum-cost flow. Indeed, the costgcc constraint sometimes represents a lower bound of the optimal solution, and this lower bound can be more or less distant from the optimal solution. So if H is the value of the optimal solution, then the min-cost flow may well have a much lower value. This is particularly true for the TSP problem.

Shortest paths are computed by using Dijkstra's algorithm and strongly connected components are computed by using Tarjan's algorithm.

The following abbreviations are used for the landmark selection methods: C for the center selection, O for the outline selection, C & O for the combination between center and outline, Deg for the selection based on the maximum degree and R for the random selection. In addition, line 5+ contains the minimum values for a number of landmarks ranging from 5 to 10.

7.1 Results Tables

We consider a shortest path calculation to be the calculation of the shortest paths from one node to all the others.

The number of shortest paths calculated is an important parameter for distinguishing between algorithms. Some shortest path computations cannot be avoided, particularly those required to detect inconsistent values. However, some shortest path computations are useless, as they do not allow us to establish the inconsistency of any value. Precisely, if the shortest path computation from b In Corollary 6 does not lead to any deletion of values then this path computation is useless.

Table 1 compares the number of shortest path computations performed by Régin's algorithm and by our approach as a function of the number of landmarks allowed in. The number of shortest paths required to compute landmarks are included.

Table 2 shows the average number of useless shortest path computations for each dataset. We consider that shortest path computations for landmarks are always useless, so they are always included. That is why there are never 0 computations with landmarks.

Table 3 gives the time required by each method.

7.2 Results Analysis

Table 1 shows that our approach generally computes significantly fewer shortest paths than Régin's algorithm for all landmarks selection methods. For the TSP instances, we compute on average between 2 and 47 times fewer shortest paths than Régin's algorithm. The difference is significant for all instances except for the StockingCost instances with Regular H. It should be noted that our approach is always better or equivalent and allows us to detect quickly whether the constraint is arc consistent in certain cases.

In the best case, our approach does not compute any shortest paths other than those required to determine landmarks. Our approach can compute more shortest paths only when there is no inconsistent arc and the extra computation is due to the landmarks. The number of useless path computations is also reduced by our method (See Table 2).

For computation times, we find the same kind of results as before (See Table 3). The gain average factors evolve between 1 and 57.

Table 1 Establishment of the arc consistency of a costgcc constraint: average number of computed shortest paths depending on the number of landmarks and the landmark selection method.

	Régin	Landmark Number	C	O	C & O	Deg	R
TSP (≤ 100 cities)	57.6	1	31.7	36.3	36.2	**27.7**	**27.7**
		2	35.3	39.9	39.8	32.5	29.5
		3	38	42.7	42.5	32.5	28.5
		4	41.6	46.3	46.1	32	30.1
		5+	44.8	50	50.2	32	32.2
TSP (> 100 & < 250 cities)	163.3	1	42.2	45	47.9	**40.5**	**40.5**
		2	44.4	47.4	46.3	41.6	41.6
		3	46	49.3	48.2	41.2	41.2
		4	48.6	51.9	50.8	42.3	42.3
		5+	50.2	54.1	52.2	43.1	43.3
TSP (≥ 250 cities)	662.7	1	18.1	19.8	19.8	17.8	17.8
		2	18.5	21.4	21.4	18.1	18.1
		3	18.5	21	19.3	**16.3**	**16.2**
		4	18.8	21.6	19.9	16.4	**16.3**
		5+	19	21.8	20.1	16.7	16.4
StockingCost (Regular H)	**493.3**	1	496.9	497.3	496.9	495.3	495.3
		2	500.8	501.2	500.8	497.3	497.2
		3	504.7	505.1	504.7	499.2	499.1
		4	508.6	509	508.6	501.2	501
		5+	512.5	512.9	512.6	503.2	503
StockingCost (Big H)	493.3	1	4	4	4	**2**	**2**
		2	4	4	4	**2**	**2**
		3	4	4	4	**2**	**2**
		4	4	4	4	**2**	**2**
		5+	4	4	4	**2**	**2**
FJSSP (Regular H)	10.4	1	8.3	5.1	4.8	**2**	6.3
		2	8.3	5.1	4.8	**2**	5.3
		3	8.3	5.1	4.8	**2**	4.6
		4	8.3	5.1	4.8	**2**	4
		5+	8.3	5.1	4.8	**2**	4
FJSSP (Big H)	10.4	1	4.5	4.3	4.3	**2**	3.2
		2	4.5	4.3	4.3	**2**	2.8
		3	4.5	4.3	4.3	**2**	2.6
		4	2.9	4.3	4.3	**2**	2.4
		5+	2.9	4.3	4.3	**2**	2.4
CHILD (Regular H)	**108**	1	112	112	112	109	110
		2	116	116	116	111	112
		3	120	120	120	113	114
		4	124	124	124	115	116
		5+	128	128	128	117	118
CHILD (Big H)	108	1	4	4	4	**2**	**2**
		2	4	4	4	**2**	**2**
		3	4	4	4	**2**	**2**
		4	4	4	4	**2**	**2**
		5+	4	4	4	**2**	**2**

7.2.1 Landmark Number and Selection Method

We can see that the results do not change much as a function of the number of landmarks. The major part of problems have best or equivalent results with 4 landmarks, but the difference is minimal. When it is not mentioned 4 landmarks are used.

Table 2 Establishment of the arc consistency of a costgcc constraint: number of average shortest paths computed uselessly with 4 landmarks.

	Régin	C	O	C & O	Deg	R
TSP ($\leq$ 100 cities)	35.8	19.7	24.4	24.3	**8.3**	**8.2**
TSP ($>$ 100 & $<$ 250 cities)	131.1	16	19.7	18.6	**10.1**	**10.1**
TSP ($\geq$ 250 cities)	649.8	5.9	8.6	6.9	**3.4**	**3.3**
StockingCost (Regular H)	**0**	3.2	7.6	11.2	7.8	7.6
StockingCost (Big H)	493.3	4	4	4	**2**	**2**
FJSSP (Regular H)	**0**	8.3	5.1	4.8	4	4
FJSSP (Big H)	10.1	2.9	4.3	4.3	**2**	2.4
CHILD (Regular H)	**0**	16	16	16	8	8
CHILD (Big H)	108	4	4	4	**2**	**2**

Table 3 Establishment of the arc consistency of a costgcc constraint: computation times (in ms) and ratio. Experimentation with 4 landmarks.

		Régin	C	O	C & O	Deg	R
TSP ($\leq$ 100 cities)	Mean	7.3	5.9	6	6.6	5.7	**4.5**
	Median	3.4	3.6	4.4	4.1	3.6	**3.3**
	Ratio		1.2	1.2	1.1	1.3	**1.6**
TSP ($>$ 100 & $<$ 250 cities)	Mean	76.6	**29.8**	30.6	30.2	**28.6**	31.1
	Median	51.2	**14.3**	16	17	15.4	**14.3**
	Ratio		**2.6**	2.5	2.5	**2.7**	2.5
TSP ($\geq$ 250 cities)	Mean	12124.9	278.9	275.2	275.4	**213**	265
	Median	2310.2	126.8	117.7	90.6	89.1	**85.9**
	Ratio		43.5	44.1	44	**56.9**	45.8
StockingCost (Regular H)	Mean	603.83	**511.8**	617.9	626.2	580.3	639.4
	Median	585.7	553.3	186.9	186.4	248	**166.4**
	Ratio		**1.2**	1	1	1	0.9
StockingCost (Big H)	Mean	534.76	34.1	32.4	**31.6**	33.2	32.6
	Median	519.1	33.8	32.4	31.9	32.8	**30.1**
	Ratio		15.7	16.5	**16.9**	16	16.4
FJSSP (Regular H)	Mean	0.4	0.5	**0.3**	0.4	0.4	0.5
	Median	**0.1**	0.3	0.2	0.3	0.2	0.3
	Ratio		0.8	**1.7**	0.75	1	0.8
FJSSP (Big H)	Mean	0.4	0.4	**0.3**	**0.3**	**0.3**	**0.3**
	Median	**0.1**	0.2	0.2	0.2	0.2	0.2
	Ratio		1	**1.3**	**1.3**	**1.3**	**1.3**
CHILD (Regular H)	Time	65.1	69.2	**54.4**	67.6	75.9	65.4
	Ratio		0.9	**1.2**	1	0.8	1
CHILD (Big H)	Time	58.2	7	6.5	7.3	**6**	**6**
	Ratio		8.3	9	8	**9.7**	**9.7**

Two methods of landmark selection appear to be more effective in practice: the method based on maximum node degrees and the random node selection method. As there is little difference between these two methods, and the former is more robust than the latter, we recommend defining landmarks based on maximum degree nodes.

7.2.2 Impact of the practical improvement of Section 5

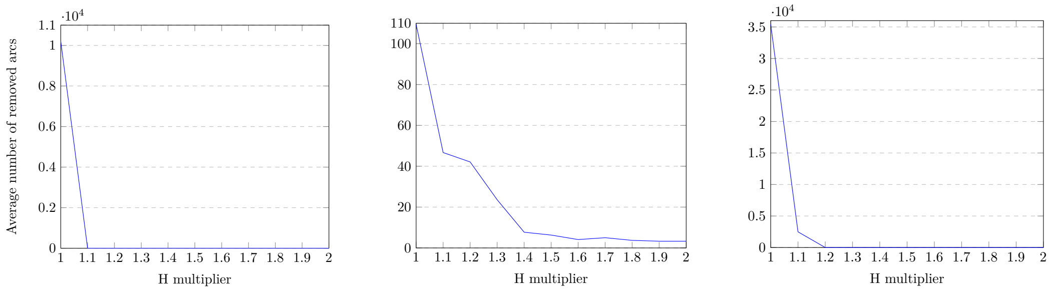
Thanks to this practical improvement all the authorized landmarks are not systematically used. This is clearly seen for StockingCost and CHILD instances with big H. The computation of a single landmark is sufficient to guarantee that all the values are consistent.

7.2.3 StockingCost, FJSSP and CHILD problems

For the StockingCost, FJSSP and CHILD problems, the results strongly depends on the value of H.

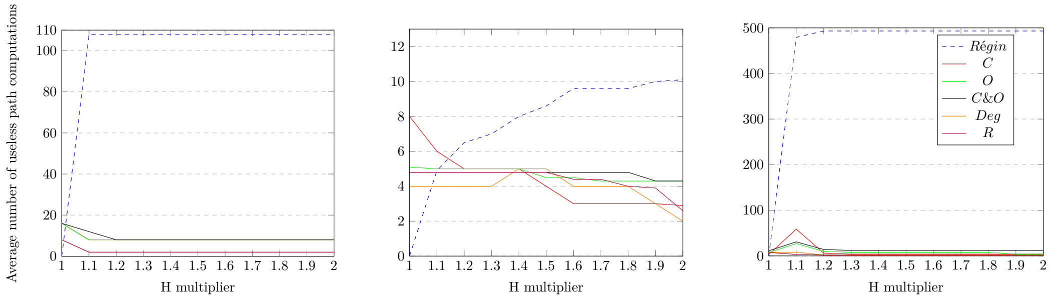
For the Regular H value the results are similar to those of Régin's algorithm. In these problems, Regular H is close to the optimal value of the min cost flow of the underlined costgcc. Thus, there is less margin and therefore more inconsistent values. FJSSP instances are also small and do not allow us to highlight the usefulness of landmarks. Indeed, in a small instance, computing a landmark gives us access to less information than in a large instance. In addition, for practical use, it is more interesting to save time on large instances since they take longer to resolve than on small instances which are already quick to resolve.

For a Big H value the landmark method clearly outperforms Régin's algorithm.



(a) CHILD. **(b)** FJSSP. **(c)** StockingCost.

Figure 4 Evolution of the average number of removed arcs for the CHILD, FJSSP and StockingCost instances in function of the multiplier of H.



(a) CHILD. **(b)** FJSSP. **(c)** StockingCost.

Figure 5 Evolution of the average number of useless path computations for CHILD, FJSSP and StockingCost instances in function of the multiplier of H. The experimentation involves 4 landmarks.

Figures 4 and 5 provide information on the relationship between H values and the number of useless path computations. The landmark approach performs very well as soon as the H value deviates a little from the optimal value, in other words, as soon as there is a little margin and therefore fewer inconsistent values.

7.2.4 TSP results

The improvement brought about by our approach for instances from the TSP problem are strong. This is mainly due to the relationship between the H value given by the TSP value and the underlying costgcc constraint. In the case of the TSP, the optimal value of the min

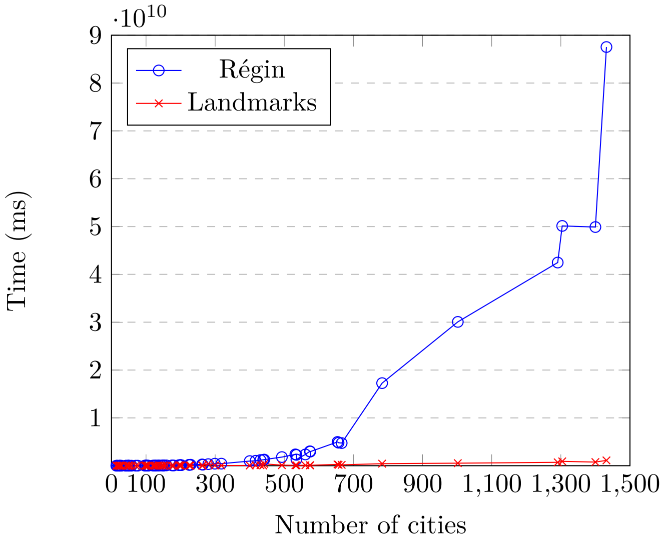


Figure 6 Evolution of the time in relation to the size of TSP instances. The landmarks are selected with the degree method and 4 landmarks. The blue plot with circles is the time of the Régin algorithm and the red with crosses is the time of the landmarks algorithm.

cost flow is lower than H, because the costgcc constraint only models a part of the problem and in fact represents a real relaxation. So even with an optimal H value for TSP, there is a margin for the costgcc constraint.

To better appreciate the performance of the landmarks, Figure 6 shows the evolution of time in milliseconds as a function of the size of the TSP dataset instances. The blue plot with circles shows the time taken by the Régin's algorithm, while the red plot with crosses shows the time taken by the algorithm using the maximum degrees and 4 landmarks. Clearly, the use of landmarks is drastically faster than the Régin's algorithm. The larger the instance, the more useful landmarks become.

Figure 7 shows the evolution of the speed-up ratio (Régin time/Landmarks time) on the instances of the TSP dataset. The landmark selection algorithm is based on maximum degrees with 4 landmarks. We can also see in this graph that the more data there is, the higher the gain factor. As mentioned above, this can be explained by the fact that in a large structure, the landmarks contain a lot of information compared with a smaller structure. In these experiments, note that if we omit the assigned variables there is only one strongly connected component in the value network. Overall, we find that our algorithm significantly speeds up the previous approach, up to about 80 times faster for large problems.

8 Conclusion

This paper proposes an efficient implementation of the arc consistency algorithm of the cardinality constraint with costs. This constraint is present in many industrial problems and the establishment of the arc consistency is often too slow to be used in practice, as it is based on finding the shortest paths from the assigned values. We introduce a new method that uses upper bounds on shortest paths based on triangular inequalities and landmarks. This approach avoids the computation of many shortest paths and improves the computation time of the arc consistency filtering algorithm. The larger the graph and the larger the margins, the greater the improvement will be. In addition, we have introduced a sufficient condition, which is quick to compute, for a costgcc constraint to be arc consistent.

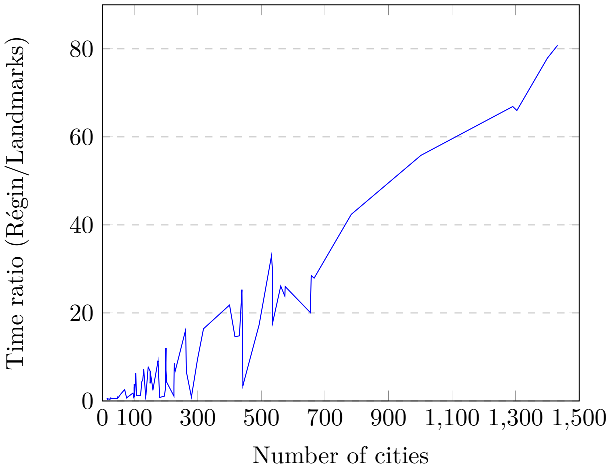


Figure 7 Evolution of the speedup ratio in relation to the size of TSP instances. The landmarks are selected with the degree method and 4 landmarks.

References

1 R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice hall, 1993.

2 C. Berge. *Graphs and Hypergraphs.* Elsevier Science Ltd, 1985.

3 G. Demassey, S.and Pesant and L.-M. Rousseau. A Cost-Regular Based Hybrid Column Generation Approach. *Constraints*, 11(4):315–333, December 2006. doi:10.1007/s10601-006-9003-7.

4 S. Ducomman, H. Cambazard, and B. Penz. Alternative Filtering for the Weighted Circuit Constraint: Comparing Lower Bounds for the TSP and Solving TSPTW. In *AAAI 2016*, Phoenix, United States, February 2016. URL: https://hal.science/hal-01420964.

5 R. Gerhard. TSPLIB–a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

6 A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 156–165, USA, 2005. Society for Industrial and Applied Mathematics.

7 S. Gualandi and F. Malucelli. Constraint Programming-based Column Generation. *Annals of Operations Research*, 204(1):11–32, April 2013. doi:10.1007/s10479-012-1299-7.

8 V. R. Houndji, P. Schaus, and L. Wolsey. The item dependent stockingcost constraint. *Constraints*, 24(2):183–209, April 2019. doi:10.1007/s10601-018-9300-y.

9 N. Isoart. *The traveling salesman problem in constraint programming.* Theses, Université Côte d'Azur, November 2021. URL: https://theses.hal.science/tel-03554009.

10 E. Lawler. *Combinatorial optimization - networks and matroids.* Holt, Rinehart and Winston, New York, 1976.

11 S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Oper. Res.*, 21:498–516, 1973. URL: https://api.semanticscholar.org/CorpusID:33245458.

12 P. Nightingale, Ö. Akgün, I. P. Gent, C. Jefferson, I. Miguel, and P. Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, 2017. doi:10.1016/j.artint.2017.07.001.

13 M. Pelleau, A. Miné, C. Truchet, and F. Benhamou. A constraint solver based on abstract domains. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pages 434–454, 2013. doi:10.1007/978-3-642-35873-9_26.

14 J.-C. Régin. Cost-based arc consistency for global cardinality constraints. *Constraints*, 7(3/4):387–405, July 2002. doi:10.1023/A:1020506526052.

15 J.-C. Régin. Filtering algorithm for constraints of difference in csps. In *Proceedings of the National Conference on Artificial Intelligence*, volume 1, July 1994.

16 J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings AAAI'96*, pages 209–215, January 1996.

17 R. E. Tarjan. *Data structures and network algorithms.* Society for Industrial and Applied Mathematics, USA, 1983.

18 W.-J. Van Hoeve, G. Pesant, and L.-M. Rousseau. On global warming: Flow-based soft global constraints. *Journal of Heuristics*, 12(4):347–373, September 2006. doi:10.1007/s10732-006-6550-4.

19 S. Varone and C. Beffa. Dataset on a problem of assigning activities to children, with various optimization constraints. *Data in Brief*, 2019.

20 T. Weise. jsspinstancesandresults: Results, data, and instances of the job shop scheduling problem, 2019–2020. A GitHub repository with the common benchmark instances for the Job Shop Scheduling Problem as well as results from the literature, both in form of CSV files as well as R program code to access them. URL: https://github.com/thomasWeise/jsspInstancesAndResults.

21 X. Zhang, J. Gao, Y. Lv, and W. Zhang. Early and efficient identification of useless constraint propagation for alldifferent constraints. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 1126–1133. International Joint Conferences on Artificial Intelligence Organization, July 2020. Main track. doi:10.24963/ijcai.2020/157.

22 X. Zhang, Q. Li, and W. Zhang. A fast algorithm for generalized arc consistency of the alldifferent constraint. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1398–1403. International Joint Conferences on Artificial Intelligence Organization, July 2018. doi:10.24963/ijcai.2018/194.

CP for Bin Packing with Multi-Core and GPUs

Fabio Tardivo [1]
Department of Computer Science, New Mexico State University, Las Cruces, NM, USA

Laurent Michel
Synchrony Chair in Cybersecurity, School of Computing, University of Connecticut,
Storrs, CT, USA

Enrico Pontelli
Department of Computer Science, New Mexico State University, Las Cruces, NM, USA

Abstract

The *BinPacking* constraint models the requirements of many logistics, resource allocation, and production scheduling applications. This paper explores new avenues based on the impressive computational power of modern GPUs to propagate the *BinPacking* constraint. This work showcases how the perspective of massive parallelization can lead to novel approaches, such as the use of a portfolio of lower bounds, to enhance the pruning of the *BinPacking* constraints. It delivers insights into the design choices and challenges presented by GPU platform for constraint propagation.

The paper evaluates a GPU-accelerated propagator against both sequential and parallel CPU versions, as well as state-of-the-art approaches. Comparisons across various benchmarks from the literature show strong performances with respect to both CPU versions and the standard pruning approach. When compared to techniques based on Linear Programming, our approach proves valuable for large instances or when spending extensive time to obtain the best possible bound is not convenient.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Massively parallel algorithms

Keywords and phrases Constraint Propagation, Bin Packing, Parallelism, GPU, Lower Bounds

Digital Object Identifier 10.4230/LIPIcs.CP.2024.28

Related Version *Previous Version*: `https://arxiv.org/abs/2402.14821`

Supplementary Material
Software: `https://bitbucket.org/constraint-programming/fzn-minicpp/src/gpu`
archived at `swh:1:dir:b7e3d8e8a2f41b87654cd2e2a81a2622099a8cf0`
Dataset: `https://bitbucket.org/constraint-programming/minicpp-benchmarks` [40]
archived at `swh:1:dir:1dfa371fc284829abddc0a89d1e60251c07d6c84`

Funding *Enrico Pontelli and Fabio Tardivo*: Supported by NSF grants 2151254, 1914635 and by a contract from Wallaroo Inc.
Laurent Michel: Partially supported by Synchrony.

Acknowledgements We would like to thank François Clautiaux, Maxence Delorme, Jürgen Rietz for their feedbacks, and Hadrien Cambazard for providing the Arc-Flow implementation.

[1] Corresponding author

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 28; pp. 28:1–28:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The *Bin Packing Problem (BPP)* consists of packing a set of items into the minimal number of bins, each with a fixed capacity. It has a fundamental role in logistics, resource allocation, and production scheduling applications. Because of its relevance, the Bin Packing Problem has been extensively studied over the last decades, both theoretically and practically. We refer interested readers to [10, 36] for a comprehensive review.

The BPP is NP-Hard in the strong sense [17] and it is challenging to solve even for a fixed number of bins [21] or a constant number of different item sizes [19]. Techniques based on Integer Linear Programming (ILP) are highly effective and represent the state-of-the-art for solving the BPP. When the BPP is a component of a larger problem, applying such techniques becomes challenging, and *Constraint Programming (CP)* emerges as a valuable alternative. There, the BPP often appears in its decision version, where the items must be packed into a fixed number of bins.

The decision variant is modeled in CP using the *BinPacking* constraint [38]. Its filtering algorithm employs an *approximated knapsack reasoning* to exclude or commit items to bins, and a *feasibility check* to prune the search if the remaining unpacked items cannot fit in the residual space. The check is performed using a lower bound on the number of bins necessary to pack the items.

The contributions of this paper are as follows:

1. describe a propagator architecture based on *parametric families* of lower bounds and their role in a portfolio setting;
2. demonstrate how the large number of bounds from those parametric families should be computed in parallel to derive the most value. In particular, the paper demonstrates that sampling bounds in a sequential or multi-core implementation is substantively weaker;
3. provide an implementation of a GPU-accelerated portfolio of lower bounds within a constraint propagator of a standard CP solver;
4. deliver an empirical evaluation comparing sequential, multi-threaded, and GPU-accelerated computation of those lower bounds, with other state-of-the-art approaches on different benchmarks.

The rest of the paper is organized as follows. Section 2 contains some general background about Constraint Satisfaction/Optimization Problems and General-Purpose computing on Graphics Processing Units (GPGPU). Section 3 summarizes related works on the *BinPacking* constraint, and on lower bounds for the Bin Packing Problem. Section 4 details the design process and implementation of the *BinPacking* propagator enhanced with the GPU-accelerated portfolio of lower bounds. Section 5 presents the results of our approach and the other techniques in the literature. Finally, Section 6 concludes the paper.

2 Background

2.1 Constraint Satisfaction/Optimization Problems

A Constraint Satisfaction Problem (CSP) is defined as a triplet $\langle X, D, C\rangle$, where $X = \{x_1, \ldots, x_n\}$ is a set of *variables*, $D = \{D_1, \ldots, D_n\}$ is a set of *domains*, and C is a set of *constraints*. Each domain $D_i \in D$ is a *finite* set of values. Each constraint $c \in C$ involves a subset of m variables $vars(c) = \{x_{i_1}, \ldots, x_{i_m}\} \subseteq X$, depending on its semantic. A constraint defines a relation $c \subseteq D_{i_1} \times \cdots \times D_{i_m}$. A *solution* is an assignment $\sigma : X \to \bigcup_{i=1}^{n} D_i$ such that $\sigma(x_i) \in D_i$ holds for every variable, and $\langle \sigma(x_{i_1}), \ldots, \sigma(x_{i_m})\rangle \in c$ holds for every constraint.

A Constraint Optimization Problem (COP) is a quadruplet $\langle X, D, C, f\rangle$ where $\langle X D, C\rangle$ is a CSP and $f : D_1 \times \cdots \times D_n \rightarrow \mathbb{R}$ is an *objective function* to be (w.l.o.g.) minimized. The goal is to find a solution σ^* that minimizes $f(\sigma(x_1), \cdots, \sigma(x_n))$.

A *constraint solver* searches for solutions of a CSP/COP by alternating non-deterministic choices and constraints propagation. The first is employed to choose the next variable and which value, from its current domain, to assign to it. The second is a method to filter the domain of the variables, removing values that are not part of any solution. Non-deterministic choices are typically implemented through backtracking and heuristic decisions that follow an ordering among variables and values. Constraint propagation is commonly implemented through a queue that tracks constraints that need to be re-evaluated. When a value is removed from a variable's domain, the constraints involving such variable are enqueued. The re-evaluation consists of extracting the constraint from the queue and applying the associated filtering algorithm or *propagator*. This iterative cycle continues until the queue is empty [23].

Filtering algorithms offer trade-offs between filtering power and computational complexity. Highly effective algorithms have been developed for *global constraints*. These constraints model a substantial portion of a CSP/COP and naturally arise in many problems.

2.2 General-Purpose Computing on Graphics Processing Units

Performance in modern hardware is the by-product of parallel computing resources in the form of multi-core central processing units (CPUs) and general purpose graphical processing units (GPUs). Modern commodity hardware features CPUs with up to 64 cores (e.g., AMD Ryzen Threadripper 7980X) and GPUs with up to 16384 cores per card (e.g., NVIDIA GeForce RTX 4090). Yet, the number of cores in CPUs and GPUs are orders of magnitude apart, the programming models are wildly different and GPUs impose restrictions on code to deliver performance.

The massive parallelism of GPUs is a golden opportunity. To harness such computing power, it is crucial to employ approaches and algorithms that align with the underlying architecture of the GPU. Recent studies indicate that GPUs can be used for computational logic, including applications like Satisfiability [8, 7], Answer Set Programming [12, 13], and Constraint Programming [41, 42].

GPU-accelerated applications rely on APIs that expose parallel computing primitives. The most prominent is *CUDA*, a C/C++ API, introduced by NVIDIA for its own GPUs [28]. In a typical GPU-accelerated application, the GPU handles only the most computationally demanding tasks. The CPU executes the main application logic and choreographs the GPU(s) activities such as data transfers as well as computing tasks known as *kernels*. The components of an NVIDIA GPU utilized for general-purpose computing are depicted in Figure 1. A current high-end GPU[2] is equipped with 128 *Streaming Multiprocessors (SM)*, each housing 128 computational units named *CUDA Cores*, and 128 KB of fast memory. This memory serves as *L1 cache* and/or scratchpad memory, in which case it is referred to as *shared memory*. In the middle and lower tiers of the memory hierarchy, there is an *L2 cache* of 72 MB and the *global memory* with a capacity of 24 GB.

The CUDA execution model is *Single-Instruction Multiple-Thread (SIMT)*, where a C/C++ function known as *kernel* is executed by many threads. Each thread utilizes its own unique index to identify the data to use or to modify its control flow. When different threads follow distinct control flows, it leads to *thread divergence*. In such scenarios, threads are

[2] NVIDIA GeForce RTX 4090

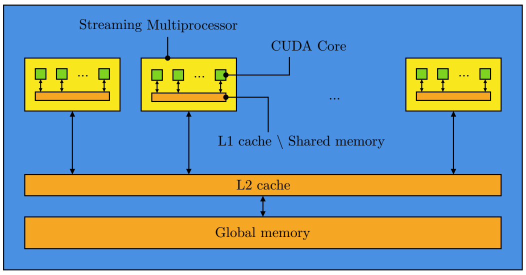


Figure 1 High level architecture of an NVIDIA GPU.

serialized, causing significant performance deterioration. Threads are organized into *blocks*, which are dispatched to the Streaming Multiprocessors. Each Streaming Multiprocessor executes the threads using its CUDA Cores, allowing efficient intra-block operations through shared memory. Communication between blocks is possible only through the use of global memory. To successfully leverage GPUs to accelerate expensive computations, it is essential to understand that they are designed to heavily trades raw execution speed for massive parallelization [20]. This often necessitates reformulating the problem to expose parallelism or exploiting shared memory to reduce costly global memory accesses.

In contrast, execution on multi-core CPUs relies on a *small number* of independent computing threads that execute fast, can have diverging behaviors with no performance penalties. Such an architecture can more readily adopt sequential code with the trade-off being the small number of threads (dozens rather than tens of thousands).

3 Bin Packing

Let $I = (c, W)$ be an instance of the Bin Packing Problem (BPP) with n items of weights $W = [w_1, \dots, w_n]$, and bins of capacity c. The underling optimization problem can be formalized as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{j=1}^{n} y_j & \\
\text{subject to} \quad & \sum_{i=1}^{n} w_i x_{ij} \leq c y_j & j = 1, \dots, n \\
& \sum_{j=1}^{n} x_{ij} = 1 & i = 1, \dots, n \\
& y_j \in \{0, 1\} & j = 1, \dots, n \\
& x_{ij} \in \{0, 1\} & i, j = 1, \dots, n
\end{aligned}$$

where the variable y_j indicates whether the j^{th} bin is used and the variable x_{ij} indicates whether the i^{th} item is packed in the j^{th} bin.

One of the most effective approaches to solving the BPP involves adopting a graph-theoretical perspective. In the *Arc-Flow* method [9], a graph is constructed such that arcs represent items, and a path from the source s to the sink t represents a set of items that can be packed into a bin (see Figure 2). A solution corresponds to a minimum flow that utilizes one arc for each item $w \in W$. This flow problem is formulated and solved using an Integer Linear Programming (ILP) model with a robust linear relaxation, albeit with a pseudo-polynomial number of variables and constraints.

Algorithm 1 Simplified propagator for the *BinPacking* constraint.

```
Procedure: $propagate(c, W, k, X, L)$
for $j \leftarrow 1$ to $k$ do
    $doLoadCoherence(j, X, W, L)$
    $doBasicLoadTightening(j, X, W, L)$                    // Basic filtering
    for $i \in \{i \mid j \in x_i \wedge |x_i| > 1\}$ do
        $doBasicItemEliminationCommitment(i, j, X, W, L)$
for $j \leftarrow 1$ to $k$ do                          // Knapsack filtering
    if $\neg isBinPackable(j, X, W, L)$ then $Fail$
    $doKnapsackLoadTightening(j, X, W, L)$
    for $i \in \{i \mid j \in x_i \wedge |x_i| > 1\}$ do
        $doKnapsackItemEliminationCommitment(i, j, X, W, L)$
$lb \leftarrow getLowerBound(c, W, k, X)$                  // Feasibility check
if $lb > k$ then $Fail$
```

In CP, the decision version of the BPP, where the items must be packed in at most k bins, is modeled as:

$$x_i = \{1, \dots, k\} \qquad i = 1, \dots, n$$
$$l_j = \{0, \dots, c\} \qquad j = 1, \dots, k$$
$$BinPacking(W = [w_1, \dots, w_n], X = [x_1, \dots, x_n], L = [l_1, \dots, l_k])$$

where the variable x_i represents the bins in which the i^{th} item can be packed, and the variable l_j represents the loads that the j^{th} bin can have. The *BinPacking* constraint was introduced in [38] and a simplified version of its filtering algorithm is listed in Algorithm 1. The following offers a brief description of each call in Algorithm 1:

doLoadCoherence Adjust the minimum/maximum load of a bin based on the total weight of the items and the load of the other bins.

doBasicLoadTightening Adjust the minimum/maximum load of a bin based on the sum of the items that are or can be packed in the bin.

doBasicItemEliminationCommitment An item is committed to a bin if it is needed to reach a valid load. An item is excluded from a bin if packing it would lead to an excessive load.

isBinPackable Checks whether a bin is packable based on an approximated knapsack reasoning to reach an admissible load.

doKnapsackLoadTightening Adjust the minimum/maximum load of a bin with an approximated knapsack reasoning.

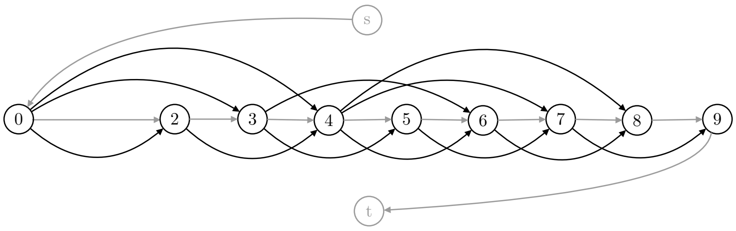
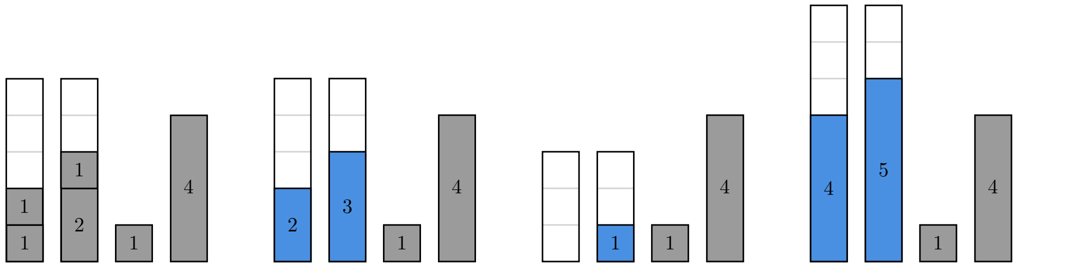


Figure 2 Graph underling Arc-Flow for an instance with $c = 9$ and $W = [4, 4, 3, 3, 2, 2]$.



(a) Partial Solution. **(b)** Reduced Solution with R_0 (merge items). **(c)** Reduced Solution with R_{Min} (Reduce bins by smallest virtual item). **(d)** Reduced Solution with R_{Max} (Enlarge bins and virtual items).

Figure 3 Illustrations of a partial packing of the instance $I = (5, [4, 2, 1, 1, 1, 1])$, and reductions $R_0 = (5, [4, 3, 2, 1])$, $R_{Min} = (3, [4, 1, 1])$, $R_{Max} = (7, [5, 4, 4, 1])$. Virtual items are colored in blue.

getLowerBound A partial packing is considered feasible if a lower bound on the number of bins does not exceed the number of available bins. This lower bound, referred to as L_2 (see Section 3.2), is calculated on a *reduced* instance derived from the current partial packing (see Section 3.1).

The literature contains various enhancement of the *BinPacking* constraint. The authors of [35, 30, 11] introduced and refined a *cardinality reasoning*, well suited when there are constraints on the number of items in each bin or when the items have similar weights. In [3], it was employed a tight lower bound derived from the *linear relaxation* of the Arc-Flow model.

3.1 Reductions

Given a partial packing of an instance $I = (c, W)$, a reduction R provides an instance $I_R = (c_R, W_R)$ such that a lower bound for I_R is valid for the partial packing. Such partial packing is inferred from the variables X.

The standard reduction, knows as R_0, maintains the same capacity, all the unpacked items, and introduces *virtual items* representing the items packed in each bin (see Figure 3b). Other reductions similar to R_0 are possible. For instance, [14] introduced R_{Min} and R_{Max}. The first decreases the capacity of the bins and the virtual items by the size of the smallest virtual item (see Figure 3c). The second increases the capacity of the bins and the virtual items by a common quantity, so that two virtual items can not fit in the same bin. This is achieved when each virtual item is bigger than half of the bin capacity (see Figure 3d).

3.2 Lower bounds

Given an instance, $I = (c, W)$, a lower bound $L(I)$ estimates the minimum number of bins necessary to store the items. The simplest lower bound is referred to as L_1, and is calculated as follows:

$$L_1(I) = \left\lceil \frac{1}{c} \sum_{w \in W} w \right\rceil$$

where the total weight of the items is divided by the bin capacity, and the ceiling function is applied. This approach is equivalent to naively packing the items, cutting those that do not entirely fit.

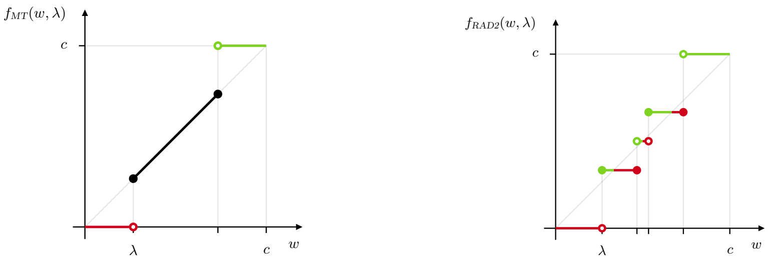


Figure 4 f_{MT} (left) and f_{RAD2} (right) for $\lambda = c\frac{4}{15}$. Weights that have been increased/decreased are shown in green/red.

An improvement of L_1, called L_2, was introduced in [25] and addresses the cases where big items cannot be packed together. It is defined as:

$$L_2(I) = \max_{0 \le \lambda \le \frac{c}{2}} L_2(I,\lambda)$$

where

$$L_2(I,\lambda) = |W_1| + |W_2| + max\left(0, \left\lceil \frac{1}{c}\left(\sum_{w \in W_3} w - \left(c\,|W_2| - \sum_{w \in W_2} w\right)\right)\right\rceil\right)$$
$$W_1 = \{w \mid w \in W \land c - \lambda < w\}$$
$$W_2 = \{w \mid w \in W \land \tfrac{c}{2} < w \le c - \lambda\}$$
$$W_3 = \{w \mid w \in W \land \lambda \le w \le \tfrac{c}{2}\}$$

The lower bound $L_2(I,\lambda)$ classifies the items as big (W_1), medium-big (W_2), medium-small (W_3), while it ignores the smallest items. Note how the definition of the sets are *parameterized* by λ. Then all the big and medium-big items are packed in different bins since they are all bigger than $\frac{c}{2}$. The medium-small items are packed as in L_1, using the available space in the bins where there is a medium-big item before considering other bins. Finally, the small items are just dropped. A direct implementation of L_2 is pseudo-polynomial, since $L_2(I,\lambda)$ has to be calculated exactly once for each $\lambda \in \left[0, \frac{c}{2}\right]$, i.e., $\Theta(c)$ times. A linear complexity can be achieved when the items are sorted in decreasing weight [24, 22]. Note how $L_2(I)$ defines a *family of lower bounds*, with one member for each $\lambda \in \left[0, \frac{c}{2}\right]$.

A general approach to enhance L_1, derived from duality theory, is based on *Dual Feasible Functions (DFFs)* [1]. Intuitively, a function $f : \mathbb{N}_0 \to \mathbb{N}_0$ is dual feasible if, for every subset $W_S \subseteq W$, the following holds:

$$\sum_{w \in W_S} w \le c \quad \Rightarrow \quad \sum_{w \in W_S} f(w) \le f(c)$$

Consider the $f_{MT}(w,\lambda)$ definition below that keeps the same capacity, while defining new weights for items. It increases the weights of large items ($c - \lambda < w$) to c, decrease the weights of small items ($w < \lambda$) to 0, and leave the weights of medium items unchanged ($\lambda \le w \le c - \lambda$), i.e., they are w. Note that increasing the weight to c is equivalent to allocating an entire bin for the item, while decreasing the weight to 0 disregards the item. The function, shown in Figure 4, depends on an *integer* parameter λ:

$$\underset{0 \le \lambda \le \frac{c}{2}}{f_{MT}(w,\lambda)} = \begin{cases} c & \text{if } c - \lambda < w \\ w & \text{if } \lambda \le w \le c - \lambda \\ 0 & \text{if } w < \lambda \end{cases}$$

The lower bound obtained by combining L_1 with f_{MT} is:

$$L_{MT}(I) = \max_{0 \le \lambda \le \frac{c}{2}} \left\lceil \frac{1}{f_{MT}(c,\lambda)} \sum_{w \in W} f_{MT}(w,\lambda) \right\rceil \tag{1}$$

and it is equal to L_2 [16]. Other DFFs have been proposed, each with a different design for revising weights. For brevity, we report only some of them and refer interested readers to [6, 1] for a comprehensive review, and to [31, 32] for further insights.

$$\underset{\frac{c}{4} < \lambda \le \frac{c}{3}}{f_{RAD2}(w,\lambda)} = \begin{cases} 0 & \text{if } w < \lambda \\ \left\lfloor \frac{c}{3} \right\rfloor & \text{if } \lambda \le w \le c - 2\lambda \\ \left\lfloor \frac{c}{2} \right\rfloor & \text{if } c - 2\lambda < w < 2\lambda \\ c - f_{RAD2}(c - w, \lambda) & \text{if } 2\lambda \le w \end{cases}$$

$$\underset{1 \le \lambda \le 100}{f_{FS1}(w,\lambda)} = \begin{cases} w\lambda & \text{if } \frac{w(\lambda+1)}{c} \in \mathbb{N} \\ \left\lfloor \frac{w(\lambda+1)}{c} \right\rfloor c & \text{otherwise} \end{cases}$$

$$\underset{1 \le \lambda \le \frac{c}{2}}{f_{CCM1}(w,\lambda)} = \begin{cases} 2\left\lfloor \frac{c}{\lambda} \right\rfloor - 2\left\lfloor \frac{c-w}{\lambda} \right\rfloor & \text{if } w > \frac{c}{2} \\ \left\lfloor \frac{c}{\lambda} \right\rfloor & \text{if } w = \frac{c}{2} \\ 2\left\lfloor \frac{w}{\lambda} \right\rfloor & \text{if } w < \frac{c}{2} \end{cases}$$

$$\underset{2 \le \lambda \le c}{f_{VB2}(w,\lambda)} = \begin{cases} 2\max\left(0, \left\lceil \frac{c\lambda}{c} \right\rceil - 1\right) - 2\max\left(0, \left\lceil \frac{(c-w)\lambda}{c} \right\rceil - 1\right) & \text{if } w > \frac{c}{2} \\ \max\left(0, \left\lceil \frac{c\lambda}{c} \right\rceil - 1\right) & \text{if } w = \frac{c}{2} \\ 2\max\left(0, \left\lceil \frac{w\lambda}{c} \right\rceil - 1\right) & \text{if } w < \frac{c}{2} \end{cases}$$

$$\underset{1 \le \lambda \le c}{f_{BJ1}(w,\lambda)} = \begin{cases} \left\lfloor \frac{w}{\lambda} \right\rfloor (\lambda - c \bmod \lambda) & \text{if } w \bmod \lambda \le c \bmod \lambda \\ \left\lfloor \frac{w}{\lambda} \right\rfloor (\lambda - c \bmod \lambda) + w \bmod \lambda - c \bmod \lambda & \text{otherwise} \end{cases}$$

Interestingly, these five definitions are all parametric in λ and define 5 additional families, most with $\Theta(c)$ members (except f_{FS1}). To get the best possible bound, one would need to compute the bounds for each family and across all parameter values in that family. To reduce the sequential computational burden, one could resort to only computing some families, or computing only a subset of different λ values in each admissible range. Alternatively, one can adopt parallel techniques as *all* families and *all* λ values can be computed independently. The next section studies this tradeoff.

4 Design and Implementation

To determine the most convenient DFF to use, we examined the lower bounds derived from various DFFs on the Falkenauer and Scholl instances (see Section 5). The results in Table 1 confirm f_{CCM1} as the best overall function [6], while the generally weak f_{RAD2} proves effective

Table 1 Statistics for different DFF-based lower bounds on the Falkenauer and Scholl instances.

DFF	Only Opt	Total Opt	Only Best	Total Best	Sum
f_{MT}	2	1151	0	55	120184
f_{RAD2}	10	189	0	36	105345
f_{FS1}	2	742	0	45	119504
f_{CCM1}	40	1219	1	60	120270
f_{VB2}	1	973	0	40	119786
f_{BJ1}	47	1101	0	50	120039

Algorithm 2 Sequential DFFs-based *getLowerBound* function.

```
   Function: $getLowerBound(c, W, k, X) \to lb$
1  $lb \leftarrow 0$
2  for $R \in \{R_0, R_{Min}, R_{Max}\}$ do
3      $(c_R, W_R) \leftarrow R(c, W, X)$
4      for $f \in \{f_{CCM1}, f_{MT}, f_{BJ1}, f_{VB2}, f_{FS1}, f_{RAD2}\}$ do
5          $L_f \leftarrow 0$
6          $(\underline{\lambda}, \overline{\lambda}, \delta) \leftarrow getParametersMinMaxStep(f, c_R, 256)$
7          for $\lambda \leftarrow \underline{\lambda}$ to $\overline{\lambda}$ by $\delta$ do
8              $sum \leftarrow \sum_{w_R \in W_R} f(w_R, \lambda)$
9              $L_f \leftarrow max(L_f, \lceil \frac{sum}{f(c_R, \lambda)} \rceil)$
10         $lb \leftarrow max(lb, L_f)$
11         if $lb > k$ then return $lb$          // Infeasibility detected, early return
12 return $lb$
```

when stronger functions are suboptimal [31]. Since no DFF family dominates, it is apparent that restricting ourselves to *choosing* a single family is not productive. Instead, a *portfolio* of independent DFFs should be computed with parallel resources to deliver stronger pruning at virtually no cost (in term of wall-clock time). Recall that the calculation of a single family of lower bound is still pseudo-polynomial and can be costly for large c values. Ideally, one would consider only a minimal subset of parameters guaranteed to lead to the tightest bound, but this is only possible for L_{MT} [16]. In practice, for the CPU implementations, we consider a sampling of 256 equispaced λ values for each family as it proved empirically adequate for obtaining effective bounds.

Similar design considerations were done about the reduction(s) to employ. The analysis in [14] suggest using both R_{Min} and R_{Max}. However, preliminary experiments showed that R_0 is beneficial in some instances, so we considered all of them.

4.1 Sequential CPU Implementation

A sequential DFFs-based implementation of the function *getLowerBound* (see Algorithm 1) is listed in Algorithm 2. It has a nested loop structure where the loop at line 2 consider the three reductions presented in Section 3.1, the loop at line 4 consider the six DFFs in the portfolio, and the loop at line 7 samples the rage of parameters. That results in computation that sequentially calculates $3 * 6 * 256 = 4608$ lower bounds.

4.2 Parallel CPU Implementation

The nested loop structure of Algorithm 2 is easily parallelizable since all iterations are independent. The only data that need to be atomically updated is the maximum lower bound at line 10.

The outermost 2 loops execute the main body of the function (lines 5–10) 18 times (i.e., 3 reductions and 6 DFFs). To easily run on commodity CPU with about 10 cores, it is appropriate to use one thread per DFF to executes the main body sequentially for all 3 reductions. This approach uses 6 threads, each calculating $3 * 256 = 768$ lower bounds. It provides a *sublinear* speedup of 2x when compared to the fully sequential implementation (see Section 5). While it is possible to also parallelize all 3 reductions on a machine with at least 18 cores, it did not seem to be a promising avenue.

The parallel implementation is obtained using OpenMP [29], a C/C++ API that enable transparent multi-threading by simply adding annotations, or *directives*, to the loops. We use the `omp parallel for num_threads(6)` to parallelize the DFF loop, and the directive `reduction(max:lowerbound)` to correctly update the maximum lower bound.

4.3 Considerations for a GPU architecture

To successfully leverage GPUs it is fundamental to understand the weakness and strengths of their architecture. The efficiency of a CPU stems from its *low latency*, which indicates the time required to execute individual operations. Mechanisms such as branch prediction, multiple levels of fast cache, and high clock speeds all contribute to making each of the "few" CPU threads extremely fast. In contrast, the efficiency of a GPU is grounded in its *high throughput*, which represents the number of operations executed per unit of time. The vast number of threads, coupled with rapid context switching, makes the GPU highly effective in performing extensive workloads, compensating for its high latency.

There are various approaches to accelerate propagation algorithms with GPUs. One approach is to parallelize the most prominent algorithm(s). While this seems appealing, it is hard to accomplish for two reasons. First, such algorithms are often designed with a sequential model in mind, making them challenging to parallelize. Data dependencies between iterations as well as the need to synchronize for data structure updates are at the heart of the problem. The second reason is the GPUs high latency, mainly due to the "simple" memory hierarchy where a L1 cache miss results in costly off-chip memory access, as well as the time required to move data and control to and from the GPUs. The optimal point to offload a computation to the GPU changes based on several factors, including hardware characteristics. It is often the case that data transfer negates the benefits of parallelization. This overhead disappears once the GPU workload is large enough. Empirically, it is generally not helpful to offload the propagation of algorithms with a time complexity of $O(n^2)$ or lower.

Another strategy involves utilizing the GPU to reduce the computational cost of strong filtering algorithms [42]. This idea can be applied to the *BinPacking* constraint by employing the GPU to perform a *complete knapsack reasoning* instead of an approximated one. Using the Dynamic Programming (DP) approach presented in [43] it is possible to obtain a stronger filtering that replace all the basic and knapsack filtering in Algorithm 1. We developed a GPU-accelerated implementation of this *pseudo-polynomial* method, leveraging bitwise operations and processing each bin in parallel. Empirical results revealed no significant gains in terms of explored nodes (within the time limits) compared to the approximated reasoning. Scalability tests further indicate that the GPU-accelerated implementation becomes faster than an optimized implementation of the approximated filtering when the number of bins is in the order of *hundreds*. This evidence indicates that theoretically interesting implementations may encounter overheads that outweigh the computational benefits. Ultimately, the disappointing results pushed this second strategy aside.

GPUs can also enhance *pruning*. In the case of the *BinPacking* constraints, this translates into improving the feasibility check to obtain the best possible lower bounds at a reduced computational cost. The tightest available lower bound is derived from the linear relaxation of the Arc-Flow model (see Section 3), which involves solving a sparse linear system. Since this task is notoriously challenging to effectively accelerate with GPUs [20], we explored the next option: considering *all the parameters* and all the DFF families.

Algorithm 3 GPU-accelerated DFFs-based *getLowerBound* function.

```
Function: $getLowerBound(c, W, k, X) \to lb$
$[I_{R_0}, I_{R_{Min}}, I_{R_{Max}}] \leftarrow calcReductions([R_0, R_{Min}, R_{Max}], (c, W, X))$
$lb \leftarrow 0$
$cudaMemcpyCpuToGpu([lb, I_{R_0}, I_{R_{Min}}, I_{R_{Max}}])$          // Asynchronous API
for $(c_R, W_R) \in \{I_{R_0}, I_{R_{Min}}, I_{R_{Max}}\}$ do
    for $f \in \{f_{CCM1}, f_{MT}, f_{BJ1}, f_{VB2}, f_{FS1}, f_{RAD}\}$ do
        $(\underline{\lambda}, \overline{\lambda}) \leftarrow getParametersMinMax(f, c_R)$
        $nThreads \leftarrow \overline{\lambda} - \underline{\lambda} + 1$
        $cudaLaunchKernel(calcDffLowerBound, nThreads, [f, c_R, W_R, \ldots])$          // Async API
$cudaMemcpyGpuToCpu(lb)$          // Asynchronous API
$waitGpu()$          // Synchronous API
return $lb$
```

4.4 GPU Implementation

We handled each combination reduction-DFF with a separate kernel, and each of the $\overline{\lambda} - \underline{\lambda} + 1$ parameter with a different thread (see Figure 5). The GPU-accelerated implementation of the method *getLowerBound* is outlined in Algorithm 3. The first operation copies the reduced instances and the initial lower bound into the GPU's global memory. The amount of transferred data is minimal, and encoded as an array of integers. After that, 18 kernels are launched, each with the appropriate number of threads and several arguments, including the DFF and reduction that they must consider. Finally, the highest lower bound is copied back from the GPU and returned.

The heart of the parallelization is the kernel *calcDffLowerBound*, listed in Algorithm 4. The line 2 shows how each thread uses its index to identify the parameter it works on. The barrier at line 4 ensures the initialization of L_f, and prevents race conditions on its value. Lines 5–6 calculates the value of $L_f(c_R, W_R)$. Finally, the barrier at line 7 guarantees that all parameters are considered before updating the best lower bound.

The pseudocode abstracts out some implementation details that are worth mentioning. From Section 2.2, we recall that the threads of a kernel are organized into *blocks*, each executing in a Streaming Multiprocessor with its own on-chip *shared memory*. This fast memory reduces accesses to the slower global memory in two ways. First, it caches c_R and W_R, ensuring fast access for subsequent lower bound calculations. Second, it maintains L_f enabling faster atomic *max* operations (line 6) that run concurrently between blocks. However, the final atomic *max* operation (line 8) must be performed on global memory, as it is the only means of communication among blocks and kernels.

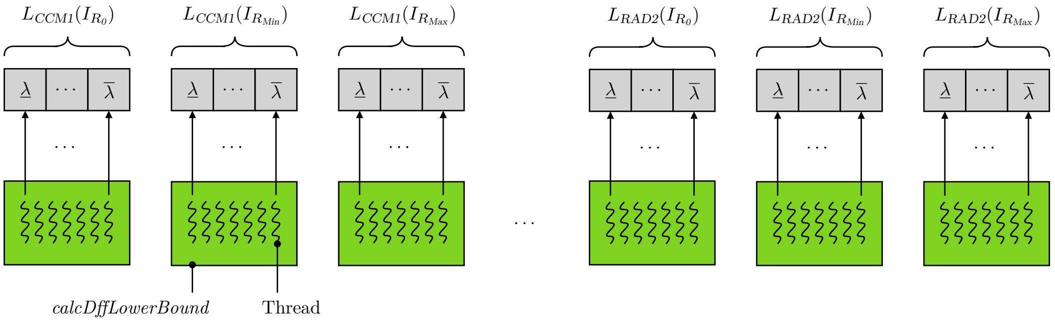


Figure 5 Parallelism of the GPU accelerated *getLowerBound*.

Algorithm 4 Pseudocode of the *calcDffLowerBound* kernel.

```
Procedure: $calcDffLowerBound(f, c_R, W_R, k, lb, \underline{\lambda}, \overline{\lambda})$
if $lb \leq k$ then
    $\lambda \leftarrow \underline{\lambda} + getThreadIdx()$
    $L_f \leftarrow 0$                                                   // Only one thread
    $threadsBarrier()$
    $sum \leftarrow \sum_{w_R \in W_R} f(w_R, \lambda)$
    $L_f \leftarrow max(L_f, \lceil \frac{sum}{f(c_R, \lambda)} \rceil)$                // Atomic operation
    $threadsBarrier()$
    $lb \leftarrow max(lb, L_f)$                      // Only one thread, atomic operation
```

4.5 Solver integration

There are no limitations that prevent the GPU-accelerated *getLowerBound* to be used in the *BinPacking* propagator of a standard CP solver. However, there are a couple of aspects that facilitates such task. Unsurprisingly, it is easiest to integrate in solvers written in C/C++ since CUDA is a C/C++ API, and no wrappers or bindings are needed. Moreover, kernels can be compiled with(in) the solver, without the need to compile them separately and load them at runtime. From the usability prospective, it would be convenient that the solver is compatible with the high-level constraint modelling language *MiniZinc* [27]. By using its annotation mechanism, it is possible to communicate to the solver which implementation of *getLowerBound* to use. For example, when a *BinPacking* constraint is added, it can be annotated with `::parallel` to use the CPU parallel version, or with `::gpu` to use the GPU-accelerated implementation.

We implemented the different versions of *getLowerBound*, along with the relative annotations, within a solver compatible with MiniZinc [39]. Such solver is based on MiniCPP [18], a C++ implementation of MiniCP [26]. We choose MiniCP(P) because it is open-source, well documented, and reasonably simple to modify.

5 Experiments

This section presents a comparison between propagators that use different lower bounds for the feasibility check. We evaluate our linear time complexity implementation of L_2 (i.e, `L2`), our sequential (i.e., `DFFs-CPU-Seq`), parallel (i.e., `DFFs-CPU-Par`), and GPU (i.e., `DFFs-GPU`) DFFs-based implementations, and the implementation from [3] which uses the Arc-Flow based lower bound (i.e., `Arc-Flow`). We select two BPP benchmarks from the literature [15, 37], and generate new instances similar to the ones proposed in [5] and [4]. This results in a total of 2072 instances [40] organized as follows:

Falkenauer This benchmark has two classes of 80 instances each. The ‘U’ instances have items with weights uniformly distributed in $[20, 100]$, $n \in \{120, 250, 500, 1000\}$ and $c = 150$. The ‘T’ instances are characterized by triplets of items that must be packed in the same bin in any optimal solution. For this class $n \in \{60,\ 120,\ 249,\ 501\}$ and $c = 1000$.

Scholl These instances are divided into three sets of 720, 480, and 10 instances. The instances in Set 1 have weights uniformly distributed to expect a number of items per bin not larger than three, $n \in \{50,\ 100,\ 200,\ 500\}$, $c \in \{100, 120, 150\}$. For the instances in Set 2 the number of expected items per bin is between three and nine items, $n \in \{50, 100, 200, 500\}$, $c = 1000$. Set 3 contains big instances with weights uniformly distributed in the range $[20000, 35000]$, $n = 200$ and $c = 100000$.

Weibull These instances are based on the Weibull probability distribution. It can model various distributions found in different problem domains by adjusting the shape parameter $k > 0$ and the scale parameter $\lambda > 0$. Similarly to [5], we generated 92 sets of weights W with the parameters $n \in \{100, 200\}$, $k \in \{0.5, 0.6, \dots, 5.0\}$, and $\lambda = 1000$. For each set W, we generate 6 instances (c, W) with $c = \sigma \cdot \max(W)$ for $\sigma \in \{1.0, 1.2, \dots, 2.0\}$. The total number of instances is 552, with capacity ranging between 1300 and 92500.

Scaled Non-IRUP These instances are derived from instances which *do not* satisfy the *Integer Round-Up Property (IRUP)*. Intuitively, an instance is IRUP if the roundup value of the (strongest) linear relaxation yields to the optimal number of bins. We considered 50 of the instances in [4]. For each instance (c, W) and $s \in \{3, 4, 5\}$, we derived (c_s, W_s) such that $c_s = s * c$ and W_s is the list containing s times the set $\{s * w \mid w \in W\}$. The total number of instances is 150, with $n \in \{45, 60, 75\}$ and c in the range $[921, 5240]$.

The model and search heuristic are the same as in previous works [38, 3], where a minimum number of bins is established and an attempt to find a solution is made. If such a solution does not exist, the number of bins is increased, and a new attempt is made. All implementations use the *decreasing best fit* search heuristic. In this strategy, the items are considered in descending order of weight and assigned to the first bin within their domain that has the smallest residual capacity sufficient to accommodate the item. Additionally, two symmetry-breaking rules are applied on backtracking: first, the bin is removed from the domain of all items of the same size, and second, all the bins with the same load are removed from the domains of these items. Finally, a dominance rule is applied before a choice point: if an item completely fills the remaining capacity of a bin, it is assigned to that bin.

The implementations `L2`, `DFFs-CPU-Seq`, `DFFs-CPU-Par`, and `DFFs-GPU` include a couple of additional techniques. First, another dominance rule is applied before a choice point: if among the set of candidate items that can be packed in a bin, only one can be packed, then the heaviest item is assigned to the bin [34]. Second, the symmetry breaking described in [33] is enforced with an additional constraint. Cardinality reasoning was considered but set aside in preliminary experiments, as it did not yield notable differences in terms of explored nodes while adding some overhead. This can be attributed to the combined effects of strong pruning and the absence of cardinality constraints in our benchmarks

The experiments are performed with 10 minutes timeout to ensure a reasonable benchmark time. The test system features an Intel Core i7-10700K (8 Cores), 32 GB of RAM, and an NVIDIA GeForce RTX 3080 (8704 CUDA Cores). The system operates on Ubuntu Linux 22.04 LTS and uses CUDA 11.8 and GCC 11.4 for our implementations, along with OpenJDK 11.0 and CPLEX 22.1 for `Arc-Flow`.

Results and Analysis

The analysis focuses on instances solved within the 10 minutes time limit. Table 2 reports, for each approach and benchmark, the number of solved instances, the average time per instance, the total solving time, and the total number of visited nodes. Instances that time out are not contributing anything to the total time, average time or nodes column.

Global Analysis. Falkenauer T instances highlight the contrast between fast and slow pruning. `DFF-GPU` quickly solved 73% of the instances, while `Arc-Flow` solved 85% of them taking, on average, 3x more time. The other DFFs-based approaches fall in the middle, and `L2` is last.

Table 2 Statistics for the solved instances of different lower bound methods.

Benchmark (Instances)	Lower Bound	Solved	Avg Time [s]	Time [s]	Nodes
Falkenauer T (80)	`L2`	38	19	733	623305
	`DFFs-CPU-Seq`	46	44	2045	105856
	`DFFs-CPU-Par`	47	28	1315	154440
	`DFFs-GPU`	58	11	650	448780
	`Arc-Flow`	**68**	31	2120	5235
Falkenauer U (80)	`L2`	30	17	496	481984
	`DFFs-CPU-Seq`	56	43	2382	106646
	`DFFs-CPU-Par`	57	39	2198	122521
	`DFFs-GPU`	60	31	1888	357108
	`Arc-Flow`	**79**	16	1303	16012
Scholl 1 (720)	`L2`	637	6	4057	7126695
	`DFFs-CPU-Seq`	696	6	3961	593028
	`DFFs-CPU-Par`	698	5	3398	1361855
	`DFFs-GPU`	703	3	1952	3997093
	`Arc-Flow`	**717**	6	4097	116135
Scholl 2 (480)	`L2`	332	2	771	2777677
	`DFFs-CPU-Seq`	391	8	3035	273011
	`DFFs-CPU-Par`	391	4	1421	273011
	`DFFs-GPU`	**440**	2	827	1235237
	`Arc-Flow`	423	69	29287	278014
Scholl 3 (10)	`L2`	–	–	–	–
	`DFFs-CPU-Seq`	–	–	–	–
	`DFFs-CPU-Par`	–	–	–	–
	`DFFs-GPU`	**3**	1	4	4322
	`Arc-Flow`	–	–	–	–
Weibull (552)	`L2`	371	6	2350	13082358
	`DFFs-CPU-Seq`	395	6	2381	342116
	`DFFs-CPU-Par`	397	4	1782	669801
	`DFFs-GPU`	**417**	6	2636	18149205
	`Arc-Flow`	286	105	30046	11103
Scaled Non-IRUP (150)	`L2`	82	52	4303	63979492
	`DFFs-CPU-Seq`	82	71	5836	4214685
	`DFFs-CPU-Par`	90	61	5520	8329009
	`DFFs-GPU`	**116**	52	6071	41873760
	`Arc-Flow`	108	7	1866	6388

In the Falkenauer U instances, `Arc-Flow` demonstrates a good balance between speed and strength, solving almost all instances in a short amount of time. The DFFs-based approaches have similar performance, suggesting that the computation of lower bounds is negligible. This happens when failures occur earlier in the propagation, during the knapsack reasoning.

In the Scholl 1 instances, the gap between `Arc-Flow` and the DFF-based approaches diminishes notably. `DFFs-GPU` outpaces the CPU approaches by a factor of 2x and 1.7x on average. Notably, while achieving tighter bounds, `DFFs-GPU` explores, on average, 14x and 5x more nodes per second compared to the CPU implementations. It has the lowest runtime per instance and completes 703 instances in half the time of all other contenders.

On Scholl 2, `DFFs-GPU` clearly dominates the field. It solves the most instances (440 out of 480), completes 35 times faster than the second best (Arc-Flow) and clearly improves on its parallel and sequential brethren (2x to 4x faster).

Scholl 3 instances are characterized by huge capacities and highlight the benefits of the GPU approach. It was the only method able to solve any instance leveraging tighter bounds than `L2`, `DFFs-CPU-Seq`, and `DFFs-CPU-Par`, while also being faster than `Arc-Flow`.

The Weibull instances, whose capacities range from medium to large, favor faster computation over strong pruning. In terms of instances solved, `DFFs-GPU` comes first, followed by the other DFFs-based approaches, then `L2`, and `Arc-Flow` last. While `DFFs-CPU-Par` ekes out a win on time per instance, it solves 10% fewer instances than its GPU version. Such instances account for the higher `DFFs-GPU` average solving time. Considering the exploration speed, `DFFs-GPU` visited, on average 48x more nodes than the sequential version and 18x more nodes than the multi-core version.

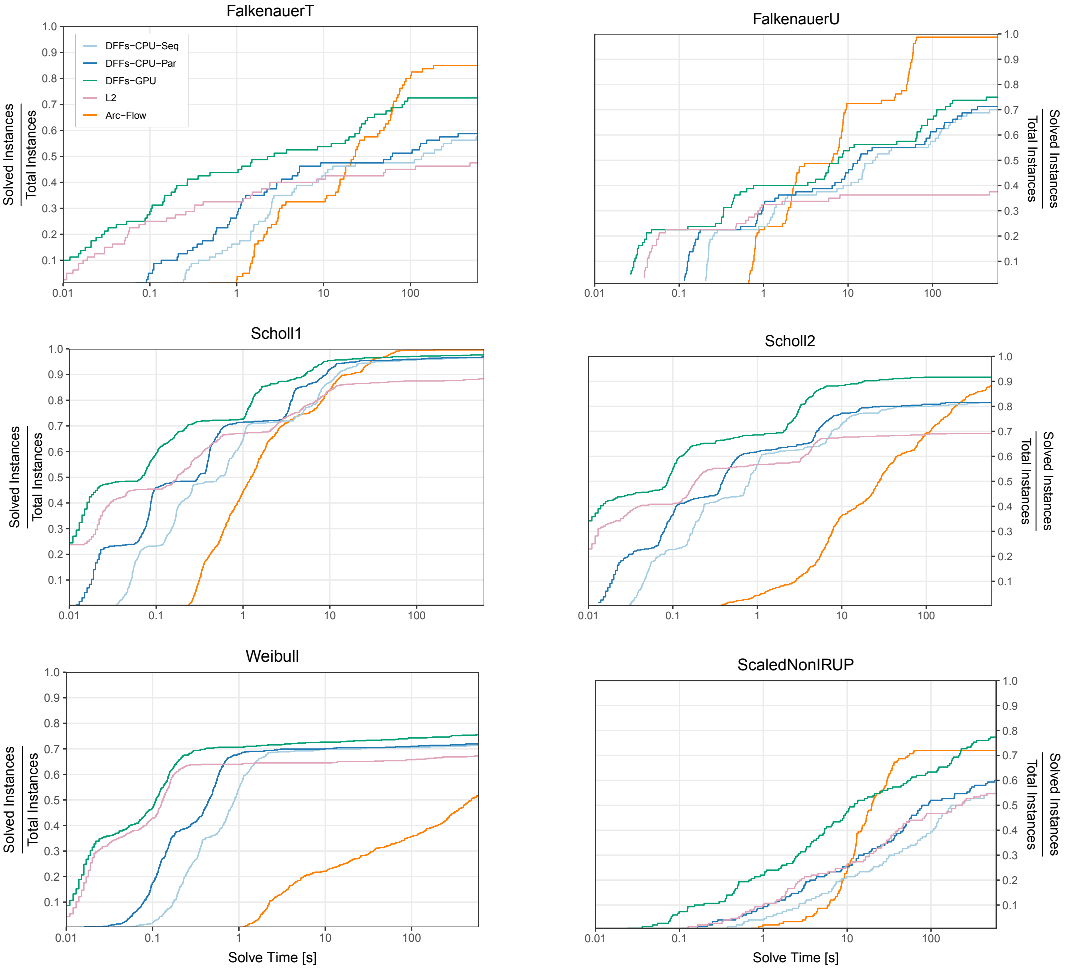


Figure 6 Plots of the empirical cumulative distribution for the benchmarks.

The Scaled Non-IRUP instances *stress-tests* the lower bound capabilities of solvers. While `DFFs-GPU` solves the most instance and is followed by `Arc-Flow`, observe that the hardness is not a function of the number of items. Indeed, `Arc-Flow` times-out on small instance with 45 items, but takes the crown on instances with 60 or 75 items. The remaining contenders are much weaker as an additional 22% of the instance solved by `DFFs-GPU` remain out of reach for `L2` and the sequential DFF implementation, further highlighting the value of a GPU.

Cumulative Analysis. Instance hardness in each benchmark suite is far from uniform. All methods can quickly solve some instances, yet they sharply diverge on others. Cumulative plots for six benchmark classes appear in Figure 6 (School 3 is omitted as `DFF-GPUs` alone could solve instances). The logarithmic horizontal axis is the solving time, while the vertical axis indicates the percentage of instances solved in that time. The `DFFs-GPU` is the green curve and it is readily apparent that it is the north-most, left-most curve in the plots. Indeed, it generally solves more instances significantly faster. The Scaled Non-IRUP instances exhibit an interesting behavior where `DFFs-GPU` and `Arc-Flow` switch roles *twice* as the most effective technique. `DFF-GPUs` is the top-most curve for most values along the x axis.

Table 3 Statistics for `DFFs-GPU` without optimizations.

Version	Solved	Time [s]	Nodes
`DFFs-GPU`	1571	1359	1582808
`DFFs-GPU-NoDom`	1544	2457	83559849
`DFFs-GPU-NoSymBrk`	1526	3970	143491180

Ablation Analysis. An ablation study was conducted on instances solved by `DFFs-GPU` in less than 60 seconds (see Table 3). The most effective technique is the symmetry breaking constraint derived from [33], which is quite general as it applies to variations of the BPP.

6 Conclusions and Future works

This paper revisits the *BinPacking* constraint from a parallel prospective and demonstrates how a parallel mindset leads to novel approaches. It presents a feasibility check based on a portfolio of lower bounds derived from Dual Feasible Functions (DFFs). Sequential, multi-threaded, and GPU-accelerated implementations are described and compared.

The results highlight the role of GPUs and how to achieve an effective balance between computational cost and pruning strength. It allows to handle large instances or situations where it is not practical to spend excessive time at nodes of the search tree. From an analytical standpoint, it would be interesting to identify DFFs that lead to tight bounds in cases where the current ones fall short. Practically, a valuable extension is to explore the effectiveness of multidimensional DFFs [2] on 2D, 3D and Vector Packing Problems.

References

1 Cláudio Alves, Francois Clautiaux, José Valério de Carvalho, and Jürgen Rietz. *Dual-feasible functions for integer programming and combinatorial optimization*. EURO Advanced Tutorials on Operational Research. Springer International Publishing, Basel, Switzerland, 2016.

2 Cláudio Alves, José M. Valério de Carvalho, François Clautiaux, and Jürgen Rietz. Multidimensional dual-feasible functions and fast lower bounds for the vector packing problem. *Eur. J. Oper. Res.*, 233(1):43–63, 2014. `doi:10.1016/J.EJOR.2013.08.011`.

3 Hadrien Cambazard and Barry O'Sullivan. Propagating the bin packing constraint using linear programming. 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 129–136. Springer, 2010. `doi:10.1007/978-3-642-15396-9_13`.

4 Alberto Caprara, Mauro Dell'Amico, José Carlos Díaz Díaz, Manuel Iori, and Romeo Rizzi. Friendly bin packing instances without integer round-up property. *Math. Program.*, 150(1):5–17, 2015. `doi:10.1007/S10107-014-0791-Z`.

5 Ignacio Castiñeiras, Milan De Cauwer, and Barry O'Sullivan. Weibull-based benchmarks for bin packing. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2012. `doi:10.1007/978-3-642-33558-7_17`.

6 François Clautiaux, Cláudio Alves, and José M. Valério de Carvalho. A survey of dual-feasible and superadditive functions. *Ann. Oper. Res.*, 179(1):317–342, 2010. `doi:10.1007/S10479-008-0453-8`.

7 Michele Collevati, Agostino Dovier, and Andrea Formisano. GPU parallelism for SAT solving heuristics. In Roberta Calegari, Giovanni Ciatto, and Andrea Omicini, editors, *Proceedings of the CILC'22*, volume 3204 of *CEUR Workshop Proceedings*, pages 17–31. CEUR-WS.org, 2022.

8 Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. CUD@SAT: SAT solving on GPUs. *J. Exp. Theor. Artif. Intell.*, 27(3):293–316, 2015. doi:10.1080/0952813X.2014.954274.

9 José M. Valério de Carvalho. Exact solution of bin-packing problems using column generation and branch-and-bound. *Ann. Oper. Res.*, 86:629–659, 1999. doi:10.1023/A\%3A1018952112615.

10 Maxence Delorme, Manuel Iori, and Silvano Martello. Bin packing and cutting stock problems: Mathematical models and exact algorithms. *European Journal of Operational Research*, 255(1):1–20, November 2016. doi:10.1016/j.ejor.2016.04.030.

11 Guillaume Derval, Jean-Charles Régin, and Pierre Schaus. Improved filtering for the bin-packing with cardinality constraint. *Constraints An Int. J.*, 23(3):251–271, 2018. doi:10.1007/S10601-017-9278-X.

12 Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Parallel answer set programming. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 237–282. Springer, 2018. doi:10.1007/978-3-319-63516-3_7.

13 Agostino Dovier, Andrea Formisano, and Flavio Vella. GPU-Based Parallelism for ASP-Solving. In Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen, and Dietmar Seipel, editors, *Declarative Programming and Knowledge Management*, volume 12057 of *Lecture Notes in Computer Science*, pages 3–23. Springer, 2019. doi:10.1007/978-3-030-46714-2_1.

14 Julien Dupuis, Pierre Schaus, and Yves Deville. Consistency check for the bin packing constraint revisited. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 117–122. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-13520-0_15.

15 Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing. *J. Heuristics*, 2(1):5–30, 1996. doi:10.1007/BF00226291.

16 Sándor P. Fekete and Jörg Schepers. New classes of fast lower bounds for bin packing problems. *Math. Program.*, 91(1):11–31, 2001. doi:10.1007/S101070100243.

17 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

18 Rebecca Gentzel, Laurent Michel, and W.-J. van Hoeve. HADDOCK: A language and architecture for decision diagram compilation. In *Lecture Notes in Computer Science*, pages 531–547. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-58475-7_31.

19 Michel X. Goemans and Thomas Rothvoss. Polynomiality for bin packing with a constant number of item types. *J. ACM*, 67(6):38:1–38:21, 2020. doi:10.1145/3421750.

20 Wen-Mei W Hwu, David B Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022. URL: https://shop.elsevier.com/books/programming-massively-parallel-processors/hwu/978-0-323-91231-0.

21 Klaus Jansen, Stefan Kratsch, Dániel Marx, and Ildikó Schlotter. Bin packing with fixed number of bins revisited. *J. Comput. Syst. Sci.*, 79(1):39–49, 2013. doi:10.1016/j.jcss.2012.04.004.

22 Richard E. Korf. A new algorithm for optimal bin packing. In Rina Dechter and Michael J. Kearns andRichard S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 731–736. AAAI Press / The MIT Press, 2002. URL: http://www.aaai.org/Library/AAAI/2002/aaai02-110.php.

23 Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977. doi:10.1016/0004-3702(77)90007-8.

24 Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., 1990.

25 Silvano Martello and Paolo Toth. Lower bounds and reduction procedures for the bin packing problem. *Discret. Appl. Math.*, 28:59–70, 1990. doi:10.1016/0166-218X(90)90094-S.

26 L. Michel, P. Schaus, and P. Van Hentenryck. MiniCP: a lightweight solver for constraint programming. *Mathematical Programming Computation*, pages 133–184, 2021. doi:10.1007/s12532-020-00190-7.

27 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

28 Nvidia Team. CUDA. URL: https://developer.nvidia.com/cuda-toolkit.

29 OpenMP Team. OpenMP. URL: https://www.openmp.org/.

30 François Pelsser, Pierre Schaus, and Jean-Charles Régin. Revisiting the cardinality reasoning for binpacking constraint. 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 578–586. Springer, 2013. doi:10.1007/978-3-642-40627-0_43.

31 Jürgen Rietz, Cláudio Alves, and José M. Valério de Carvalho. Theoretical investigations on maximal dual feasible functions. *Oper. Res. Lett.*, 38(3):174–178, 2010. doi:10.1016/J.ORL.2010.01.002.

32 Jürgen Rietz, Cláudio Alves, and José M. Valério de Carvalho. Worst-case analysis of maximal dual feasible functions. *Optim. Lett.*, 6(8):1687–1705, 2012. doi:10.1007/S11590-011-0359-2.

33 Khadija Hadj Salem and Yann Kieffer. An experimental study on symmetry breaking constraints impact for the one dimensional bin-packing problem. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the 2020 Federated Conference on Computer Science and Information Systems, FedCSIS 2020, Sofia, Bulgaria, September 6-9, 2020*, volume 21 of *Annals of Computer Science and Information Systems*, pages 317–326, 2020. doi:10.15439/2020F19.

34 Pierre Schaus. *Solving Balancing and Bin-Packing problems with Constraint Programming*. PhD thesis, University of Louvain, 2009. URL: http://cp2013.a4cp.org/sites/default/files/pierre_schaus_-_mr.pdf.

35 Pierre Schaus, Jean-Charles Régin, Rowan Van Schaeren, Wout Dullaert, and Birger Raa. Cardinality reasoning for bin-packing constraint: Application to a tank allocation problem. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 815–822. Springer, 2012. doi:10.1007/978-3-642-33558-7_58.

36 Guntram Scheithauer. *Introduction to Cutting and Packing Optimization*. Springer International Publishing, 2018. doi:10.1007/978-3-319-64403-5.

37 Armin Scholl, Robert Klein, and Christian Jürgens. Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Comput. Oper. Res.*, 24(7):627–645, 1997. doi:10.1016/S0305-0548(96)00082-2.

38 Paul Shaw. A constraint for bin packing. In *Principles and Practice of Constraint Programming – CP 2004*, pages 648–662. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-30201-8_47.

39 Fabio Tardivo. Fzn-MiniCPP. URL: https://bitbucket.org/constraint-programming/fzn-minicpp.

40 Fabio Tardivo. MiniCPP-Benchmarks. URL: https://bitbucket.org/constraint-programming/minicpp-benchmarks.

41 Fabio Tardivo, Agostino Dovier, Andrea Formisano, Laurent Michel, and Enrico Pontelli. Constraint propagation on GPU: A case study for the AllDifferent constraint. *Journal of Logic and Computation*, page exad033, June 2023. doi:10.1093/logcom/exad033.

42 Fabio Tardivo, Agostino Dovier, Andrea Formisano, Laurent Michel, and Enrico Pontelli. Constraint propagation on GPU: A case study for the cumulative constraint. 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 336–353. Springer, 2023. doi:10.1007/978-3-031-33271-5_22.

43 Michael A. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Ann. Oper. Res.*, 118(1-4):73–84, 2003. doi:10.1023/A:1021801522545.

Mutational Fuzz Testing for Constraint Modeling Systems

Wout Vanroose
DTAI, KU Leuven, Belgium

Ignace Bleukx
DTAI, KU Leuven, Belgium

Jo Devriendt
DTAI, KU Leuven, Belgium

Dimos Tsouros
DTAI, KU Leuven, Belgium

Hélène Verhaeghe
DTAI, KU Leuven, Belgium

Tias Guns
DTAI, KU Leuven, Belgium

Abstract

Constraint programming (CP) modeling languages, like MiniZinc, Essence and CPMpy, play a crucial role in making CP technology accessible to non-experts. Both solver-independent modeling frameworks and solvers themselves are complex pieces of software that can contain bugs, which undermines their usefulness. Mutational fuzz testing is a way to test complex systems by stochastically mutating input and verifying preserved properties of the mutated output. We investigate different mutations and verification methods that can be used on the constraint specifications directly. This includes methods proposed in the context of SMT problem specifications, as well as new methods related to global constraints, optimization, and solution counting/preservation. Our results show that such a fuzz testing approach improves the overall code coverage of a modeling system compared to only unit testing, and is able to find bugs in the whole toolchain, from the modeling language transformations themselves to the underlying solvers.

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

Keywords and phrases fuzz testing, Constraint modeling language, bugs, mutational testing, modeling, constraint reformulation

Digital Object Identifier 10.4230/LIPIcs.CP.2024.29

Supplementary Material *Software*: `https://github.com/CPMpy/fuzz-test`
archived at `swh:1:dir:2da6f1206cfd0e1e23c66548ea83f455bafc569c`

Funding This research received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Grant No. 101002802, CHAT-Opt).

1 Introduction

Constraint solving is a declarative AI reasoning technique that is used in a variety of high-stakes applications ranging from scheduling production lines [19] to automated verification of computer programs [21] and aerospace applications [34]. All of these applications require constraint solvers to provide correct and reliable solutions to the constraint specifications.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 29; pp. 29:1–29:25
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

To leverage the power of modern constraint solvers, it is common for users to write down the problem specification in a high level, declarative *constraint modeling language* such as MiniZinc [26], XCSP [33], Essence [2] or CPMpy [17]. These modeling languages play a fundamental role in enabling the wider adoption of CP technology across various domains as they provide high-level, expressive, and intuitive methods for users to define complex problem constraints. They offer an abstraction from the details of encoding high-level constraints into the specific constraints supported by a solver, allowing users to focus on the problem at hand rather than the specifics of the solvers.

Modeling systems then reformulate the high-level user-constraints into solver-specific expressions such as clauses, linear constraints or unnested global constraints. For this, the code base of modeling systems typically contains multiple reformulation and encoding algorithms.

Modeling systems are also made more complex by optimizations such as Common Subexpression Elimination (CSE) [27, 28, 30], used to reduce the number of generated low-level constraints. In some cases, these transformations are mixed-and-matched in different ways for different solvers.

Like all complex software, modeling systems and constraint solvers can contain bugs. In the case of modeling systems, bugs can cause a range of undesired behavior: from experiencing crashes of the system itself to returning an invalid or non-optimal solution to the constraints stated by the user. Especially the latter can have a major impact on the user and the application at hand. Moreover, it can also decrease the trust of users towards the underlying solving techniques.

To mitigate the number of bugs in computer programs, it is good practice to use some kind of *automated testing* during software development. *Unit testing* [13] is such a technique to test isolated parts of the code using small test cases. While unit testing is very useful to verify the intended behavior of a program, it is time-consuming for developers to write as it necessitates testing for both expected and unexpected inputs. Therefore, tricky edge cases may be overlooked when designing the test suite. In constraint solving, this is especially the case for non-trivial combinations of constraints that share variables.

Fuzz testing is a family of techniques that automatically test computer programs on randomly constructed inputs. These techniques can either be *generation-based* or *mutation-based*: the former generates input from scratch, while the latter uses existing inputs and applies mutations to them in order to construct a valid new input. Fuzz testing has proved to be extremely successful in finding bugs in a variety of computer programs: from testing Android apps [43], to crashes of Unix command-line utilities [25], and SMT solvers [23, 42]. Although fuzz testing has been used to test several solver-specific algorithms such as propagation routines [3, 10, 23, 29, 42], it has not yet been applied to solver-independent constraint modeling languages, despite their rapid development in recent years.

In this paper, we draw inspiration from systems such as STORM [23] and YinYang [42] tailored to test SMT solvers, and propose HURRICANE, a method to use *mutational fuzz testing* for generic constraint modeling systems. The input that will be mutated in this case, are entire CP constraint specifications.

New opportunities for fuzz testing arise, because of the rich constraint specification that CP modeling languages allow. These include the use of global constraints and their decompositions [38], the use of n-ary aggregate functions, the possibility of arbitrarily nested expressions (even global constraints) that may require flattening, the use of objective functions, and the changing transformation flows that are used for different available backend solvers.

Our contributions are the following:

1. We propose a generic, mutation-based, automated testing framework, HURRICANE, for verifying the correctness of solver-independent CP modeling languages and their solvers;
2. We investigate the use of 3 families of mutations; as well as 5 methods to verify the mutated models do not contain bugs; and
3. We evaluate HURRICANE by mutating and testing CP problems modelled in the CPMpy constraint modeling system [17], and show its effectiveness at finding bugs in the system itself as well as its underlying solvers.

2 Related work

Automated testing of computer programs finds its roots in *unit testing* [13]. A unit test consists of a small use case of a part of the software as envisioned by the developers. The technique was made popular by the JUnit testing framework in Java [36].

In recent years, researchers have studied ways to automatically synthesize unit tests in order to improve *code coverage* of the test suite [22]. Code coverage quantifies the number of lines of code in a program that is executed by a (set of) tests. While this is not a foolproof metric [40], it is a reasonable proxy to evaluate how thoroughly a system is tested.

Fuzz testing has been used in combinatorial solving before. An early form of testing SAT-solvers uses generation-based techniques [11], and more recently, several solvers who entered the 2022 edition of the Max-SAT competition were subjected to fuzz testing [29]. In the field of CP, *generation-based fuzz testing* has already been adopted as an automatic testing technique for solvers. For example, the propagation algorithms present in the MINION solver have been automatically fuzz tested throughout its development [3]. The input used for testing such propagation routines is a randomly generated set of constraints within the relatively simple grammar supported by the solver. The output of the solver is verified using simpler, but equivalent algorithms. A hybrid approach between fuzzing and formal specifications for testing CP solvers has also been used by the SolverCheck system [14].

Compared to the API of a constraint solver, CP modeling languages allow for a much richer set of expressions to be written down by a user (e.g., nested constraints). This makes stochastic *generation* of inputs more complex [35], hence we turn our attention to *mutational fuzz testing* techniques that mutate existing constraint specifications. The idea of mutating constraint specifications has previously been explored for satisfiability checking SMT solvers [23, 42, 9]. These techniques can generate deeply nested expressions in the language that SMT solvers natively accept as input. While also applicable to high-level constraint modeling languages, we propose new mutations and verification methods based on the richer input CP modeling languages allow.

Finally, a very different kind of technique to detect bugs in combinatorial solvers is through the use of *proof logging*. Proof logging requires a system to write down the result of its algorithms as relatively simple mathematical reasoning steps. Such proofs are then *verified* automatically by a third-party checker [15, 16, 18]. SAT solvers are required to output proof logs (mathematical search certificates) in order to enter the yearly SAT competition[1]. In recent years, proof logging has successfully found its way to other combinatorial search algorithms such as those used in (Max-)SAT-, ASP-, SMT- and CP [4, 5, 8, 24, 31, 39]. However, proof logging for now remains a low-level technique that is not directly applicable to algorithms that translate any high-level expressions into multiple equivalent low-level solver constraints.

[1] `https://satcompetition.github.io/`

3 Preliminaries

A *Constraint Satisfaction Problem (CSP)* is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ [32] with

- $\mathcal{X}$ a set of *decision variables*;
- $\mathcal{D}$ a set of *domains* of *values* for each variable in $\mathcal{X}$;
- $\mathcal{C}$ a set of *constraints*, each over some subset of $\mathcal{X}$.

An *assignment* maps variables in $\mathcal{X}$ to a value in their domain. A *constraint* maps assignments to true or false. An assignment *satisfies* a constraint if the constraint maps it to true. We make no assumption on the structure of a constraint, that is, it can be a nested expression as we will see below. A *solution* to a CSP is an assignment for all variables in $\mathcal{X}$ that satisfies all constraints in $\mathcal{C}$. The set of solutions of a set of constraints, projected to a set of variables $\mathcal{X}$ is written as $sols_{\mathcal{X}}(\mathcal{C})$. E.g., given the following set of constraints $\mathcal{C} = \{p + q + z \leq 2, p < q\}$ and positive domains for p, q and z, we observe the following sets of solutions:

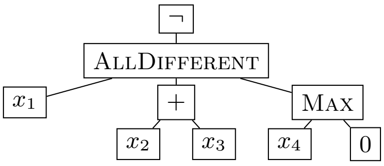
$$\begin{aligned} sols(\mathcal{C}) &= \{\{p \mapsto 0, q \mapsto 1, z \mapsto 0\}, \{p \mapsto 0, q \mapsto 1, z \mapsto 1\}, \{p \mapsto 0, q \mapsto 2, z \mapsto 0\}\} \\ sols_{\{p,q\}}(\mathcal{C}) &= \{\{p \mapsto 0, q \mapsto 1\}, \{p \mapsto 0, q \mapsto 2\}\} \end{aligned}$$

A CSP allowing no solutions is *unsatisfiable*. In CP it is common to use an *objective* function to quantify the quality of a solution. A *Constraint Optimization Problem (COP)* is a quadruple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$ with f a function that maps assignments to a numeric value. An *optimal* solution is a solution to the COP such that no solution exists with a lower/higher objective value for minimization/maximization problems.

It is common to use the term *constraint network* for what we call a CSP. A CSP would then be the problem of finding solutions to the constraint network. In this paper we will use the term CSP for both. In the context of constraint modeling languages, a CSP could be called a *model*, we use these terms interchangeably.

Global constraints are one of the essential features of constraint programming and capture high-level relations between a (non-fixed) number of variables [38]. Well-known examples of global constraints are the AllDifferent [37] constraint or the Cumulative [1] constraint. More examples can be found in the global constraint catalog [6].

Typically, constraints and objectives are represented by expressions in some formal syntax. E.g., the constraint $\neg\textsc{AllDifferent}(x_1, x_2 + x_3, \textsc{Max}(x_4, 0))$ maps those assignments to true where x_1, $x_2 + x_3$, and the maximum of x_4 and zero do not all take different values. Equivalently, constraints can be inductively defined as expression trees. Its leaf nodes are variables or values. Its non-leaf nodes are formed by applying *operators*, *global constraints*, *functions*, and *comparisons* to other expressions. The expression tree representing the previously mentioned complex expression is shown in Figure 1a.



(a) Expression tree.

$$\begin{aligned} &\neg b_1 \\ &b_1 \leftrightarrow \textsc{AllDifferent}(x_1, n_1, n_2) \\ &n_1 = x_1 + x_2 \\ &n_2 = \textsc{max}(x_4, 0) \end{aligned}$$

(b) Flattened version.

Figure 1 Expression tree and flattened version of $\neg\textsc{AllDifferent}(x_1, x_2 + x_3, \textsc{Max}(x_4, 0))$.

3.1 Solvers and modeling systems

CSPs are solved by *constraint solvers*: highly optimized combinatorial search systems that accept a set of constraints and return (optimal) solutions or report that none exist. Constraint solvers do not accept arbitrary expression trees as constraints. Instead, they have a restricted input and rarely a solver would accept a complex expression like the one given in Figure 1a as an input constraint.

Instead of having to manually transform a problem to the format of each solver, a *model and solve* approach is used, where a user specifies the constraints in an expressive, high-level *modeling language*. Then, an underlying compiler translates these constraints to simpler, low-level constraints that are passed to a solver. The translation involves multiple complex *transformation* steps, with *flattening* (unnesting of nested expressions) and *global constraint decomposition* (decomposition of unsupported global constraint) as notable examples [30]. Because different solvers can accept different inputs, distinct transformation paths are necessary for different solvers. When using MIP solvers, the constraints have to be linearised into mixed integer linear inequalities [7], for SAT solvers only propositional clauses should be left, or for CP solvers non-nested constraints over variables, where globals constraints that are not supported are decomposed. Note that such transformations, like flattening, can introduce auxiliary variables, that are not visible to the user but necessary for obtaining an equivalent set of constraints that the solver accepts. When presenting a solution to the user, the solution the solver found has to be projected back to the original variables that the user knows about.

▶ **Example 1** (Flattening). In Figure 1b, we show the flattened version of the expression $\neg\textsc{AllDifferent}(x_1, x_2 + x_3, \textsc{Max}(x_4, 0))$. The flat output is constructed by traversing the expression tree in Figure 1a and introducing auxiliary variables n_1, n_2 and b_1 for every non leaf-node. n_1 and n_2 are numerical variables while b_1 is Boolean. Additional transformations might be needed, depending on the constraints supported by a solver.

3.2 CPMpy

As a concrete modeling system, we will use CPMpy [17], a constraint modeling library embedded in the Python programming language. It translates high-level expressions written by a user, to different constraint solvers using a sequence of generic *transformations*. A list of these internal transformation can be found in Appendix B. Multiple solvers are supported, including CP, SAT, MIP, SMT and Pseudo-Boolean solvers.

CPMpy's input language allows arithmetic operations $(+, -, /, \times \dots)$, comparisons $(= , \neq, <, >, \leq, \geq)$, logical operations $(\neg, \wedge, \vee, \rightarrow, \oplus)$, functions ($\textsc{Max}$, $\textsc{Count}$, $\textsc{Abs}$ ...) and global constraints ($\textsc{AllDifferent}$, $\textsc{Cumulative}$ ...). Expressions in CPMpy are either of Boolean or integer type. With $\mathcal{B}$ we denote the Boolean expressions, with $\mathcal{N}$ the integer ones. Any Boolean expression in CPMpy can also be used as an integer expression (with true treated as 1 and false as 0). In other words, $\mathcal{B} \subseteq \mathcal{N}$.

CPMpy allows users to arbitrarily nest expressions. For example, a disjunction can be used as a *constraint* or as an argument to an operator, a function or even a global constraint. Similarly, global constraints can be arbitrarily nested and used as any Boolean expression. E.g., $\textsc{Max}(10 \cdot \textsc{Circuit}(x_1, x_2, x_3), x_1/x_4) \neq 7$ is a valid CPMpy expression. Therefore, we avoid the use of the word "constraint" to represent a Boolean expression, as such a Boolean expression might be used as a subexpression instead. We use the concept of *top-level expression* to denote that the expression was given to the solver as a constraint.

4 Mutational testing

We now introduce HURRICANE, a framework for *mutational fuzz testing* of constraint modeling systems, inspired by the STORM [23] and YinYang [42] systems for testing SMT-solvers. A high-level overview is shown in Algorithm 1.

Algorithm 1 HURRICANE.

```
Input: set of $m$ CSP models $\{(\mathcal{X}_j, \mathcal{D}_j, \mathcal{C}_j)\}$, set of mutations $\mathcal{M}$, a verification
       method $\mathcal{V}$ and $n$, a number of mutations to apply to each instance
while true do
    $(\mathcal{X}, \mathcal{D}, \mathcal{C}) \leftarrow$ pick an instance from the input set
    for $i = 1 \ldots n$ do
        $M \leftarrow$ pick a mutation from $\mathcal{M}$
        $\mathcal{C} \leftarrow \mathcal{C} \cup M(\mathcal{C})$
    if $\mathcal{V}(\mathcal{C})$ does not succeed then
        yield bug with constraints $\mathcal{C}$
```

Our method takes as input a set of m constraint satisfaction or optimization problems that are known to be *satisfiable*. In each iteration of the algorithm, we randomly pick one of the models and apply a number of *mutations* to its constraints. A mutation is a function M that takes as input a set of constraints and outputs a set of new constraints $M(\mathcal{C})$. We investigate different mutations in Section 5. These newly generated constraints are then *added to the model*. Notice this allows to generate weaker constraints without altering the set of solutions of the model. After applying these mutations, we *verify* whether the resulting set of constraints satisfies certain properties, e.g., whether the mutated model is still satisfiable. Whenever this check fails, the algorithm has found a bug in the system and this is logged to the user. Section 6 discusses the methods that can be used in order to verify the mutated models. When a verification step fails, we know there is a bug *somewhere* in the system. However, because the system consists of different components (internal CPMpy transformations, solver interfaces, backend solvers, the mutations and the verification step), further investigation will be required to identify which part contains the bug.

As our algorithm involves several random components, it is common to (re-)discover the same error or bug with multiple combinations of mutations. In an attempt to minimize this to some extent, we exclude any mutation-model combinations which have already produced a bug, without showing this explicitly in the pseudocode.

Input models

To construct a varied dataset of feasible input models, we extract the constraint models used for the unit tests of the given modeling language. We only use those that have at least one constraint, and at least one solution, since our verification step will rely on this. From a practical point of view, this is useful as unit test models are readily available and kept up-to-date. Many of models used in unit tests also tend to be small and fast to solve. Moreover, unit tests are highly diverse and it is reasonable to assume these models will contain all language constructs (such as global constraints and functions). Finally, additional test cases are often added to the unit tests as part of a bug-fix, hence a fix is tested more rigorously by applying fuzz testing on the newly added test-model too.

Throughout this paper we use the following input model as a running example.

▶ **Example 2** (Running example). Consider the following constraint satisfaction problem with integer variables x, y, z, p and q with domains $[1..5]$ and a Boolean variable b.

$$\textsc{AllDifferent}(x, y, z), \quad y + \textsc{Min}(p, q) > 3, \quad 2 \cdot (x + p) \leq 7$$

5 Mutations

We consider three families of mutations. The first of which is based on the reformulation methods built into constraint modeling systems such as flattening, or linearization of constraints. Second, we focus on *top-level* mutations which combine existing *top-level expressions* to create a new expression, and lastly, we consider *sub-expression-level* mutations which can replace nodes at arbitrary depth in the expression tree. All of these mutations generate constraints which do not disallow any of the solutions of the original constraints. Because we also leave the original constraints in the mutated model (see Algorithm 1), this means the set of solutions projected to the original variables should remain unchanged after any mutation. This property of our mutations is exploited in Section 6 to verify the output of the modeling system after mutating the constraint model.

5.1 Reformulation mutations

Constraint modeling systems implement *reformulation* methods in order to rewrite constraints into semantically equivalent ones. For example, when a modeling system interfaces a MIP solver, it implements some procedure to *linearize* constraints. That is, to rewrite any constraint into weighted sums and linear comparisons. Similary, CP modeling systems *decompose* unsupported global constraints or *flatten* complex expression trees.

CPMpy provides this functionality as standalone *transformation functions* which take as input a set of constraints and output a set of (simpler) constraints that imply the input constraints[2]. As these transformations are supposed to create sets of constraints that leave the solutions of the CSP unaltered, we can directly use each of them as a candidate mutation in the mutational testing framework. By re-using these transformation functions, we are able to test these core components of the modeling language on a wide range of expressions, even if the backend solver does not require that specific transformation. The full list of the transformation functions used and their description can be found in Appendix B.

5.2 Top-level mutations

The first set of mutations we use in our framework is based on logical operations with the main idea being the following: given two Boolean expressions from the *top-level* of the constraint model, combine them to create an implied expression. As both input expressions will be enforced to be satisfied by the constraint solver, the newly generated expressions do not alter the set of solutions when added to the model and can be considered *redundant* from a logical point of view.

We compile a set of top-level mutations as summarized in Section 5.2. They are inspired by the mutations described in [23] and derived from the truth table of the logical operation relation whose name is shown as subscript in the function descriptions below. We repeat

[2] `https://github.com/CPMpy/cpmpy/tree/master/cpmpy/transformations`

that these operations are only done on top-level constraints, so they are all implied under the condition of $a \land b$ being enforced. Hence, all these constraints can be added to the model without changing the set of solutions.

$$M_{neg}(a) = \{a, \neg(\neg a)\} \tag{1a}$$

$$M_{conj}(a,b) = \{(a \land b), \neg(a \land \neg b), \neg(\neg a \land b), \neg(\neg a \land \neg b)\} \tag{1b}$$

$$M_{disj}(a,b) = \{(a \lor b), (a \lor \neg b), (\neg a \lor b), \neg(\neg a \lor \neg b)\} \tag{1c}$$

$$\begin{aligned} M_{impl}(a,b) = \{&(a \rightarrow b), (\neg a \rightarrow b), (b \rightarrow a), (\neg b \rightarrow a), \\ &\neg(a \rightarrow \neg b), (\neg a \rightarrow \neg b), \neg(b \rightarrow \neg a), (\neg b \rightarrow \neg a)\} \end{aligned} \tag{1d}$$

$$M_{xor}(a,b) = \{(a \oplus \neg b), (\neg a \oplus b), \neg(a \oplus b), \neg(\neg a \oplus \neg b)\} \tag{1e}$$

Note that we add all these constraints as is, e.g. we do not simplify $\neg(a \land \neg b)$ to $(\neg a \lor b)$ but leave this expression for future mutations to manipulate further, and for the transformations and solvers to handle correctly.

Our proposed mutation will randomly pick one of the above mutations and add the corresponding sets of implied constraints to the model.

► **Example 3.** Given the constraint model shown in Example 2. Imagine HURRICANE selects the constraints $a \coloneqq \textsc{AllDifferent}(x,y,z)$ and $b \coloneqq \textsc{Min}(p,q) > 3$ and the top-level mutation derived from the disjunction operator, M_{disj}. Then the following set of constraints is generated and added to the model, resulting in a CSP with seven constraints.

$$\begin{aligned} \{&(\textsc{AllDiff}(x,y,z)) \lor (2 \cdot (x+p) \leq 7), &&\neg\big(\neg\textsc{AllDiff}(x,y,z) \lor \neg(2 \cdot (x+p) \leq 7)\big), \\ &(\neg\textsc{AllDiff}(x,y,z)) \lor (2 \cdot (x+p) \leq 7), &&(\textsc{AllDiff}(x,y,z)) \lor \neg(2 \cdot (x+p) \leq 7)\} \end{aligned}$$

5.3 Subexpression mutations

The mutations described in the previous section operate on top-level Boolean expressions. However, we can also modify the expression trees themselves by replacing any of the nodes (e.g. an argument of an expression) with an equivalent one. Such modified expression trees may trigger different code paths, for example during flattening if the modified argument was a variable and is now a nested expression instead.

In order to find a set of subexpressions to use for the mutation, we first recursively traverse the expression tree of each of the top-level constraints. Whenever we find a (sub)expression of the required type – e.g., a numeric subexpression/argument – we add that subexpression to the set of candidates to sample from. Once this set of candidate expressions is found, we sample the required amount of expressions to use in the mutation. In the remainder of this section we discuss two types of subexpression mutations.

Semantic fusion

As a way to combine numeric sub-expressions, *semantic fusion* was introduced in the context of testing SMT-solvers [42]. The key idea is to fuse two expressions and create an auxiliary variable for it, and then replace the original expressions with an equivalent one involving that variable.

In general, semantic fusion requires a *fusion function* $f(a,b)$ which takes as input two numeric expressions; an auxiliary variable v and two *inversion functions* $r_a(v,b)$ and $r_b(v,a)$. We can then mutate constraints in which a and b occur, by replacing the occurrences of a and b by their now equivalent $r_a(v,b)$ and $r_b(v,a)$ expressions.

▶ **Example 4.** We sample two numeric subexpressions from the CSP given in Example 2. For example, we take $a := \textsc{Min}(p, q)$ and $b := 2 \cdot (x + p)$, which are sampled from the second and third constraint in the CSP respectively. Using the fusion function $f(a, b) = a + b$, we now define a new auxiliary variable v to link the new fused expression as $v = \textsc{Min}(p, q) + 2 \cdot (x + p)$. We can now define a relation from a to b and vice versa involving the auxiliary variable. E.g., we replace $\textsc{Min}(p, q)$ with $v - 2 \cdot (x + p)$ and the occurrences of $2 \cdot (x + p)$ with $v - \textsc{Min}(p, q)$.

This yields the two constraints $y + (v - 2 \cdot (x + p)) > 3$ and $v - \textsc{Min}(p, q) \leq 7$ which are then added to the model.

Multiple operations can be used for the fusion function, even Boolean operators (in which case boolean sub-expressions should be selected), though an appropriate inverse function must exist. For example $f(a, b) = a \vee b$ and $f(a, b) = a \wedge b$ do not allow constructing appropriate inversion functions. In practice, we make use of the fusion functions shown in Table 1.

Table 1 Functions which can be used in semantic fusion of arithmetic expressions.

Origin	Fusion Function	Inverse Functions
Sum	$f(a, b) = a + b$	$r_a(v, b) = v - b$ $r_b(v, a) = v - a$
Weigthed sum	$f(a, b) = c_1 \cdot a + c_2 \cdot b + c_3$	$r_a(v, b) = (v - c_2 \cdot b - c_3)/c_1$ $r_b(v, a) = (v - c_1 \cdot a - c_3)/c_2$
Substract	$f(a, b) = a - b$	$r_a(v, b) = v + b$ $r_b(v, a) = a - v$

Equivalent comparisons

The second type of subexpression mutators generates equivalent comparisons. This is done by selecting a random comparison in the expression tree of the constraint model and applying the same operation to both its sides. These operations can either *add* a constant, *subtract* a constant or *apply multiplication* by a constant. The constant itself is picked at random. Although this mutation is based on a straightforward idea, we did not find any mention of it in literature.

▶ **Example 5.** Imagine the algorithm picks the second constraint of the running Example 2: $y + \textsc{Min}(p, q) > 3$ and the *multiply by a constant* mutator. If the constant used is “5”, then applying the mutation results in the expression $5 \cdot (y + \textsc{Min}(p, q)) > 5 \cdot 3$.

Depending on the exact grammar allowed by the modeling language, the same could in principle be done with a fresh variable or even an existing numeric subexpression from another constraint, but in this case we just use an integer constant.

6 Verification methods

To detect whether a bug has occurred, we need to *verify* that certain properties hold for the mutated constraints. In fuzz testing for SMT research [9, 23, 42], the authors check if, after mutations, the model still admits a solution. However, more elaborate checks are possible as well. In particular, as the mutations presented in Section 5 *should* not alter the set of solutions projected to the original variables. The verification methods presented in the following sections are all methods which check whether indeed this set of solutions is preserved. Different trade-offs between efficiency, code coverage, and thoroughness of the verification present themselves. We compare and evaluate them experimentally in Section 9.

6.1 All-solutions

A first method to check whether the set of solutions is unchanged is to enumerate the solutions of the original model and those of the mutated model and checking for equivalence of solution sets. Some of the mutations presented in Section 5 can introduce auxiliary variables. E.g., semantic fusion introduces a *fusion variable* but also the built-in reformulations such as *flattening* can introduce new variables into the model. Therefore, in order to compare both sets of solutions, we need to project them to the original set of decision variables $\mathcal{X}$. I.e., this verification method checks whether the following equivalence holds:

$$sols_{\mathcal{X}}(\mathcal{C}) \equiv sols_{\mathcal{X}}(\mathcal{C} \cup M(\mathcal{C}))$$

Note that enumeration of all solutions is a costly operation – $\#\mathcal{P}$-complete in general [12] – but solvers oftentimes have built-in methods for doing so. CPMpy implements enumeration of all solutions using the `solveAll` function. This in turn calls the built-in enumeration method of the solver if available, otherwise it implements the enumeration using repeated solve calls and blocking clauses. Clearly, using this verification method does not only allow for a theoretically strong verification of the mutations, but can also trigger different code paths in either the modeling system or the solver itself.

6.2 Solution count

Apart from checking whether *projected* sets of solutions are equivalent, we also want to check whether new solutions are introduced by the mutations, with respect to auxiliary variables. E.g., if a mutation introduces an unconstrained Boolean auxiliary variable, the total number of solutions will be doubled. While this behaviour is unwanted for any of the mutations presented in this paper, it is undetected by the **All-solutions** verification method as the sets of solutions are projected to the original variables. This is however not the case when counting the number of solutions without enumerating them, because this count is provided by the back-end solver that operates on the model with auxiliary variables. Therefore, we propose to also check whether the total number of solutions of the mutated model is unchanged to the original number of solutions. I.e., we check whether whether

$$|sols(\mathcal{C})| \equiv |sols(\mathcal{C} \cup M(\mathcal{C}))|$$

Similar to enumeration of all solutions, counting solutions is also a costly operation, but may trigger new code paths in modeling systems or solvers. Note that solution counting and checking equivalence of projected solutions sets are complementary to one another. While solution counting discovers bugs related to auxiliary variables, **All-solutions** can discover bugs related to assigned values of the decision variables.

6.3 1-solution

Instead of checking whether *all* solutions remain for the mutated constraints, we can check whether a predefined solution is preserved by the mutations. Conceptually, we check for a given solution θ whether

$$\theta \in sols(\mathcal{C} \cup M(\mathcal{C}))$$

In practice, we implement this by adding the assignment of a pre-computed solution to the set of mutated constraints and asking the solver if the resulting constraints are satisfiable. E.g., for the CSP from Example 2, we can test if after mutation of the constraints, the assignment $\{b \mapsto \mathit{false}, x \mapsto 2, y \mapsto 3, z \mapsto 1, p \mapsto 2, q \mapsto 1\}$ is still a solution of the CSP.

Notice that checking satisfiability can be extremely fast here, as the solver does not require any search when all variables are fixed! Naturally, finding a solution for the original CSP requires invoking a solver nevertheless.

6.4 Satisfiability

A weaker verification method than checking whether a computed assignment is a solution of the mutated model, is to check whether the mutated model admits a solution at all. This verification method is similar to the work on fuzz testing SMT-solvers [9, 23, 42]. Naturally, this check does not detect subtle changes in the set of solutions of the mutated model, but rather checks if the sets of solutions is non-empty.

6.5 Optimization

In constraint programming, it is common to use an objective function in order to quantify the quality of a solution. E.g., when scheduling a set of tasks on a machine, it is common to find a schedule which runs in the least amount of time or requires the smallest amount of energy. When such an objective function is set in a constraint model, we can check whether solving the mutated model to optimality yields the same objective value.

This verification is conceptually stronger compared to checking the satisfiability of the model, and solving to optimality will trigger different code paths. There are two disadvantages: First, it requires the existence of an objective function in the model and second, finding an optimal solution to a CSP is harder than finding any satisfying solution to the constraints, and hence will take more time compared to checking the satisfiability of the mutated constraints.

7 Dealing with bugs

Computer programs can exhibit several types of bugs. Similar to the authors of [23], we define three classes of bugs that occur in constraint modeling systems. Section 7.1 discusses errors in the logic of modeling systems and solvers, while Section 7.2 and Section 7.3 focus on bugs which impact the runtime environment of modeling systems. Lastly, in Section 7.4, we discuss a practical method to minimise bugged models.

7.1 Soundness bugs

The first type of bug are those where the modeling system returns a wrong answer to a verification check from Section 6. Such bugs are critical as the user is given a wrong answer to the constraints, without any indication that something went wrong, like an error message. E.g., the solver returns a non-optimal solution to an optimization problem or declares a set of constraints to be unsatisfiable when in fact they admit a solution.

Soundness bugs can be caused by either the solver itself, or by the modeling system. For example, when a solver's propagation function for a (global) constraint removes values from a domain which allowed a solution, the root-cause of the bug lies with the solver.

When the bug is caused by the modeling system this could be due to a flawed interface to the solver or an improper reformulation of the constraints.

Overall, soundness bugs are critical but difficult to detect in day-to-day use of a modeling language, as this usage rarely includes verifying the result in a later stage.

7.2 Crashes

During the execution of HURRICANE, it is possible the runtime of the modeling system crashes. We identify two main points of possible failure: applying a mutation and verifying the mutated model.

We noticed crashes or errors occurring during the mutation of set of constraints are often triggered when a reformulation mutation is chosen. For example, during linearization of a set of constraints, an assertion error was thrown because certain edge cases were not covered.

When a crash occurs during verification of the set of mutated constraints, this can be caused by either the backend solver or the modeling system. For example, during the development of our tool, a crash in a solver was caused by an integer overflow error – causing the solver to return an error message. An example when CPMpy was identified to be the cause of a crash happened when one of the interfaces to a solver did not implement all primitive constraints properly.

Most crashes are easy to detect in the day-to-day use of modeling systems as a user always receives an error message. Still, the severity of a crash can vary widely as it mostly depends on how the system is used. E.g., when the modeling system crashes when used in an integrated system of a manufacturing plant, the crash has likely far greater implications compared to when it is used in an interactive session.

7.3 Performance issues

The last type of bugs we identified are related to the performance and efficiency of the library. For example, when we verify whether the mutated model satisfies at least one solution, the time it takes for the modeling system to receive an answer from the solver may be significantly higher compared to the original model. This can again have several reasons caused by either the modeling system or the solver. For example, the mutated model may contain global constraints which get decomposed in a particularly inefficient way when nested by HURRICANE. Sometimes, either the solver or modeling system may even get stuck in an infinite loop! In practice we overcome this by setting a hard time-limit on the call to the verification method. Naturally, this may trigger false-positives as the mutated model may simply be harder to solve due to the surplus in variables and constraints. Still, we log these bugs as it may uncover interesting inefficiencies in the code.

7.4 Minimizing buggy models

The mutations defined in this paper can result in very large and deeply nested constraint models. However, often only a (small) subset of the constraints are the root cause of the bug. In our work, we utilize a simple deletion-based method that iteratively removes a single constraint from the model as long as the remaining model exhibits the bug. This method is similar to delta-debugging and is often used in combination with fuzz testing [44]. It should be noted that a crash of the system often gives some sort of message pointing to the expressions that caused the crash. Therefore, we deem delta debugging to be especially useful when dealing with a soundness bug.

Another way to simplify the debugging process is by automatically detecting bugs that are already identified. HURRICANE will keep logging a bug until it is fixed, so the same bug will be logged many times over. A first way to find out which of the bugged models are cause by Bug X, is to fix Bug X and then simply check which buggy models do no longer exhibit a bug. It's of course not always possible to quickly fix a bug, even after it is identified. We then turn to matching the error messages and location of the error in the code, as well as the input model or transformation that lead to the bug. For soundness bugs we can compare the results of multiple solvers to see if they match. This is enough information to confidently categorise most bugs in a semi-automated process.

8 Summary of found bugs

We coded up HURRICANE in Python 3.11 for CPMpy using the mutations and verification methods described previously. During development, which covers a period of about 1.5 years, we discovered **52 unique bugs** in total. This includes 19 bugs found in CPMpy during a master thesis that preceded this work [20]. Out of all bugs discovered, **13 bugs where soundness bugs**, 5 of which had their origin in backend solvers. In particular, we found 2 soundness bugs in the OR-tools solver and three in the MiniZinc system. The vast majority of bugs (29) were crashes of the CPMpy runtime environment. One of these crashes was traced back to a backend solver crashing. Lastly, we found three performance issues, one of which was again found in a backend solver.

Out of these 52 bugs, 14 remained at the time of the experiments described in the next section: 6 bugs in backend solvers and 8 in CPMpy. We shortly discuss these bugs in Appendix A. Full experimental data is also shown there.

9 Experimental evaluation

In this section, we investigate each of the components of our fuzz testing framework. In particular, we aim to answer the following experimental questions:

EQ1. What are the tradeoffs between increasing the number of mutations on each model and increasing the number of models being tested?

EQ2. How effective are the different verification methods for finding bugs in constraint modeling systems?

EQ3. To what extend does fuzz testing improve the overall coverage of tested code, compared to CPMpy's builtin suite of unit tests?

We configure HURRICANE to use different numbers of mutations and different types of verification methods. We test each of the five verification methods described in Sections 6.4 - 6.5 separately. For each of the verification methods, we employ four numbers of mutations applied to the input model before verification: $n = \{1, 2, 5, 10\}$. As backend solvers, we test the OR-Tools CP-SAT solver v.9.9 and MiniZinc v.2.8.3 with Gecode version 6.3.0. This combination of settings results in a total of 40 configurations, each of which was ran for 10 hours on an Ubuntu 20.04.6 LTS machine with an Intel Core i7-2600 CPU@3.40Ghz and 16GB of RAM. During these experiments, we keep track of which lines in CPMpy's code-base are executed using the `coverage` utility in Python.

We used 1240 constraint models as input, 7 of which are optimization problems. As discussed in Section 4, the models were extracted from the unit tests of CPMpy[3]. In the following sections, we aggregate the results of the above evaluation in order to answer the experimental questions.

[3] all code and input data can be found at `https://github.com/CPMpy/fuzz-test`

9.1 EQ1: effect of number of mutations

In this first experiment, we investigate the influence of the number of mutations (n) used in Algorithm 1 before verifying the mutated models. The more mutations used, the more diverse the output can be, and the more likely it is for a bug to be found. This can clearly be seen from the #unique column in Table 2 where we notice a steady increase in number of unique bugs found, with respect to the number of applied mutations. Notice this number of unique bugs is not in direct correlation with the number of errors reported. E.g., when testing OR-Tools and using two mutations before verification, many errors with the same root-cause (bug) are found by HURRICANE.

Mutations can increase the size of a model hyper-linearly: when applying a transformation such as flattening or decomposing global constraints, a single constraint can easily become a large set of constraints. Hence, it is likely the subsequent mutations will be slower as they have to run on bigger input, as does the verification check. From the #models column in Table 2, we can indeed conclude more mutations will result in less models tested for the given time-frame of ten hours.

The optimal value for n will of course depend on the time HURRICANE is ran for, since for smaller n we can find bugs more quickly, but for big n we expect to find those bugs *eventually*. We therefore propose that the best way of using HURRICANE would be to increase n over time, causing the easily detected bugs to get found quickly while making it possible to find the more obscure bugs later on.

Table 2 Number of mutations for each iteration compared to the number of bugs found and number of models handled. (Aggregated over the different verification methods).

	OR-Tools			MiniZinc			Total
#mutations	#models	#errors	#unique	#models	#errors	#unique	#unique
1	9166418	5747	1	218377	289	3	3
2	6672588	11002	3	216527	723	6	6
5	2270441	8975	5	128884	1495	8	11
10	344710	2783	7	57191	423	9	13

9.2 EQ2: effect of verification methods

The next dimension of our algorithm we investigate is the different types of verification methods. We aggregate the results for this experiment for all number of mutations. I.e., the results as reported in Table 3 result from testing the algorithm with all settings of n.

First of all, we notice a big difference in the amount of models that the different methods can verify. The results for the optimization verification method should be interpreted cautiously, because they run on a smaller subset of input models that have an objective function. These models happen to be small, explaining why the optimization verification solves more models than we would expect it to. More interesting is the difference in the number of models checked for the satisfiability and 1-solution verifications compared to counting and equivalence. This however does not translate to a large advantage in discovered bugs, indicating the usefulness of the computationally more expensive counting and equivalence verifications.

The 1-Solution verification performs best, regarding the number of unique bugs. This can be understood because it is a stronger check than the satisfiability check, but seems even faster. This is due to the fact that we send the instantiated solution to the solver when verifying the mutated model, leading to faster propagation.

Interestingly we observe that the solution counting, 1-solution and optimization methods all found at least 1 bug that was not detected by any of the other methods. This was not the case for All-solutions or satisfiability checking, and we could consider those redundant in the context of our experiments. Although verifying All-solutions is theoretically a stronger check than solution counting, and they can test models at a similar speed, both methods found bugs that the other did not. For example in an earlier experiment a bug was found in the `solveAll` routine of CPMpy, only detected using solution count. This highlights the advantage of using different verification methods to cover all aspects of the toolchain.

Table 3 Number of verification steps and errors found for different verification methods in 40 hours. (Aggregated over the different values of n).

	OR-Tools			MiniZinc			Total
verification	#models	#errors	#unique	#models	#errors	#unique	#unique
All sol	13441	460	4	11167	312	7	8
Counting	14551	539	5	11623	325	6	8
One sol	4095185	25695	5	194495	1983	8	10
Sat	3679400	180	4	186119	116	5	8
Opt	10651580	1633	2	217575	194	3	4

9.3 EQ3: effect on code coverage

As mentioned in Section 2, code coverage is a common proxy to measure the efficacy of a test suite. In this experiment, we compare the code coverage of running all unit test *models* (**unit-models**), running HURRICANE for 400 hours with these unit test models (200 hours for each backend solver) (**HURRICANE**), running all unit tests (not just the models that appear in them) (**unit-tests**), and the combined code coverage (**combined**) of **HURRICANE** and **unit-tests**.

The results are presented in Table 4. The data in this table is shown for the different solvers, with each sub-row representing a part of the code base. *expressions* contains the construction and evaluation code for all expressions (operators, functions, global constraints, etc.), *transformations* the internal transformation routines, and *ortools.py* and *minizinc.py* contain the solver-specific interfacing code

The results show that HURRICANE improves code coverage over just solving the unit models, but not over running all unit tests. Still, HURRICANE does cover new parts of the code, as the combined coverage is higher than just unit tests on its own. Because HURRICANE uses the internal transformations as mutations, we see a high code coverage on *transformations* too, even when using a solver like MiniZinc that requires only a few of the transformations in CPMpy.

Table 4 Segmented code coverage for different components of CPMpy.

Solver	files	**unit-models**	**HURRICANE**	**unit-tests**	**combined**
OR-Tools	*expressions*	54.6%	**64.6%**	87.3%	**88.6%**
	transformations	59.3%	**83.6%**	86.4%	**88.2%**
	ortools.py	64.1%	**81.5%**	90.4%	**91.5%**
MiniZinc	*expressions*	51.1%	**64.0%**	87.3%	**88.6%**
	transformations	22.1%	**82.6%**	86.4%	**88.2%**
	minizinc.py	70.6%	**84.3%**	83.0%	**89.2%**

10 Discussion and future work

We presented a method to automatically test constraint modeling languages given a set of input CSPs and COPs. We show that a sufficiently diverse set of input models can be obtained from the unit tests of the modeling language. Based on recent work in SMT-testing, we proposed a set of mutations to use over these models, in order to generate new and more complex inputs to CP modeling languages.

As shown in Section 9, our method is able to find a significant number of bugs for the CPMpy framework and its solvers, ranging from crashes to soundness bugs and finding downstream bugs in MiniZinc and OR-Tools. Moreover, using our framework improves the code coverage compared to the unit testing implemented in the library. Our proposed fuzz testing techniques also neatly allows *continuous integration* with modeling language development: when new features and bug fixes are added to a modeling language, the fuzz testing framework can just continue with the latest version on some remote server, testing the codebase 24/7.

While our methods are highly effective in finding bugs, one of the major difficulties remains how to avoid re-finding similar bugs, and producing minimal bug instances. We leave this topic for future investigation. Compared to testing SMT-solvers, CP offers several interesting dimensions on which we only briefly touched in this paper. These features include optimization, which can be tested more thoroughly in the future by also mutating objective functions. Another key feature of CP is the notion of global constraints. Based on [9], we would like to include mutations which can introduce *new* global constraints into the models as currently we rely on the global constraints already being present in the input.

Recent work in SMT-solving showcases the power of using voting between multiple solvers to verify the answer any of the solvers produce [41]. Crucially, solver voting allows to use mutations where the result of the solver does not have to be known upfront, i.e., one does not have to know what properties the mutations have. Using multiple solvers perfectly suits the testing of constraint modeling languages, as their core function is to translate constraint specifications to multiple solvers and solving paradigms. We are optimistic that this work will remain useful in the future, by applying it to more solvers, adding more mutations, and encouraging more developers to make use of it.

References

1 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*, page 51, 1992.

2 Özgür Akgün, Alan M. Frisch, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artif. Intell.*, 310:103751, 2022. doi:10.1016/j.artint.2022.103751.

3 Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, 2018. doi:10.1007/978-3-319-98334-9_46.

4 Mario Alviano, Carmine Dodaro, Johannes Klaus Fichte, Markus Hecher, Tobias Philipp, and Jakob Rath. Inconsistency proofs for ASP: the ASP - DRUPE format. *Theory Pract. Log. Program.*, 19(5-6):891–907, 2019. doi:10.1017/S1471068419000255.

5 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 W. Barrett. Flexible proof production in an industrial-strength SMT solver. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 15–35. Springer, 2022. doi:10.1007/978-3-031-10769-6_3.

6 Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog, (revision a), 2012.

7 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 - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 2016. doi:10.1007/978-3-319-44953-1_4.

8 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pages 3698–3707. AAAI Press, 2022. URL: https://ojs.aaai.org/index.php/AAAI/article/view/20283, doi:10.1609/AAAI.V36I4.20283.

9 Mauro Bringolf. Fuzz-testing smt solvers with formula weakening and strengthening. Master's thesis, ETH Zurich, 2021.

10 Robert Brummayer and Armin Biere. Fuzzing and delta-debugging smt solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 1–5, 2009.

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

12 Nadia Creignou and Miki Hermann. Complexity of generalized satisfiability counting problems. *Inf. Comput.*, 125(1):1–12, 1996. doi:10.1006/inco.1996.0016.

13 Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, pages 201–211. IEEE Computer Society, 2014. doi:10.1109/ISSRE.2014.11.

14 Xavier Gillard, Pierre Schaus, and Yves Deville. Solvercheck: Declarative testing of constraints. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming*, pages 565–582, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-30048-7_33.

15 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Veripb: The easy way to make your combinatorial search algorithm trustworthy. In *workshop From Constraint Programming to Trustworthy AI at the 26th International Conference on Principles and Practice of Constraint Programming (CP'20). Paper available at http://www. cs. ucc. ie/bg6/cptai/2020/papers/CPTAI_2020_paper_2. pdf*, 2020.

16 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. 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 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CP.2022.25.

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

19 Ahmet B. Keha, Ketan Khowala, and John W. Fowler. Mixed integer programming formulations for single machine scheduling problems. *Comput. Ind. Eng.*, 56(1):357–367, 2009. doi:10.1016/j.cie.2008.06.008.

20 Ruben Kindt and Mattias Guns. Fuzz testing of constraint programming. Master's thesis, KU Leuven, 2023. URL: https://kuleuven.limo.libis.be/discovery/fulldisplay?docid=alma9993364582101488&context=L&vid=32KUL_KUL:KULeuven&search_scope=All_Content&tab=all_content_tab&lang=en.

21 Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 171–182. ACM, 2008. doi:10.1145/1328438.1328461.

22 Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated unit test generation for python. *CoRR*, abs/2007.14049, 2020. arXiv:2007.14049.

23 Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 701–712. ACM, 2020. doi:10.1145/3368089.3409763.

24 Matthew J. McIlree and Ciaran McCreesh. Proof logging for smart extensional constraints. 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 26:1–26:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CP.2023.26.

25 Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990. doi:10.1145/96267.96279.

26 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

27 Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, and Ian Miguel. Automatically improving constraint models in savile row through associative-commutative common subexpression elimination. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 590–605. Springer, 2014. doi:10.1007/978-3-319-10428-7_43.

28 Peter Nightingale, Patrick Spracklen, and Ian Miguel. Automatically improving SAT encoding of constraint problems through common subexpression elimination in savile row. 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 330–340. Springer, 2015. doi:10.1007/978-3-319-23219-5_23.

29 Tobias Paxian and Armin Biere. Uncovering and classifying bugs in maxsat solvers through fuzzing and delta debugging. Update reference when published, 2022. URL: http://www.pragmaticsofsat.org/2023/live/POS23_paper_4.pdf.

30 Andrea Rendl. *Effective compilation of constraint models*. PhD thesis, University of St Andrews, UK, 2010. URL: https://hdl.handle.net/10023/973.

31 Robert Robere, Antonina Kolokolova, and Vijay Ganesh. The proof complexity of SMT solvers. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 275–293. Springer, 2018. doi:10.1007/978-3-319-96142-2_18.

32 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.

33 Olivier Roussel and Christophe Lecoutre. XML representation of constraint networks: Format XCSP 2.1. *CoRR*, abs/0902.2362, 2009. arXiv:0902.2362.

34 Gilles Simonin, Christian Artigues, Emmanuel Hebrard, and Pierre Lopez. Scheduling scientific experiments for comet exploration. *Constraints An Int. J.*, 20(1):77–99, 2015. doi:10.1007/S10601-014-9169-3.

35 Hussain Bilal Syed. *Model selection and testing for an automated constraint modelling toolchain*. PhD thesis, University of St Andrews, UK, 2017. URL: https://hdl.handle.net/10023/10328.

36 Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. *JUnit in Action, 2nd Edition*. Manning Publications Company, 2011. URL: https://www.manning.com/books/junit-in-action-second-edition.

37 Willem Jan van Hoeve. The alldifferent constraint: A survey. *CoRR*, cs.PL/0105015, 2001. URL: https://arxiv.org/abs/cs/0105015.

38 Willem-Jan van Hoeve and Irit Katriel. Global constraints. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 169–208. Elsevier, 2006. doi:10.1016/S1574-6526(06)80010-6.

39 Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. Qmaxsatpb: A certified maxsat solver. In Georg Gottlob, Daniela Inclezan, and Marco Maratea, editors, *Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, 2022. doi:10.1007/978-3-031-15707-3_33.

40 T.W. Williams, M.R. Mercer, J.P. Mucha, and R. Kapur. Code coverage, what does it mean in terms of quality? In *Annual Reliability and Maintainability Symposium. 2001 Proceedings. International Symposium on Product Quality and Integrity (Cat. No.01CH37179)*, pages 420–424, 2001. doi:10.1109/RAMS.2001.902502.

41 Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. *Proc. ACM Program. Lang.*, 4(OOPSLA):193:1–193:25, 2020. doi:10.1145/3428261.

42 Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT solvers via semantic fusion. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 718–730. ACM, 2020. doi:10.1145/3385412.3385985.

43 Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In René Mayrhofer, Luke Chen, Matthias Steinbauer, Gabriele Kotsis, and Ismail Khalil, editors, *The 11th International Conference on Advances in Mobile Computing & Multimedia, MoMM '13, Vienna, Austria, December 2-4, 2013*, page 68. ACM, 2013. doi:10.1145/2536853.2536881.

44 Andreas Zeller. Yesterday, my program worked. today, it does not. why? In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1999. doi:10.1007/3-540-48166-4_16.

A Overview of bugs found during experimental evaluation

We identify 2 OR-Tools bugs, 4 MiniZinc bugs and 8 CPMpy bugs, and give a short description in this section.

Bug 1

Some mutated models are declared unsatisfiable when solving them using Gecode through its MiniZinc interface. Solving with another solver confirms that the models are in fact satisfiable. This is a critical soundness bug.[4]

Bug 2 & 3

The next 2 bugs are also considered soundness bugs in MiniZinc but are not as severe as the first one. There are some models where MiniZinc does not output a value for all the variables after solving. This happens for most but not all of the available solvers within MiniZinc. The reason we count 2 different bugs is that a third similar bug has already been solved after HURRICANE found it earlier on, but this didn't resolve the ones we found here. Further distinction lies in the fact that Bug 2 occurs when solving to satisfiability and Bug 3 happens when solving to optimality.[5]

Bug 4

When using MiniZinc Python some models do not respect the given time limit when solving. This is due to the compiler optimisation phase getting stuck.[6]

Bug 5

A bug in CPMpy's MiniZinc interface, that causes a crash when a nested sum appears in the arguments of the global constraint: AllDifferentExcept0.[7]

Bug 6

A bug in CPMpy's MiniZinc interface, that causes a crash when the Count global constraint appears as an argument in a weighted sum.[8]

[4] https://github.com/MiniZinc/MiniZincIDE/issues/199
[5] https://github.com/MiniZinc/libminizinc/issues/803
[6] https://github.com/MiniZinc/libminizinc/issues/805
[7] https://github.com/CPMpy/cpmpy/issues/460
[8] https://github.com/CPMpy/cpmpy/issues/461

Bug 7

The helper function canonical_comparison contained a bug where weighted sums were incorrectly transformed. This is a soundness bug.

Bug 8

Inconsistent implementation of the relational semantics for constraint modeling languages meant that handling of partial functions such as ELEMENT leads to missing solutions where the constraint is undefined, but occurs in a nested context.

Bug 9

CPMpy's helper function is_bool did not recognise a specific datatype to be Boolean.[9]

Bug 10

The internal transformation canonical_comparison can create weighted sums with zero arguments, leading to a crash later in the transformation pipeline.

Bug 11

An assertion error gets triggered in the internal function canonical_comparison, when a CPMpy sum operator is encountered that only contains integers and no variables.

Bug 12

An equation between an integer and a Boolean expression was treated as reification by the flatten transformation of CPMpy.[10]

Bug 13

Crash in the OR-Tools solver causing the Python runtime environment to crash.[11]

Bug 14

A soundness bug in OR-Tools' presolve where the ordering of constraints influences whether a model was declared to be satisfiable or not. [12]

A.1 Occurrences of each bug

In Table 5 and Table 6, we show the unaggregated data of how many times each bug was found by HURRICANE during our experimental evaluation.

[9] https://github.com/CPMpy/cpmpy/issues/452
[10] https://github.com/CPMpy/cpmpy/issues/442
[11] https://github.com/google/or-tools/issues/4169
[12] https://github.com/google/or-tools/issues/4168

Table 5 Bugs found by different verification methods when running with MiniZinc.

Verif	#mut	B1	B2	B3	B4	B5	B6	B7	B8	B9	#bugs	#models
All sol	1	-	-	-	-	1	-	-	23	-	24	3492
	2	-	-	-	-	2	-	-	59	-	61	3594
	5	2	-	-	-	5	1	36	86	13	143	3242
	10	-	-	-	1	1	-	38	28	16	84	839
counting	1	-	-	-	-	1	-	-	24	-	25	3633
	2	-	-	-	-	-	-	-	61	-	61	3655
	5	2	-	-	-	3	1	40	94	15	155	3496
	10	1	-	-	-	1	-	39	28	15	84	839
One sol	1	-	-	66	-	15	-	-	133	-	214	65029
	2	-	-	68	-	29	4	1	429	2	533	64725
	5	2	-	91	-	108	8	12	903	32	1156	61554
	10	-	-	6	1	7	-	3	60	3	80	3187
sat	1	-	-	-	-	26	-	-	-	-	26	88981
	2	-	-	-	-	61	6	-	-	1	68	87419
	5	1	-	-	1	6	1	-	-	2	11	6554
	10	1	-	-	-	6	1	-	-	3	11	3165
opt	1	-	-	-	-	-	-	-	-	-	-	57242
	2	-	-	-	-	-	-	-	-	-	-	57134
	5	2	-	-	-	-	-	28	-	-	30	54038
	10	2	7	-	-	-	-	155	-	-	164	49161

Table 6 Bugs found by different verification methods when running with OR-Tools.

Verif	#mut	B7	B8	B9	B10	B11	B12	B13	B14	#bugs	#models
All sol	1	-	26	-	-	-	-	-	-	26	4102
	2	-	64	-	-	-	-	-	-	64	3786
	5	37	95	13	-	-	-	-	-	145	3332
	10	116	70	38	-	-	-	-	1	225	2221
Counting	1	-	26	-	-	-	-	-	-	26	4152
	2	-	69	-	-	-	-	-	-	69	4128
	5	42	117	16	-	-	-	1	1	177	3718
	10	139	78	43	-	-	-	-	7	267	2553
One sol	1	-	5695	-	-	-	-	-	-	5695	2226130
	2	6	10761	79	-	-	-	-	-	10846	1400180
	5	84	7449	212	1	-	-	-	-	7746	412874
	10	71	1250	80	1	6	-	-	-	1408	56001
Sat	1	-	-	-	-	-	-	-	-	-	1958248
	2	4	-	19	-	-	-	-	-	23	1292747
	5	40	-	55	1	-	-	-	-	96	379361
	10	28	-	29	1	3	-	-	-	61	49044
Opt	1	-	-	-	-	-	-	-	-	-	4973786
	2	-	-	-	-	-	-	-	-	-	3971747
	5	811	-	-	-	-	-	-	-	811	1471156
	10	820	-	-	-	-	2	-	-	822	234891

B Reformulations as mutations

We summarize the constraint reformulations implemented in CPMpy which are used in our mutational testing framework.

Unnesting and normalization of lists

This transformation is the first in the transformation pipeline of any solver implemented in CPMpy and all subsequent transformation expect as input a flat list of constraints. This Additionally any conjunction at the top-level of the constraint model will be split up into separate constraints

$$M_{unnest}([c_1, [c_2, c_3], [c_4 \wedge c_5]])$$

with $c_n, n \in 1..5$ being arbitrary constraints, results in

$$[c_1, c_2, c_3, c_4, c_5]$$

Flattening

Makes sure no nested constraints remain in the expression tree. This reformulation introduces a fresh variable to be equated with a (numerical) expression and un-nests each constraint accordingly. The output of this reformulation is a set of Boolean expressions within a restricted grammar defined by CPMpy's developers. For example, given the expression list

$$[\textsc{AllDifferent}(\textsc{Min}(w, x), y, z)] \tag{2}$$

the result of the flattening is

$$[\textsc{AllDifferent}(e, y, z), e = \textsc{Min}(w, x)] \tag{3}$$

with e an auxiliary variable with the right bounds.

Decomposing global constraints

This function is one of the elementary operations in constraint modeling languages. While many CP-solvers support a variety of global constraints, these advanced relations between variables are oftentimes not supported by solvers from other solving paradigms. Hence, when a model containing a global constraint has to be solved by for example an SMT-solver, it needs to be decomposed into simpler expressions first. This reformulation does exactly that. For example, if AllDifferent is not supported by the solver, it is decomposed to a conjunction of pairwise disequality constraints.

Unnesting of reified constraints

This transformation is applied to ensure no unsupported expressions remain reified. For some of the backend solvers in the CPMpy library, reification is only supported on a subset of expressions. This reformulation is applied after flattening, and ensures further unnesting such that only reifications of supported constraints remain. For example, given the unsupported expression $b \rightarrow \textsc{Max}(x, y, z) \leq 10$, a valid transformation in order to remove the reification of the Max is

$$(b \rightarrow a \leq 10) \wedge (\textsc{Max}(x, y, z) = a) \tag{4}$$

with a an auxiliary variable with the appropriate bounds. Input constraints must not contain unsupported global constraints, and must be flattened first.

Only half-reification

It removes all "full reification constraints" from the expression tree and ensures all reifications end up in the form $b \rightarrow \mathit{bexpr}$. This transformation always has to be preceded by the previous *only boolean variables reify* transformation. For each constraint of the type $b \leftrightarrow \mathit{bexpr}$, two half-reification constraints are introduced: $b \rightarrow \mathit{bexpr}$ and $\neg b \rightarrow \neg \mathit{bexpr}$. This transformation also simplifies the negated Boolean expression whenever possible. For example, given $b \leftrightarrow x \wedge y$ as input, the transformation returns $\{b \rightarrow x \wedge y$ and $\neg b \rightarrow (\neg x \vee \neg y)\}$.

Normalization of reifications

This transformation rewrites any reification such that the Boolean variable occurs on the left hand side. E.g., constraints of the type $\mathit{bexpr} \rightarrow b$ are rewritten to $\neg b \rightarrow \neg \mathit{bexpr}$, full-reification constraints $\mathit{bexpr} \leftrightarrow b$ are swapped to $b \leftrightarrow \mathit{bexpr}$. Similar to the previous transformation, negated Boolean expressions are simplified when possible. Input constraints must be flat.

Linearize

It ensures any flattened constraint is transformed into a canonicalized linear constraint, i.e., a comparison with a weighted sum of integer or Boolean variables on the left-hand side and a constant on the right-hand side. The ouput is thus always of the form

$$\sum w_i x_i \quad \langle \mathit{cmp} \rangle \quad c$$

where $\langle \mathit{cmp} \rangle$ is the one of the comparison operator allowed ($=, \leq$ or $\geq$), the w_i are the integer weights and x_i the Boolean/integer variables. Before linearizing, unsupported global constraints must be decomposed, and must contain only boolean implications.

Normalized numerical expressions

This transformation is targeted to be used with solvers that don't support comparisons ($<$, $\leq$, $\geq$, $>$, $\neq$) between an expression and a constant. An auxiliary variable is thus required to transform it into a simple comparison. For example, if $\textsc{Max}(x, y, z) \leq 10$ is not supported, it will be transformed into

$$(\textsc{Max}(x, y, z) = e) \wedge (e \leq 10) \tag{5}$$

by using the auxiliary variable e (with appropriate bounds). Input constraints must be flat.

Converting negated Boolean variables

After linearization of a set of constraints, it helps make the constraints more compatible with the API of a typical Mixed Integer Programming solver. Pseudo-Boolean constraints (weighted-sum over Boolean variables) are converted such that only positive Boolean variables remain on the left-hand side of the comparisons. For example, the expression $\neg p + q + r \geq 1$ is re-written as $-p + q + r \geq 0$ by creating a negative weight and allowing no negation operator in the formula. Input constraints must be linear.

Conversion of flat expressions to CNF

It is required when using SAT-solvers as backend solvers. This transformation rewrites any Boolean operator with Boolean variables as arguments to CNF. For example, $(w \wedge x) \vee (y \wedge z)$ is re-written in

$$(w \vee y) \wedge (w \vee z) \wedge (x \vee y) \wedge (x \vee z) \tag{6}$$

Input must ensure only boolean implications

Push negation to leaves

This one simplifies the number of nodes in the expression tree. The reformulation applies simple equivalence rules such as DeMorgan's laws to make sure the only negation operators left in the tree are bound to Boolean variables or global constraints. For example, it would transform the expression $\neg(a \vee b)$ into $\neg a \wedge \neg b$, or the expression $\neg(a \leq b)$ into $a > b$. The negation of a global constraint such as $\neg\textsc{AllDifferent}(a, b, c)$ can not be simplified any further, except by decomposing the global constraint first. This will happen in the "decomposing globals" transformation, depending on solver support.

Simplification of Boolean comparisons

This operation can be done when a Boolean expression is compared to a constant. In that case, it is trivial to convert the Boolean expression at hand to itself or to its negation. For example, comparison $b < 1$, where b is a Boolean variable, can be simplified to $\neg b$. And $b \geq True$ can be converted to just the literal b.

Learning Precedences for Scheduling Problems with Graph Neural Networks

Hélène Verhaeghe[1]
DTAI, KU Leuven, Belgium

Quentin Cappart
Polytechnique Montréal, Canada

Gilles Pesant
Polytechnique Montréal, Canada

Claude-Guy Quimper
Université Laval, Quebec, Canada

Abstract

The *resource constrained project scheduling problem* (RCPSP) consists of scheduling a finite set of resource-consuming tasks within a temporal horizon subject to resource capacities and precedence relations between pairs of tasks. It is $\mathcal{NP}$-hard and many techniques have been introduced to improve the efficiency of CP solvers to solve it. The problem is naturally represented as a directed graph, commonly referred to as the *precedence graph*, by linking pairs of tasks subject to a precedence. In this paper, we propose to leverage the ability of *graph neural networks* to extract knowledge from precedence graphs. This is carried out by learning new precedences that can be used either to add new constraints or to design a dedicated variable-selection heuristic. Experiments carried out on RCPSP instances from PSPLIB show the potential of learning to predict precedences and how they can help speed up the search for solutions by a CP solver.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Computing methodologies → Machine learning; Mathematics of computing → Combinatorial optimization

Keywords and phrases Scheduling, Precedence graph, Graph neural network

Digital Object Identifier 10.4230/LIPIcs.CP.2024.30

Funding This research received funding from IVADO and the Canada First Research Excellence Fund / Apogée fund PostDoc-2022-2378128196 and in part by NSERC Discovery Grants 218028/2017. It also received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 101070149, project Tuples.

Acknowledgements We thank the anonymous reviewers for their constructive criticism which helped us improve the original version of the paper.

1 Introduction

Scheduling problems arise in many fields, from assembling planes to scheduling maintenance tasks. Constraint programming (CP) has been successfully used to solve many types of scheduling problems [35, 29]. This is mainly due to the combination of global constraints and efficient dedicated heuristics used when solving such problems. When they are subject to precedence constraints, scheduling problems are often $\mathcal{NP}$-hard [20]. Nevertheless, every such precedence may help to improve the inferences made by global constraints. Precedences are naturally represented as a directed graph by linking two tasks subject to a precedence. *Graph neural networks* (GNNs) [28] are designed to learn from graph-structured data, including

[1] The first author was affiliated to Polytechnique Montréal during the majority of this work.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 30; pp. 30:1–30:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

deciding whether an edge is present. A natural question our paper tries to answer is then: Can GNNs help us identify additional precedences? And as a follow-up question: How useful can these learned precedences be?

Following this idea, this paper proposes to leverage the ability of GNNs to learn new precedences between tasks for the standard *resource constrained project scheduling problem* (RCPSP) [26]. These precedences are used to enhance the solving process of a CP solver, both by adding new constraints in the model and by designing a dedicated variable-selection heuristic. The learning is carried out from a precedence graph representation of the RCPSP and the GNN is trained using best-so-far solutions found by solving, up to a fixed time limit, a set of instances specially considered for training. The models are trained and evaluated on RCPSP instances from PSPLIB [19]. The results obtained show the promise of learning to predict precedences and their relevance for speeding up the search for solutions by a CP solver.

Our approach shows that it is possible to learn precedences using a simple GNN architecture. In synergy with the dynamic metaheuristic SBPS [9, 36] and VSIDS [24], the learned precedences manage to improve our baseline (i.e., solving the problem without any additional precedences). Using the learned preferences as additional constraints allows to get better bounds, but comes with the drawback of potentially deteriorating the optimum for each wrong precedence added. Using the preferences as a piece of information to drive the search preserves the optimum while leading to better first solutions.

The paper is structured as follows. The next section presents the technical background regarding existing solving processes for the RCPSP, graph neural networks, and the learning of heuristics. Then the methodology we introduce to learn and leverage precedences is described. Finally the experiments carried out and the results obtained are discussed.

2 Technical Background and Related Work

2.1 Resource Constrained Project Scheduling Problem

An instance of the *resource constrained project scheduling problem* (RCPSP) [26] consists of a finite set of n tasks (or activities) T to be executed with the help of a finite set of m resources R. Each resource $r \in R$ has a finite capacity C_r, and each task $i \in T$ has a starting time s_i, an ending time e_i, and is executed without interruption during p_i units of time (i.e. we have $s_i + p_i = e_i$) while using c_{ir} units of each resource r. In addition, there are precedences between some pairs of tasks. We say that task i precedes task j if the execution of i should be finished before starting the execution of j (i.e., we have $e_i \leq s_j$).

We call *precedence graph* of an RCPSP instance the directed graph $P = (V, E)$ where there exists an injective function $M : T \to V$ mapping each task to a vertex. If and only if task i precedes task j, there exists an arc going from $M(i)$ to $M(j)$. We call the *transitive closure* of an RCPSP instance the set of precedences which also contains all the transitive precedences: given task i, j and k, if the instance defines that i precedes j and j precedes k, we know by transitivity that i precedes k. We call "i precedes k" a transitive precedence. The *transitive closure precedence graph* is thus the precedence graph representing the transitive closure of the instance. The `cumulative` constraint is one of the main components of solving RCPSP problems using CP. Its filtering algorithms [35, 29] prevent the resources from being overloaded.

Many heuristics are efficient in solving the RCPSP problem [8]. The most effective are *variable state independent decaying sum* (VSIDS) [24] and *solution-based phase saving* (SBPS) [9, 36]. The heuristic VSIDS uses counters for each variable. Each counter is incremented when a constraint involving the variable is generated. The choice of variable and value is based on the values of these counters. On the other hand, SBPS is only a value-selection heuristic. Each time a solution is found, the values are stored and when a variable is selected, the value to branch on is the one used in the last solution found. We note that both heuristics can be used together.

2.2 Graph Neural Network

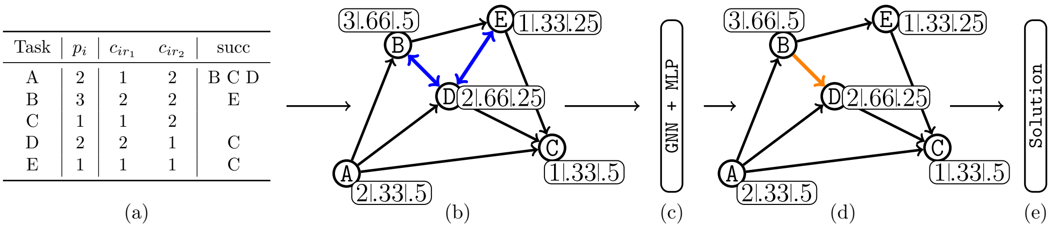
A *graph neural network* (GNN) [28] is a specific neural network architecture dedicated to computing a vector representation, also referred to as *embedding*, for each node of a graph given as input [28, 38] (e.g., a precedence graph). This is carried out by aggregating several times information from neighboring nodes. Each aggregation operation is referred to as a *layer* of the GNN and involves weights that must be learned. This operation can be performed in many ways, and there exist in the literature different variants of GNNs, such as graph attention network [34] or gated graph sequence neural network [21]. Our work is based on the well-known *GraphSAGE* architecture [15], which can efficiently generate representations for graphs unseen during the training phase. Formally, let $G = (V, E)$ be a graph where V is the set of vertices, and where E is the set of edges. A GNN is composed of K layers. Let $h_v^k \in \mathbb{R}^d$ be a d-dimensional vector representation of a node $v \in V$ at layer k, and let $h_v^{k+1} \in \mathbb{R}^l$ be a l-dimensional vector representation of v at the next layer. We highlight that the dimension of the representation can be different at each layer (d for the layer k, and l for the layer $k+1$). The inference process of a GNN consists in computing the next representation (h_v^{k+1}) from the previous one (h_v^k) for each node v. The fundamental equations of *GraphSAGE* are given in Equations (1) to (3):

$$a_v^{k+1} = \mathsf{AGGREG}\left(h_u^k \;\middle|\; u \in N(v)\right) \qquad \forall v \in V \qquad (1)$$

$$c_v^{k+1} = g\left(\Theta^k \times \mathsf{CONCAT}\left(h_v^k, a_v^{k+1}\right)\right) \qquad \forall v \in V \qquad (2)$$

$$h_v^{k+1} = \frac{c_v^{k+1}}{||c_v^{k+1}||_2} \qquad \forall u \in V \qquad (3)$$

Three operations are carried out. First, Eq. (1) aggregates information from neighbors for each node. This is often done by taking the mean of each value (*mean-pooling*) or their sum (*sum-pooling*). A key aspect of *GraphSAGE* aggregation is that the neighborhood function $N(v)$ gives a fixed-size random subset of nodes sampled from the whole neighborhood of node v. A new subset is randomly sampled at each layer. Second, Eq. (2) concatenates the current representation of a node $h_v^k \in \mathbb{R}^d$ with the aggregated representation of its neighbors $a_v^{k+1} \in \mathbb{R}^d$. This vector is then linearly transformed by weight matrix $\Theta^k \in \mathbb{R}^{l \times 2d}$ which is learned during the training phase through backpropagation. A non-linear transformation g such as ReLU is subsequently performed [13]. Third, the value obtained (c_v^{k+1}) is normalized using the Euclidean norm, and the node representation at layer $k+1$ is obtained (h_v^{k+1}). The process terminates after K layers, and a final representation (h_v^K) is obtained for each node v in the graph. Such representations can after that be used for different graph-related tasks, such as classification [11], link prediction [22, 39, 2], or combinatorial tasks [3]. In another context, GNNs are increasingly considered in combinatorial optimization [3], either directly as an end-to-end solver [17], or as a mechanism to enhance existing solvers [12, 4].



Task	p_i	c_{ir_1}	c_{ir_2}	succ
A	2	1	2	B C D
B	3	2	2	E
C	1	1	2	
D	2	2	1	C
E	1	1	1	C

Figure 1 Overview of the methodology introduced (on a toy example with $|R| = 2$). (a) An RCPSP instance, with $C_{r_1} = 3$ and $C_{r_2} = 4$. (b) The instance is transformed into a graph with the features of each node (time, normalized usage of r_1, and r_2). The blue arcs are the candidate arcs. (c) The GNN takes the graph as input and outputs an embedding used by the MLP to provide the prediction. (d) New predicted edges for the graph. (e) Prediction is used to find a solution.

2.3 Learning Heuristics in Constraint Programming

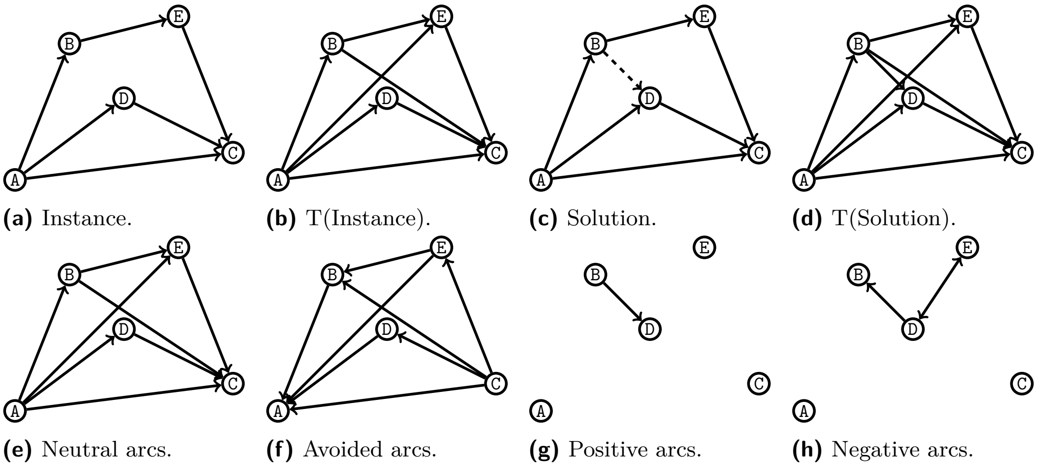
Designing branching heuristics for CP thanks to machine learning has already been considered in related works, either to learn a variable-selection or a value-selection heuristic. For instance, [30] proposed to combine reinforcement learning and graph neural networks to learn the next variable to branch for constrained satisfaction problems. In [32], graph neural networks were leveraged to initialize such a heuristic for a hybrid CP-SAT solver. Learning has also been considered for online settings by [10]. Concerning the value-selection heuristic, [4] introduced a framework able to leverage models trained with reinforcement learning inside a CP solver. It is done thanks to a recursive formalization. This idea has been further extended by [5, 23] who proposed to carry out the learning inside the solver. A general framework for learning value heuristics, and relevant features for training the model, has been also introduced by [7]. Compared to these works that aim to generate heuristics for a large range of problems and do not achieve state-of-the-art performance, our contribution is focused on scheduling problems and strive to improve upon competitive CP models. To do so, features specific in scheduling (i.e., precedence graphs) are leveraged. Recently, [31] proposed an approach close to ours. In their paper, they construct a graph with nodes representing either a task or a resource. Using GNNs, they predict the starting time of each of the tasks, then construct an ordering of the tasks based on these predicted starting times and finally use an algorithm (serial schedule generation scheme) to construct a feasible solution with respect to this ordering. They tackle classical RCPSPs and stochastic RCPSPs.

3 Enhancing CP with Learned Precedences

This section introduces the core contribution of the paper. It describes the complete methodology we designed to learn relevant precedences and how they can be used to speed up a CP solver. The methodology is based on the hypothesis that the knowledge of precedences belonging to a high-quality solution is information that is relevant to solving instances faster. Inside a CP solver, this can be achieved by enabling additional propagation (e.g., with additional constraints), or by directing the search. An overview of the methodology is illustrated in Fig. 1. The following sections describe each part of the methodology.

3.1 Step 1: Training Set Construction

The first step is to build a relevant dataset to train the model. To do so, the main idea is to: (1) solve a large number of RCPSP instances with an expensive solving process (e.g., a CP solver), (2) consider all the precedences obtained in the solutions generated, (3) use



(a) Instance. **(b)** T(Instance). **(c)** Solution. **(d)** T(Solution).

(e) Neutral arcs. **(f)** Avoided arcs. **(g)** Positive arcs. **(h)** Negative arcs.

Figure 2 Example of all the types of arcs for the same toy example as in Figure 1.

this information as data to train the model. The expected outcome is to obtain quickly precedences occurring in high-quality solutions of unseen instances, without requiring the execution of the expensive solving process. The learning is supervised as it requires a ground truth. Let $P = (V, E)$ be a precedence graph as defined previously (for example, the precedence graph defined in Fig. 2a), and let $D : \{(x^{(i)}, y^{(i)}), \dots, (x^{(n)}, y^{(n)})\}$ be the training set we need to build to train the model. In the machine learning terminology, $x^{(i)}$ corresponds to a specific *input* ($i \in \{1, \dots, n\}$), and $y^{(i)}$ corresponds to the *target value* associated to that input.

Let s^P be a solution of P, obtained by any solving process (Fig. 2c is the optimal solution in our example). We note that this solution corresponds to a more constrained precedence graph than P, with the same number of nodes, and more edges. A first important design choice is to determine *which solutions should be considered for the training?*

A natural idea is to take the optimal solution of previously solved instances. However, this is barely feasible in practice, as it assumes that there exists a solving process able to prove the optimality of these solutions in a reasonable time, which is intractable for challenging instances. For such a reason, we relaxed this idea and considered the best solution obtained instead, up to a predetermined timeout as the base for our first training set.

Then, we define $T(s^P)$ as the transitive closure of s^P (for example, Fig. 2d). This corresponds to a precedence graph $P' = (V', E')$ where $V' = V$ and $E' \supseteq E$. Before their integration in the training set, the solutions are transformed by their transitive closure (for example, Fig. 2d). This is done to reduce the diameter of the graphs. The intuition behind this design choice is that high-diameter graphs usually require deeper and more expensive GNNs to be efficient [6]. Each node V' is decorated with two types of features, recording information about the instances. They are as follows:

1. The processing time p_i of the task linked to node $i \in V'$.
2. The normalized usage of each resource r, i.e. the usage of the task i (c_{ir}) divided by the total availability of this resource within the instance (C_r).

As the instances we considered involve four resources each, we have five features for each node. Other features were considered (e.g., the proportion of the different resources used) but were not selected in the final model as they did not improve the performance. Finally, let us consider the set of pairs between vertices in P'. Each pair corresponds to a specific relation between two tasks in a solution. We identified four possible relations. Given $P = (V, E)$ (i.e., the transitive closure precedence graph of the instance) and $P' = (V', E')$ (i.e., the graph corresponding to the transitive closure precedence graph of a solution s^P, also referred to as $T(s^P)$), the four relations are as follows:

1. *Neutral arcs* (e.g., Fig. 2e): these arcs correspond to the ones included in E. Intuitively, such arcs relate to specific settings of the problem. They define the structure of the graph but are not relevant for computing the training loss.
2. *Avoided arcs* (e.g., Fig. 2f): considering all the arcs $(v, u) \in E$, the avoided arcs correspond to the set of arcs (u, v). Intuitively, we know that these arcs will never be part of a feasible solution because of the structure of the problem. Such arcs are not relevant for the loss either.
3. *Positive arcs* (e.g., Fig. 2g): these arcs correspond to the ones included in the set $E' \setminus E$. It corresponds to the additional precedences that are present in the solution obtained and missing in the initial graph. They are the arcs we want to be able to predict and should be used for the training.
4. *Negative arcs* (example Fig. 2h): these arcs correspond to all the other arcs that do not belong to any of the last three sets. Such arcs represent negative samples that can be used for training. The model should be able to predict that these arcs should not be part of a good solution.

In our first experiments, we considered only two categories and every arc was part of the loss computation. However, it quickly became evident that neutral and avoided arcs degraded the stability of the learning loss. This is what led us to consider these four categories. Only considering part of the arcs (i.e., the positive and negative ones) for the computation of loss is referred to as *masking* in the machine learning community [16]. It helps reduce the impact of noisy data. In our case, neutral and avoided arcs are arcs from which no information is learned and for which no prediction will be asked. However, they cannot be discarded totally as they are an inherent part of the problem and must be considered for the optimization phase. This forms our first training set, based on one best-so-far solution per instance considered.

In our experiments, we also considered a second training set based on multiple best-so-far solutions. This one is an extension of the previous one. Based on the first best-so-far gathered after a first time-out, we let the solver run for a second time-out, searching for at most K solutions with the same (or better, in case it was only a best-so-far) optimum. These solutions are then aggregated by keeping the precedences present in a majority of the solution (majority defined by a given threshold percentage). This aggregated solution allows us to get rid of the potentially noisy precedences, created by some tasks that have room to move a bit within the schedule, and focus on the ones more mandatory in an optimal solution. In this training set, the neutral and avoided arcs are the same, the positive arcs are the ones presents in the majority of the solutions and the negative are the remaining ones.

In summary, to perform the learning, a specific input $x^{(i)}$ corresponds to the graph obtained from the transitive closure of a precedence graph of a solution (or aggregate solution), with features associated to each vertex, and the related target $y^{(i)}$ is a real value 0 (negative) and 1 (positive) for each unmasked pair of vertices (i.e., a possible learnable precedence link).

3.2 Step 2: Link Prediction with GNNs

Provided with training data and supervised labels, the next step is to build a function $f : P(V, E) \rightarrow [0, 1]^{V \times V}$ taking a precedence graph annotated with the node features as input and giving as output a probability for each link $(u \rightarrow v) \in V \times V$ to correspond to a precedence occurring in a high-quality solution of the instance. We designed this function using two neural networks: (1) a graph neural network computing an embedding for each node, and (2) a fully connected neural network computing a probability for each link.

First, the graph neural network is a function $f_1 : P(V, E) \rightarrow V \times \mathbb{R}^d$ which computes a d-dimensional vectorized representation for each vertex $v \in V$. The architecture is based on three SAGEConv layers (Equations (1) to (3)). The hidden dimension of each layer has a fixed size of d, and a ReLU non-linearity is used for the first two layers. The result is a graph where each node is represented by an d-dimensional embedding. We set $d = 16$.

Second, the link prediction is carried out by a function $f_2 : \mathbb{R}^d \times \mathbb{R}^d \rightarrow [0, 1]$ which computes a value for a pair of vertices $(u \rightarrow v)$. This value corresponds to the probability of this link being part of a good solution. The function is parametrized as a simple 2-layer fully-connected neural network with a ReLU activation for the hidden layer. The hidden dimension also has a size of 16. A sigmoidal transformation is finally applied to the output of the last layer to obtain a value between 0 and 1 (i.e., a probability).

Both networks are trained together with standard backpropagation using Adam optimizer [18]. Once trained, the model is used to predict new arcs from a precedence graph of an RCPSP instance. A confidence threshold $\theta \in [0, 1]$ is then used to determine whether a new precedence should be added. Let $\hat{y}^{(u\rightarrow v)}$ be the prediction obtained for a pair $(u \rightarrow v)$ that is not neutral nor avoided. For each pair we consider the arc to be positive if the prediction is greater than the threshold (i.e., $\hat{y}^{(u\rightarrow v)} \geq \theta$).

3.3 Step 3: Solving the RCPSP with New Precedences

We propose two uses of the learned preferences: (1) as new constraints in the CP model, and (2) as an information for directing the search.

First, each learned precedence (between task i and j) can be directly integrated into the model as an additional constraint ($e_i \leq s_j$). This practice is ubiquitous in constraint programming. The expected goal is to reduce the search space. It relates to streamlined constraint reasoning [14] where a partition of the search space is done using additional cutting constraints. Despite the simplicity of this approach, it comes with the critical drawback that we have no mechanism to recover from prediction errors, as adding incorrect precedence as constraints may prune the optimal solution. Consequently, the solution obtained with this first option corresponds only to an upper bound, and we lose the ability to obtain an optimality proof.

A second option is to use the new precedences to direct the search. We propose to do it by introducing an easy *ad-hoc* problem dedicated to finding an appropriate task ordering (i.e. a topological order on the precedence graph). The problem is defined as follows. A variable o_i is defined for each task i, with a domain ranging from 0 to $n - 1$. For each precedence from the instance (i.e., the neutral arcs) and each precedence learned, a constraint enforcing the precedence is added ($o_i < o_j$ if i precedes j). All the variables are also subject to an `allDifferent` constraint. Solving this problem gives an ordering satisfying all the precedences learned. By integrating randomness within the search heuristic, bias toward some solutions can be removed [33]. This ordering can then be used as a static variable ordering heuristic to solve the instance. As no additional constraint is added to the initial model, there is no risk to prune the optimal solution, and we are still able to prove optimality.

However, both options are subject to a critical concern. They are consistent only if there is no cycle obtained from the learned precedences (i.e., $a < b$, $b < c$, and $c < a$). In such a situation, the model obtained (either with additional constraints or with the ad-hoc problem) is unsatisfiable. We addressed this difficulty by adding each precedence in a graph structure maintaining the transitive closure in polynomial time. Candidate precedences are tested one by one, in order of decreasing score, and added only if they do not create a cycle. This ensures that a solution is always possible whether we add the learned precedences to the model or we use them to construct a topological order.

4 Experiments

For our experiments, we used the instances available on the PSPLIB website[2]. The best bound found so far by the community (referred by *UB* in our figures[3]) can be found on the website. PSPLIB is composed of four sizes of instances (30, 60, 90, or 120 tasks). For each size, the instances were generated by varying a given number of parameters [19] (min/max durations of the tasks, min/max number of successors, etc.). For a given set of settings, 10 instances were generated by the benchmark authors. It is thus composed of four sizes times 48 (60 for the 120 tasks) combinations of values for the parameters times 10 generations per set of parameters, yielding a total of 2040 instances. We split the instances as follows:

1. The SEEN set, composed of all but 5 of the 48 (60 for the 120 tasks) series for each of the four sizes. Among these, we select 8 instances among the 10 composing each series (1472 instances) in total. The positive and negative arcs of the best-so-far solutions of these instances are split by a k-cross-validation to train the GNN and the MLP.
2. The UNSEEN sent, composed of the remaining 2 among the 10 instances from the series selected for the SEEN set. They form the validation set as they are similar to the ones seen by the learning process but still unseen.
3. The UNKNOWN set, composed of the 5 remaining series of 10 instances for each of the four sizes (200 instances in total). They constitute the generalization set as the learning process has not seen them and has seen no other instances generated with the same set of parameters.

In some cases, we analyzed the results by size. It is then indicated within the name, i.e. UNSEEN$_{J90}$ is the instances of 90 tasks within the UNSEEN group or UNSEEN$_{\leq J60}$ for the instances of 60 tasks or less. We note that the results we report always use the same splits for comparison purposes, e.g. if a specific instance belongs to UNSEEN$_{J90}$, it is included in UNSEEN$_{J90}$ in every experiment. We also used the same split for each solution in the dataset, e.g. an instance solved with a specific timeout and options will always have the same k folds for the cross-validation.

Our neural architecture is implemented in Python using *Deep Graph Library* [37] and *Pytorch* [25]. Our experiments were run on a computing cluster equipped with a *AMD Rome 7532* CPU. For reproducibility purposes, the environment, the splits, the models, and the outputs of the runs have been stored and added to a publicly available repository[4].

4.1 Baseline CP Model for the RCPSP and Training Set

Our base CP model to solve the RCPSP is the following. For each task, the start time of task i is represented by variable[5] s_i, whose domain spans from 0 to *horizon*[6]. For each resource, we add a `cumulative` constraint. In addition, a precedence constraint ($s_i + p_i \leq s_j$

[2] `https://www.om-db.wi.tum.de/psplib/`
[3] as of March 2024
[4] `https://github.com/363734/LearningPrecRCPSP`
[5] The PSPlib files define two additional dummy tasks: 0 and $n + 1$. We do not define variables for them.
[6] We used the horizon defined within the PSPlib files.

if task i precedes task j) is added to the model for each precedence. The CP solver used is Chuffed (branch *develop*, commit *b2152f3*). The `cumulative` constraint used is the global one implemented within chuffed with the parameters `tt_filt_on`, `ttef_check_on` and `ttef_filt_on`. These parameters activate the timetabling filtering algorithm, the time-table-edge-finder check, and the time-table-edge-finder filtering algorithm [35]. The search heuristic used is *min-min-value* which corresponds to selecting the variable with the smallest minimal value and selecting its minimal value first (i.e., selecting the task that could be scheduled the earliest). We also consider two options influencing the search heuristic: `sbps` and `vsids`.

Using this model and options we generated, for each instance of PSPLIB, the best-so-far (*bsf*) solution after several timeouts (1 second, 1 minute, 10 minutes, and 1 hour) to be able to compare the influence of the quality of the dataset over the whole process. As expected, for a given fixed timeout, the solver manages to solve smaller instances to optimality. For a timeout of 1 second, the simple model (i.e., without `sbps`/`vsids`) manages to get the best *bsf* in general. The trend reverses when increasing the timeout to 10 minutes or more, where the model with `sbps` and `vsids` leads to better *bsf* solutions and more optimal solutions proven. This difference is due to the overhead of the `sbps` and `vsids` techniques. The `sbps` option alone allows to reach good solutions rapidly but has difficulties to prove optimality (`chuffed` is even warning the user when using `sbps` that it must be used with an activity-based search to optimize its efficiency). Using both options is also better at closing instances when a bigger timeout is allowed. In the rest of our experiments, we will focus on the solutions from the model with both or no options used. These best-so-far solutions are used both as training data and for comparing the methods.

4.2 Performance of the Training

We tried multiple configurations to train our model.

Training with one best-so-far per instance

Concerning the training set with one best-so-far solution per instance, we have: the timeout used to generate the dataset (1 second, 1 minute, 10 minutes or 1 hour), the options chosen (with or without `sbps`/`vsids`), and the subset of graphs chosen for training (SEEN$_{\leq J30}$, SEEN$_{\leq J60}$, SEEN$_{\leq J90}$ or SEEN$_{\leq J120}$). The learning rate is also tuned to 10^{-2}, 10^{-3} or 10^{-4}. The training is carried out for $1,000$ epochs. We performed a k-cross-validation with $k = 10$. For each of the k runs, the one with the best loss is selected. The final model is the one with the best loss among the k runs. Training takes from around $k \times 10$ min (smallest training set) to $k\times$ 1-2 hours (bigger ones). The evaluation of our training is done using the following metrics.

- Precision (*prec*): fraction of predicted positive arcs that are truly positive;
- Recall (*rec*): percentage of positive arcs correctly predicted as positive;
- True negative (*tn*): percentage of negative arcs correctly predicted negative;
- F1-score (*f1*): harmonic mean of the precision and recall.

As a general observation, there are generally more negative arcs than positive ones. Discarding some negative ones (at random) to reach equality between the two sets improves the precision of our method by a few percent (up to $3-4\%$). This technique is commonly referred to as *under-sampling*.

A preliminary analysis of the learning curves and evolution of the metrics on the training set and testing set along the learning allowed us to discard some parameters. The learning rate of 10^{-4} led to a too-slow learning and bad metrics value even after 1000 epochs compared

Table 1 Validation metrics of the two chosen models. Learning rate 10^{-2}, training set SEEN$_{\leq J120}$ generated with a timeout of 1 hour, without Sbps/Vsids for model A, with Sbps/Vsids for model B.

	Model A (“*without*”)				Model B (“*with*”)			
	f1	*prec*	*rec*	*tn*	*f1*	*prec*	*rec*	*tn*
SEEN$_{J120}$	0.79	0.89	0.71	0.91	0.71	0.82	0.62	0.87
UNSEEN$_{J120}$	0.78	0.89	0.70	0.92	0.71	0.83	0.62	0.87
UNKNOWN$_{J120}$	0.80	0.90	0.72	0.92	0.72	0.84	0.64	0.87
ALL$_{J120}$	0.79	0.89	0.71	0.91	0.71	0.83	0.62	0.87

Table 2 Validation metrics of models trained on smaller instances. Same training characteristic as Model B, except training set (SEEN$_{\leq J60}$ and SEEN$_{\leq J90}$). (to be compared with second column of Tab. 1).

	Model B-SEEN$_{\leq J60}$				Model B-SEEN$_{\leq J90}$			
	f1	*prec*	*rec*	*tn*	*f1*	*prec*	*rec*	*tn*
SEEN$_{J120}$	0.72	0.82	0.64	0.86	0.71	0.83	0.62	0.87
UNSEEN$_{J120}$	0.72	0.83	0.64	0.87	0.71	0.83	0.62	0.88
UNKNOWN$_{J120}$	0.73	0.83	0.65	0.87	0.73	0.83	0.64	0.87
ALL$_{J120}$	0.72	0.83	0.64	0.87	0.71	0.83	0.62	0.87

to the others. The learning rate of 10^{-2} and 10^{-3} led to very similar results. While the learning curves of 10^{-2} oscillated more, the validation metrics were generally slightly better (1% increase). The datasets generated with Sbps/Vsids require fewer epochs to start stabilizing (i.e., having a loss close to the best loss among all the epochs) compared to the models generated on datasets without it.

Table 1 displays the result of the validation of the two most promising models. The table is organized into four rows for the evaluation of each subset of graphs (SEEN, UNSEEN, UNKNOWN and ALL) and each main column corresponds to one model. The same learning rate (10^{-2}), the same timeout (1h) for the dataset, and the same dataset (SEEN$_{\leq J120}$) are used in both models, the only distinction being the model used to create the dataset (without `sbps/vsids` for Model A and with for Model B).

In this table, we can first see that our model provides good precision but a weaker recall. In other words, the model does not make many false positives but more false negatives. This is a good prospect for our application as only predicted positive arcs lead to an impact on the model. A second observation is that the models trained with the dataset generated using the options `sbps/vsids` are globally worse than their counterparts. This is probably due to the quality of the solution. As our problem is $\mathcal{NP}$-hard [1], it is expected that learning true optimal solutions is difficult. When examining the validation results of the other models (the ones trained on benchmarks with a smaller timeout to generate the instances), we could observe the same trend as during learning, where using the 1-hour benchmarks provides better results.

We also look at the generalization capabilities of our architecture. Table 2 displays the results of two models against Model B, trained with the same characteristics as Model B, except for the instances size of the training sets, either only using the SEEN$_{\leq J60}$ or the SEEN$_{\leq J90}$. The metrics are computed for all sub-groups of different sizes. We can see that the models have good generalization capacities and have similar metrics to the one trained on SEEN$_{\leq J120}$.

Table 3 Validation metrics of Model C-SEEN$_{\leq J60}$, trained on multi-solution aggregate instances. Same training characteristic as Model C-SEEN$_{\leq J60}$, except the training set is not composed of one solution per instance, but, for each instance, of the precedences present in 70% of at most 100 solutions generated using an additional 10 minutes timeout.

	Model C-SEEN$_{\leq J60}$			
	f1	prec	rec	tn
ALL$_{J30}$	0.67	0.80	0.57	0.86
ALL$_{J60}$	0.68	0.79	0.59	0.84
ALL$_{J90}$	0.67	0.78	0.59	0.83
ALL$_{J120}$	0.72	0.83	0.64	0.87

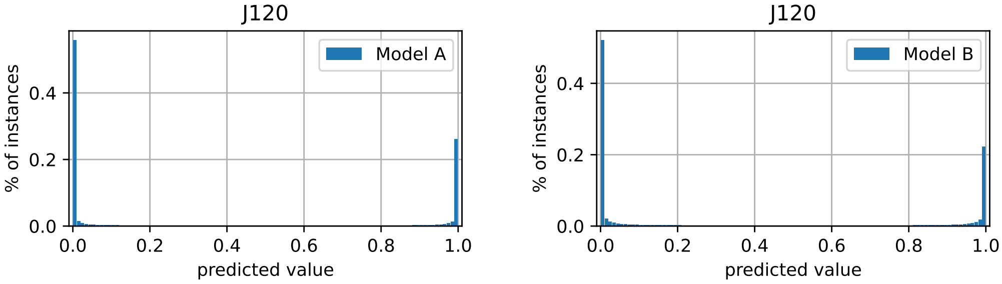
One can also notice a large gap between the metrics of smaller size compared to the j120 ones. This gap is not only present for these models with these characteristics but also present for all models trained with the "*with spbs/vsids*" training set but not in the one without. We speculate it is due to the accuracy of the solutions in the training set. In the "*with spbs/vsids*" benchmark, a majority of the solutions of instances with up to 90 tasks are optimal ones, while on the 120-task instances, it is no longer the case. Precedences within optimal solutions seem harder to predict, as the results of Table 1 were already showing.

Training with an aggregate of multiple best-so-far per instance

A natural question is to ask if choosing only one solution per instance impacts the prediction as often, there exist multiple optimal solutions for such problems. In general, there can exist multiple optimal solutions when not all the paths in the precedence graphs are critical (i.e., paths where each task starts exactly when the one preceding it ends). If the resources allow it, starting these tasks a little bit later can lead to another optimal solution. For some of these solutions, this does not change the precedences, but in some cases, it could slightly modify some of them.

To test whether our predictions had a bias while trained on a single solution for each instance, we generated, for the J30 and J60 instances, up to 100 optimal solutions (we gave an additional time out of 10 minutes to generate up to 100 solutions with the same best so far optimal value as the one found after 1 hour). From these 100 solutions, we keep only the precedences present in a majority (70%) of the solutions and trained on this subset. We thus have precedences most likely present in many solutions. We choose to focus on the smaller instances as they have shown to generalize well in training (Tab. 2). Also, as for most of them, the initial timeout of 1 hour reached the optimal solution. This training set is composed of real optimal solutions and not best-so-far solutions. Furthermore, as they are the easiest instances to solve, the 10-minute timeout allowed for the computation of 100 solutions was enough for a majority of them.

We trained a Model C-SEEN$_{\leq J60}$ on this benchmark, using the same configuration as Model B and Model B-SEEN$_{\leq J60}$ (learning rate of 0.001, 10-fold cross-validation, under-sampling and training set generated using SBPS/VSIDS active). The validation metrics (Tab. 3) are very similar to the ones present in Tab. 2. From a training point of view, there does not seem to be an impact.



(a) Edges predicted by Model A.

(b) Edges predicted by Model B.

Figure 3 Distribution of the learned edges.

Table 4 Number of improvements/deteriorations compared to the baselines (J120 instances only).

θ	Predictions used as constraints				Predictions used for ordering			
	to=1s	to=1m	to=10m	to=1h	to=1s	to=1m	to=10m	to=1h
	Predictions from Model A used on a model without SBPS/VSIDS							
0.5	0/600	0/600	0/600	0/600	24/391	4/373	6/360	8/349
0.55	0/600	0/600	0/600	0/600	28/387	4/367	2/358	7/348
0.75	0/599	0/600	0/599	1/598	26/394	4/370	4/362	3/357
0.95	1/585	0/584	2/584	3/582	22/405	5/372	9/357	8/350
0.99	4/559	6/557	5/554	4/553	27/399	6/371	5/359	7/350
	Predictions from Model B used on a model without SBPS/VSIDS							
0.5	0/600	0/600	0/599	0/599	31/392	7/371	7/365	7/355
0.55	0/600	0/599	1/599	0/598	21/404	6/367	6/363	9/360
0.75	1/597	0/597	2/594	3/594	31/388	6/372	7/358	7/352
0.95	5/554	4/556	2/553	3/550	23/404	9/365	11/357	9/349
0.99	21/487	5/497	7/495	6/493	25/393	7/373	5/362	8/356
	Predictions from Model A used a model with SBPS/VSIDS							
0.5	480/94	124/476	7/593	4/596	439/0	185/6	5/0	1/0
0.55	483/98	122/478	9/590	7/592	440/11	188/5	6/3	1/3
0.75	470/114	116/480	13/586	12/587	431/3	186/5	7/4	2/5
0.95	416/163	106/474	17/564	24/561	413/50	166/50	6/20	2/12
0.99	411/163	108/454	24/533	28/526	413/45	172/39	6/25	1/13
	Predictions from Model B used a model with SBPS/VSIDS							
0.5	475/110	116/484	10/590	8/590	445/0	186/4	5/1	2/2
0.55	447/127	116/483	11/588	10/590	421/25	176/19	7/7	2/4
0.75	457/113	114/481	15/580	14/577	454/0	191/1	7/1	2/2
0.95	413/159	111/449	26/525	21/532	433/26	179/27	6/14	2/8
0.99	419/152	125/386	26/469	34/459	433/26	179/25	8/17	2/8

4.3 Performance for Solving the RCPSPs

We decided to focus the presentation of our results on the ALL_{J120} as it contains the most open instances and thus the most interesting. We analyzed the results split among the SEEN_{J120}, UNSEEN_{J120}, and UNKNOWN_{J120} and did not find a significant difference between them, allowing us to confirm no sign of overfitting, as hinted by Tab. 1, hence why we present the results on ALL_{J120} only.

Distribution of predictions

We analyzed first the distribution of the prediction done among the edges of the J120 case in Fig. 3, for both Model A (Fig. 3a) and Model B (Fig. 3b). We can see that our process is confident on the outcome for most of the edges. Normally, we select as a positive prediction

Table 5 Generalization ability (to be compared with the last line of Tab. 4).

θ	Model B-SEEN$_{\leq J60}$							
	1s	1m	10m	1h	1s	1m	10m	1h
	Predictions used as constraints				Predictions used for ordering			
0.99	421/141	119/398	27/484	35/481	430/17	187/18	6/10	1/4

the precedence with a prediction score of at least 0.5. Given the distribution of the edges and the fact that the recall score is not that high, we would want to verify if increasing the threshold of the selected prediction could impact the final results. To this end, we decided to test multiple thresholds θ: 0.5, 0.55, 0.75, 0.95, and 0.99.

Comparison to Baseline

We run Model A and Model B to predict the edges of the full J120 benchmark. These predictions are then filtered given the threshold and then used either as *additional constraints* or to create an *ordering-based heuristic*, both to be used with the model with the SBPS/VSIDS options or without the options (Tab. 4). The tables gather, for each threshold, each timeout, each usage, and each generative model, the number of instances among the 600 where there is an improvement/deterioration of the bound compared to the baseline (i.e., the model with the same options used and the same timeout but without the learned approach).

Multiple observations can be made by comparing these numbers. First, using our ordering-based technique without a SBPS/VSIDS heuristics does not provide good results. This can be explained by the fact it has already been observed that static ordering performs generally slower than dynamic ones [27], allowing an easier deterioration of the bound. Another observation is that using precedence as an additional constraint leads to many deterioration. This is a logical consequence of the fact that adding new constraints creates a restriction and thus, if one of these constraints is wrong, removes the optimum. We can see that increasing the threshold mitigates this effect, as expected. Solutions based on the use of prediction from Model A are also generally a bit less good than when using Model B. A model trained on a more qualitative training set is thus preferable.

Interestingly, we can also notice that with shorter timeouts, using the ordering-based method manages to improve more bounds, as it can guide the search towards better solutions first. The restriction-based one improves more bounds with bigger timeouts, as by reducing the search space, it can potentially reach part of the search space that was not reached before within the timeout. This, however, works only if the restriction does not select bad precedences, as shown by the important number of deteriorations.

Generalization Ability

Figure 5 compares the results of Model B against Model B-SEEN$_{\leq J60}$. Our validation results on this model show that it had similar accuracy to Model B. To confirm it, we made the predictions on all J120 instances and solved them using the model with SBPS/VSIDS. Our results show that it generalizes very well as the results are comparable to the ones of Model B with the same other parameters (last line of Tab. 4).

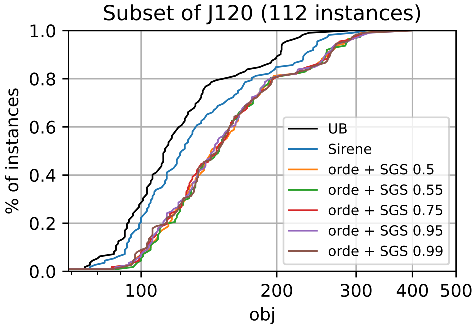


Figure 4 SGS-based solution.

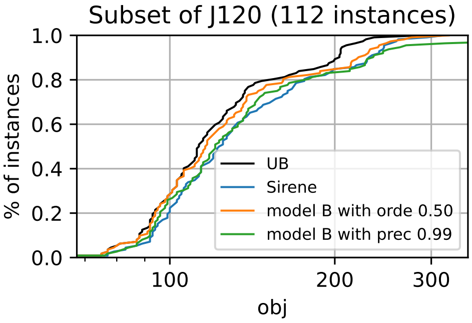


Figure 5 Comparison with SIRENE [31].

Schedule Generation Scheme (SGS)

A third way to use the prediction is to use an SGS, as done in [31]. Given an ordered set of tasks, the SGS greedily constructs a schedule, ensuring the resources and precedences constraints are respected. Fed with an ordering corresponding to the optimal solution, the SGS can reconstruct the optimal solution. Figure 4 displays, in a cactus plot, the bound of the solutions found after applying the SGS on our ordering produced by the prediction of Model B with the various thresholds. We compare to the SIRENE algorithm [31], using the same SGS. For the comparison, we used the results stored on their public repository. These bounds are available for 408 instances among the PSPLIB benchmark and were used to generate Figures 4 and 5 of their paper. Figure 4 focuses on the J120 among these (112 among the 600). The discrepancy between their method and our use of the SGS can be explained by the fact that their method works on a global view. By extracting an ordering from a close-to-a-solution, they guarantee that if some part of the schedule is right, it will stay the same after using the SGS. In our case, it only works if there are no errors within the predicted arcs. Observing that there is not much improvement in the solutions after using the SGS on our ordering with different thresholds confirms that there are predictive errors even when the learning process is sure of its prediction.

Comparison to Sirene

SIRENE [31] is the closest approach also using GNN found in the literature. While the two methods are very close to each other, they are in fact complementary. SIRENE focuses on a more global type of prediction (i.e., a potential solution to be corrected) while we focus on a more local prediction (i.e., new precedence between tasks to be used). Another conclusion we share is that improving the bounds obtained in our training set is difficult. In their case, they improve drastically the runtime while obtaining no bound improvement (Fig. 5) compared to their baseline. In our case, we managed to improve a few bounds (Tab. 4), and especially improve the quality of early solutions. It confirms the potential of GNNs to replicate statistical distribution (here the distribution of solutions provided in our training sets) but not to be able to solve the problem given and reason on it. Another limitation compared to SIRENE is that our current approach as our model is targeted to 4-resource problems. In our opinion, for instances with more features, we think an aggregation of multiple predictions for multiple sub-problems defined by selecting four features of the instance is a possible solution to get around this limitation. Fig. 5 shows a cactus plot, comparing the best-known upper bound (*UB*) to the makespan (from PSPLIB website), SIRENE, the predictions of Model

Table 6 Results when training on an aggregate of multiple solutions (to be compared with Fig.5).

θ	Model C-SEEN$_{\leq J60}$							
	1s	1m	10m	1h	1s	1m	10m	1h
0.99	Predictions used as constraints				Predictions used for ordering			
	436/108	154/332	46/427	48/424	451/0	193/1	7/2	0/2

B with $\theta = 0.5$ used for ordering on a model using Sbps/Vsids and time of 1 hour and the predictions of Model B with $\theta = 0.99$ used for additional constraint on a model using Sbps/Vsids and timeout of 1 hour.

Impact of multiplicity of optimal solution

Table 6 shows the result when Model C-SEEN$_{\leq J60}$ is used to predict precedences on the J120 instances. We also used them either in addition to the model either to create an ordering, both being used with the Sbps/Vsids generic heuristics when solving. When comparing to Fig. 5, one can see that using an aggregate of multiple solutions can help to improve the results. This comes from the fact that the precedences learned are the ones present in many optimal solution at once thus directing to any of these solutions the same way.

However, the drawback of this training set is that it requires multiple solutions. By using the generalization abilities of our model we were able to keep the additional computations cost relatively low. However, doing the same with bigger instances in the training set (by training on SEEN$_{\leq J120}$ like Model A and Model B) would be intractable as for many of the bigger instances, generating one good solution sometimes takes up to one hour, generating several of them would then be too tedious.

5 Conclusion and Perspective

This paper proposed a novel approach based on graph neural networks to predict new precedences for the resource-constrained project scheduling problem. The learned precedences can then be used either as additional constraints to get a stronger filtering or as a heuristic to drive the search. A high precision in the precedences learned has been obtained after the training. Our experiment on the PSPLIB benchmark confirms that, due to the $\mathcal{NP}$-hardness of the problem, a high recall is difficult to reach, but that we can nevertheless speed up the solving process when using a dynamic ordering. An improvement of our baseline (i.e., best-so-far after given time-out) has been observed in our experiment but remains difficult to achieve. The quality of the prediction depends on the quality of the training set. Using aggregates of multiple solutions allows learning of more crucial precedences. Our experiments shows also a good generalization as models trained on instances with less or equal than 60 tasks can achieved similar results on instances with 120 tasks as models trained on instances with less or equal than 120 tasks. Our method is solver agnostic and could even be combined with other metaheuristics such as a large neighborhood search.

One of our perspectives is to study whether the learned precedences are dependent on the training benchmark. Another is to reflect on how we can make predictions on instances with a different number of resources than trained for. Among our perspectives is also to apply this methodology to other variants of scheduling problems such as the RCPSP with time windows and the job shop scheduling problem. We also expect this could generalize to other combinatorial problems with an underlying graph structure such as job-shop scheduling problems.

References

1 Jacek Blazewicz, Jan Karel Lenstra, and AHG Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete applied mathematics*, 5(1):11–24, 1983.

2 Lei Cai, Jundong Li, Jie Wang, and Shuiwang Ji. Line graph neural networks for link prediction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):5103–5113, 2021.

3 Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Velickovic. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023.

4 Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre A Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

5 Félix Chalumeau, Ilan Coulon, Quentin Cappart, and Louis-Martin Rousseau. Seapearl: A constraint programming solver guided by reinforcement learning. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5–8, 2021, Proceedings 18*, pages 392–409. Springer, 2021.

6 Mark Cheung. *Geometric Deep Learning: Impact of Graph Structure on Graph Neural Networks*. PhD thesis, Carnegie Mellon University, 2022.

7 Geoffrey Chu and Peter J Stuckey. Learning value heuristics for constraint programming. In *Integration of AI and OR Techniques in Constraint Programming: 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings 12*, pages 108–123. Springer, 2015.

8 Bert De Reyck et al. A branch-and-bound procedure for the resource-constrained project scheduling problem with generalized precedence relations. *European Journal of Operational Research*, 111(1):152–174, 1998.

9 Emir Demirović, 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 *Principles and Practice of Constraint Programming: 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings 24*, pages 99–108. Springer, 2018.

10 Floris Doolaard and Neil Yorke-Smith. Online learning of variable ordering heuristics for constraint optimisation problems. *Annals of Mathematics and Artificial Intelligence*, pages 1–30, 2022.

11 Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A fair comparison of graph neural networks for graph classification. *arXiv preprint arXiv:1912.09893*, 2019.

12 Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 32, 2019.

13 Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

14 Carla Gomes and Meinolf Sellmann. Streamlined constraint reasoning. In *International Conference on Principles and Practice of Constraint Programming*, pages 274–289. Springer, 2004.

15 Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

16 Bo Han, Jiangchao Yao, Gang Niu, Mingyuan Zhou, Ivor Tsang, Ya Zhang, and Masashi Sugiyama. Masking: A new perspective of noisy supervision. *Advances in neural information processing systems*, 31, 2018.

17 Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017.

18 Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

19 Rainer Kolisch and Arno Sprecher. Psplib-a project scheduling problem library: Or software-orsep operations research software exchange program. *European journal of operational research*, 96(1):205–216, 1997.

20 Jan Karel Lenstra and AHG Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978.

21 Yujia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. Gated graph sequence neural networks. In *International Conference on Learning Representations*, 2016.

22 Linyuan Lü and Tao Zhou. Link prediction in complex networks: A survey. *Physica A: statistical mechanics and its applications*, 390(6):1150–1170, 2011.

23 Tom Marty, Tristan François, Pierre Tessier, Louis Gautier, Louis-Martin Rousseau, and Quentin Cappart. Learning a generic value-selection heuristic inside a constraint programming solver. In *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

24 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.

25 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

26 A Alan B Pritsker, Lawrence J Waiters, and Philip M Wolfe. Multiproject scheduling with limited resources: A zero-one programming approach. *Management science*, 16(1):93–108, 1969.

27 Patrick Prosser. The dynamics of dynamic variable ordering heuristics. In *Principles and Practice of Constraint Programming—CP98: 4th International Conference, CP98 Pisa, Italy, October 26–30, 1998 Proceedings 4*, pages 17–23. Springer, 1998.

28 Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

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.

30 Wen Song, Zhiguang Cao, Jie Zhang, Chi Xu, and Andrew Lim. Learning variable ordering heuristics for solving constraint satisfaction problems. *Engineering Applications of Artificial Intelligence*, 109:104603, 2022.

31 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 Sven Koenig, Roni Stern, and Mauro Vallati, editors, *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling, July 8-13, 2023, Prague, Czech Republic*, pages 623–633. AAAI Press, 2023. `doi:10.1609/ICAPS.V33I1.27244`.

32 Ronald van Driel, Emir Demirović, and Neil Yorke-Smith. Learning variable activity initialisation for lazy clause generation solvers. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5–8, 2021, Proceedings 18*, pages 62–71. Springer, 2021.

33 Mathieu Vavrille, Charlotte Truchet, and Charles Prud'homme. Solution sampling with random table constraints. *Constraints*, pages 1–33, 2022.

34 Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

35 Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 8th International Conference, CPAIOR 2011, Berlin, Germany, May 23-27, 2011. Proceedings 8*, pages 230–245. Springer, 2011.

36 Julien Vion and Sylvain Piechowiak. Une simple heuristique pour rapprocher dfs et lns pour les cop. *Proceedings of JFPC'17*, pages 39–45, 2017.

37 Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

38 Lingfei Wu, Peng Cui, Jian Pei, Liang Zhao, and Le Song. *Graph neural networks*. Springer, 2022.

39 Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Advances in neural information processing systems*, 31, 2018.

Inverting Step-Reduced SHA-1 and MD5 by Parameterized SAT Solvers

Oleg Zaikin
ISDCT SB RAS, Irkutsk, Russia

Abstract

MD5 and SHA-1 are fundamental cryptographic hash functions proposed in 1990s. Given a message of arbitrary finite size, MD5 produces a 128-bit hash in 64 steps, while SHA-1 produces a 160-bit hash in 80 steps. It is computationally infeasible to invert MD5 and SHA-1, i.e. to find a message given a hash. In 2012, 28-step MD5 and 23-step SHA-1 were inverted by CDCL solvers, yet no progress has been made since then. The present paper proposes to construct 31 intermediate inverse problems for any pair of MD5 or SHA-1 steps $(i, i+1)$, such that the first problem is very close to inverting i steps, while the 31st one is almost inverting $i+1$ steps. We constructed SAT encodings of intermediate problems for MD5 and SHA-1, and tuned a CDCL solver on the simplest of them. Then the tuned solver was used to design a parallel Cube-and-Conquer solver which for the first time inverted 29-step MD5 and 24-step SHA-1.

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

Keywords and phrases cryptographic hash function, MD5, SHA-1, preimage attack, SAT, Cube-and-Conquer

Digital Object Identifier 10.4230/LIPIcs.CP.2024.31

Funding This study was funded by the Ministry of Education of the Russian Federation, project No. 121041300065-9.

1 Introduction

A cryptographic hash function maps a message of arbitrary finite size to a hash of a fixed size [25]. A secure cryptographic hash function must be resistant to preimage attacks, i.e. it must be computationally infeasible to invert it by finding a message for a given hash. MD5 [37] and SHA-1 [11] are among the most influential and widespread cryptographic hash functions. Given a message, MD5 produces a 128-bit hash. MD5's core component is a compression function that operates on a 128-bit internal state in 64 steps. On each step, the state is modified by mixing with one 32-bit message word. SHA-1 has a similar design, but it produces a 160-bit hash, while its compression function operates in 80 steps. Nowadays both MD5 and SHA-1 are used in practice, e.g. to verify the data integrity. Partially this is because they are still preimage resistant.

It is well known that a cryptographic hash function's resistance can be practically analyzed by algorithms for solving the Boolean satisfiability problem (SAT) [6]. Since it is infeasible to invert MD5 and SHA-1, their weakened versions are usually considered, where some last steps are omitted. In 2007, 26-step MD5 was inverted [10], while for 27- and 28-step MD5 it was done in 2012 [22]. In 2008 and 2012, 22- and 23-step SHA-1 were inverted, respectively [42, 22]. In all these cases it was done via Conflict-Driven Clause Learning [24] (CDCL) solvers. Since 2012, no further progress has been made towards inverting 29-step MD5 or 24-step SHA-1 because the corresponding computational problems are extremely hard. This paper aims to fill these gaps.

When the number of steps of a cryptographic hash function is reduced, the inverse problem can be further simplified by reducing the number of known hash bits [35, 4]. For example, in case of MD5, 29 steps and 64 known hash bits instead of 128 can be considered.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 31; pp. 31:1–31:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, even if this inverse problem is solved, it is not clear if any progress compared to inversion of 28-step MD5 has been made as a result. It is also possible to weaken an inverse problem by partially assigning some bits in a message [8]. However, again, it is unclear whether such weakening contributes to progress or not.

This paper proposes a new type of intermediate inverse problems. Consider an arbitrary cryptographic hash function such that its compression function is divided into steps and on each step a k-bit message word m is mixed with an internal state. Note that most existing cryptographic hash functions, including MD5 and SHA-1, match this condition. Consider a pair $(i, i+1)$ of steps. The idea is to construct $k-1$ intermediate inverse problems by simplifying step $i+1$, while the first i steps are not modified. In the first problem, in step $i+1$ the k-bit message word m is replaced by a k-bit word, where $k-1$ rightmost bits are 0s, while the remaining bit is equal to the leftmost bit in m. In the second problem, $k-2$ rightmost bits are 0s, while 2 remaining bits are equal to that in m and so on. Finally, in the $(k-1)$-th problem, only the rightmost bit in m is replaced by 0. As a result, for a state-of-the-art CDCL solver the first intermediate problem is slightly harder than inverting i steps, then the hardness gradually increases towards inverting $i+1$ steps.

We construct SAT encodings of 31 intermediate inverse problems between 28- and 29-step MD5 for one regular hash. Some of them are solved via the CDCL solver Kissat on a computer in reasonable time. Kissat's parameters are tuned on several simplest problems, and as a result more intermediate problems are solved. A parallel Cube-and-Conquer solver based on the same tuned Kissat inverts 29-step MD5. For SHA-1, its own tuning is performed, and as a result the parallel solver inverts 24 steps.

The paper is organized as follows. Preliminaries on MD5 and SHA-1 are given in Section 2. Intermediate inverse problems are proposed in Section 3. SAT encodings are presented in Section 4. Experiments on the default Kissat are discussed in Section 5. Section 6 presents an algorithm for tuning Kissat. Section 7 describes how Cube-and-Conquer based on the tuned Kissat inverted 24-step SHA-1 and 29-step MD5. Finally, related work is discussed and conclusions are drawn.

2 Preliminaries

This section gives preliminaries on the cryptographic hash functions MD5 and SHA-1.

Cryptographic Hash Function

A *cryptographic hash function* h maps a *message* of arbitrary finite size to a *hash* of finite size [25]. An obligatory property of any cryptographic hash function is that the mapping must be easy to compute, but hard to invert. Consider the following types of resistance.

1. *Collision resistance*: it is infeasible to find any two messages x and x' such that $x \neq x', h(x) = h(x')$.
2. *Preimage resistance*: for any given hash y, it is infeasible to find such a message x' that $h(x') = y$.
3. *Second-preimage resistance*: for any given message x, it is infeasible to find x' such that $x' \neq x, h(x) = h(x')$.

A secure cryptographic hash function must possess all three properties. There are two types of preimage attacks: (i) *practical preimage attack* implies solving an inverse problem, i.e. finding a preimage (message) for a certain hash; (ii) *theoretical preimage attack* is an algorithm for solving an inverse problem with lower complexity than brute force.

The main component of most cryptographic hash functions is a *compression function* that maps an input of fixed size to a shorter output of fixed size. In order to show that such a cryptographic hash function is not secure, it is sufficient to break the resistance of its compression function. In this paper we focus on studying the practical preimage resistance of compression functions of cryptographic hash functions MD5 and SHA-1.

MD5

The Message Digest 5 (MD5) cryptographic hash function was proposed in 1992 [37]. It is a more secure version of MD4 proposed in 1990 [36]. Given a message of arbitrary finite size, *padding* is applied to obtain a message that can be divided into 512-bit blocks. Then a 128-bit hash is produced in accordance with the Merkle-Damgard construction [26, 9], i.e. the compression function is iteratively applied to the blocks.

Given a 512-bit message block, MD5 compression function produces a 128-bit output. The function consists of four rounds, sixteen steps each, and operates by transforming data in four 32-bit registers A, B, C, D. For the first message block, the registers are initialized with the constants specified in the standard. Otherwise, the registers are initialized with an output produced at the previous iteration. The message block is divided into sixteen 32-bit words. In each step, three registers are updated by permuting the current values, while the remaining register is updated by mixing one message word, the current values of all four registers, an additive constant, and a result of the previous step. The mixing is partially done by a round-specific function, while additive constants are step-specific. As a result, all sixteen words take part in each round. When all steps are executed, the registers are incremented by the values they had after the initialization, and the output is produced as a concatenation of A, B, C, D.

Consider a pseudocode of an MD5 step in Algorithm 1. Here $g, 0 \leq g \leq 15$ is a message word index, $1 \leq i \leq 64$ is a step number, $\boxplus$ is addition modulo 2^{32}, and $\lll$ is circular left bit rotation.

Algorithm 1 The i-th step of MD5.

Input: Current registers' values A, B, C, D; step number i; message word index g; shift amount s; round function $Func$.

Output: Updated values A, B, C, D.

```
$temp \leftarrow Func(B, C, D) \boxplus A \boxplus K[i-1] \boxplus M[g]$
$A \leftarrow D$
$D \leftarrow C$
$C \leftarrow B$
$B \leftarrow B + (temp \lll s)$
```

Inputs for all 64 steps are specified in [37]. The round functions are as follows: $(x \wedge y) \vee (\neg x \wedge z)$; $(x \wedge z) \vee (y \wedge \neg z)$; $x \oplus y \oplus z$; $y \oplus (x \vee \neg z)$.

In 2004, the first MD5 collisions were published [44]. Regardless, it is still preimage resistant and second-preimage resistant.

SHA-1

The Secure Hash Algorithm 1 (SHA-1) cryptographic hash function was proposed in 1995 as another more secure extension of MD4 [11]. The main differences compared to MD5 are listed below.

1. A 160-bit hash is produced.
2. The compression function operates in 80 steps (4 rounds, 20 steps each) and returns a 160-bit output.
3. Five 32-bit internal registers A, B, C, D, E.
4. Additive constants are round-specific.
5. New round functions: $(x \wedge y) \vee (\neg x \wedge z)$; $x \oplus y \oplus z$; $(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$; $x \oplus y \oplus z$.
6. Instead of using parts of the message M directly in transformations, W is used such that $W[t] = M[t]$, $0 \le t \le 15$, and $W[t] = (W[t-3] \oplus W[t-8] \oplus W[t-14] \oplus W[t-16]) \lll 1$ if $16 \le t \le 79$.

Consider a pseudocode of a SHA-1 step in Algorithm 2.

Algorithm 2 The i-th step of SHA-1.

```
Input: Current registers' values $A, B, C, D, E$; step number $i$; round index $q$; round function $Func$.
Output: Updated values $A, B, C, D, E$.
$temp \leftarrow (A \lll 5) \boxplus Func(B, C, D) \boxplus E \boxplus K[q] \boxplus W[i-1]$
$E \leftarrow D$
$D \leftarrow C$
$C \leftarrow B \lll 30$
$B \leftarrow A$
$A \leftarrow temp$
```

In 2017, the first SHA-1 collisions were published [43], but it is still preimage resistant and second-preimage resistant.

3 Intermediate Inverse Problems for MD5 and SHA-1

This section first proposes a new type of intermediate inverse problems for cryptographic hash functions, then it describes how such problems can be constructed for MD5 and SHA-1.

Intermediate Inverse Problems

Consider an arbitrary cryptographic hash function h such that its compression function f is divided into r steps, and on each step an internal state is mixed with a k-bit word m, where m is either a message word (like in MD5) or a mix of messages' words (like in SHA-1). Besides MD5 and SHA-1, most existing cryptographic hash functions match this condition, e.g. MD4, RIPEMD-160, and SHA-256.

During preliminary experiments, for different triples (cryptographic hash function, SAT encoding, CDCL solver) we attempted to invert modified step-reduced functions, where the last step was weakened in different ways. It turned out that if m is not used, then the last step does not really make the inverse problem harder for the solver. Based on this observation, let us propose the following notation.

Consider i-step and $(i+1)$-step reduced versions of h, where $1 \le i \le r-1$. Construct $k-1$ *intermediate step-reduced cryptographic hash functions*, where the first i steps are used as usual, while step $i+1$ is weakened as follows. In the j-th intermediate function, $1 \le j \le k-1$, the word m in step $i+1$ is replaced by the word m_{weak} such that the rightmost $k-j$ bits in m_{weak} are equal to 0s, while the remaining j bits are equal to the leftmost j bits in m. Note that this is not the same as replacing m by m_{weak} in the whole h – if m is used in several steps, then the proposed action affects only step $i + 1$. In the *j-th intermediate inverse problem* it is needed to find a message given a hash produced by the j-th intermediate step-reduced hash function.

The intention behind this approach is that for a state-of-the-art CDCL solver the first intermediate problem will be slightly harder than the inversion of i steps, then the hardness gradually increases towards the inversion of $i + 1$ steps.

Note that if for the j-th intermediate inverse problem between steps i and $i+1$ a preimage is found for a hash, then with probability $\frac{1}{2^{k-j}}$ the preimage inverts the same hash produced by unmodified $i + 1$ steps because $k - j$ bits of the corresponding word m can be considered as random bits that can coincide with $k - j$ 0s with the mentioned probability. This is the first take away of the proposed approach – *sometimes it is sufficient to solve an intermediate inverse problem to invert $i + 1$ steps.*

Intermediate Inverse Problems for MD5

During preliminary experiments, we tried to weaken step $i + 1$ of the MD5 compression function for different values of i by deleting $Func(B, C, D)$ from the sum in the first line of Algorithm 1. However, it did not lead to simpler inverse problems from a state-of-the-art CDCL SAT solver point of view. It was also tried to delete other addends in the first line's sum, to delete the addend B from the fifth line's sum, and to omit shifting in the fifth line. It turned out, that the only action that significantly decreases the hardness is deleting the addend $M[g]$ in the first line. Moreover, if $M[g]$ is deleted, then inverting $i + 1$ steps is almost similar to inverting i steps for a CDCL solver.

A pseudocode of an MD5 step weakened according to the proposed idea is presented in Algorithm 3. Here $weakM$ is a 32-bit word, and as a result of two shifts in the first line its rightmost $32 - j$ bits are equal to 0, while the remaining j bits are equal to the leftmost j bits in $M[g]$. Then $weakM$ is used instead of $M[g]$. Therefore, to form 31 intermediate inverse problems with increasing hardness, j should be varied from 1 to 31.

Algorithm 3 The $(i + 1)$-th weakened step of MD5.

Input: Current registers' values A, B, C, D; step number i; message word index g; shift amount s; round function $Func$; intermediate hash function number j.
Output: Updated values A, B, C, D.

```
1: $weakM \leftarrow (M[g] \gg (32 - j)) \ll (32 - j)$
2: $temp \leftarrow Func(B, C, D) \boxplus A \boxplus K[i] \boxplus weakM$
3: $A \leftarrow D$
4: $D \leftarrow C$
5: $C \leftarrow B$
6: $B \leftarrow B + (temp \lll s)$
```

In the rest of the paper, the j-th intermediate hash function between MD5 steps i and $i + 1$, $1 \leq i \leq 63$, is called *(i j/32)-step MD5*. Note, that according to this notation $j = 32$ corresponds to $(i + 1)$-step MD5.

Intermediate Inverse Problems for SHA-1

Consider a pair $(i, i + 1), 1 \leq i \leq 79$, of SHA-1 steps. Similar to the previous subsection, the idea is to weaken step $i + 1$, while the first i steps are used as usual. During preliminary experiments, it was tried to omit operations in Algorithm 2 for different values of i. The picture is the same as for MD5 – when the usage of W is omitted, from the SAT solving point of view the inverse problem becomes the same as the inverse of the first i steps. Yet omitting any other operation, including the usage of the round function, does not make the problem easier. Based on these results, intermediate inverse problems are formed in the same way as for MD5, see Algorithm 4.

Algorithm 4 The $(i+1)$-th weakened step of SHA-1.

Input: Current registers' values A, B, C, D, E; step number i; round index q; round function $Func$; intermediate hash function number j.
Output: Updated values A, B, C, D, E.

```
$weakW \leftarrow (W[i] \gg (32-j)) \ll (32-j)$
$temp \leftarrow (A \lll 5) \boxplus Func(B,C,D) \boxplus E \boxplus K[q] \boxplus weakW$
$E \leftarrow D$
$D \leftarrow C$
$C \leftarrow B \lll 30$
$B \leftarrow A$
$A \leftarrow temp$
```

4 SAT Encoding

In this study, inverse problems for step-reduced SHA-1 and MD5 compression functions are considered as was done earlier in [22, 35]. This section describes the corresponding SAT encodings.

SHA-1 Encoding

Several SAT encodings of the SHA-1 compression function have been proposed so far [42, 35, 22, 28]. However, it is well known that at the moment the best one is Vegard Nossum's encoding [35]. Compared to the competitors, it produces more compact CNFs which are easier for CDCL solvers [35]. That is why Nossum's encoding has been used in many further studies, e.g. [34, 43].

Recall that in each SHA-1 step 5-ary addition is performed, see Section 2. In the Nossum's encoding, the column addition algorithm is applied, and each column sum is expressed via a pseudo-Boolean constraint which in turn is encoded in the clausal form using the ESPRESSO logic minimizer. Vegard Nossum implemented his encoding in the form of the program SHA1-SAT, which is available online[1]. Given the number of SHA-1 steps and a hash, the program produces the corresponding inverse problem in the CNF form.

For the present study, SHA1-SAT was extended to maintain intermediate inverse problems proposed in Section 3. Assume that $(i+1)$-step SHA-1 is considered, where a combination of message words, the word $W[i]$, is used as an addend, see Section 3, and the corresponding CNF is produced by SHA1-SAT. To encode the j-th intermediate inverse problem between steps i and $i+1$, the following additional actions are performed:

1. An additional 32-bit word $weakW$ is introduced in the form of 32 Boolean variables.
2. The rightmost $32-j$ bits of $weakW$ are assigned to 0 via adding unit clauses to the CNF.

[1] https://github.com/vegard/sha1-sat

3. The equality conditions for the leftmost j bits of $weakW$ and the corresponding j bits of $W[i]$ are added in the form of $j \times 2$ binary clauses.
4. 32 Boolean variables of $weakW$ are used instead of $W[i]$'s 32 variables in the clauses where the addition is encoded in step $i + 1$.

By using the modified program, CNFs for intermediate inverse problems were constructed between the following pairs of SHA-1 steps: $(21, 22)$, $(22, 23)$, and $(23, 24)$ for 10 hashes: 160 0s, 160 1s, and 8 random ones. Also, for 21, 22, 23, and 24 steps, CNFs for standard inverse problems were generated for the same 10 hashes. In Table 1, the characteristics of CNFs for 23, 23 16/32, and 24 steps are shown.

Table 1 Characteristics of CNFs that encode inverse problems for the main considered step-reduced versions of SHA-1 and MD5.

Hash	Steps	Variables	Clauses	Literals
SHA-1	23	4 288	132 672	873 727
SHA-1	23 16/32	4 480	138 812	913 700
SHA-1	24	4 448	138 764	913 620
MD5	28	7 168	92 520	506 320
MD5	28 16/32	7 424	95 818	524 312
MD5	29	7 392	95 770	524 232

MD5 Encoding

Recently, several SAT encodings of the MD5 compression function have been proposed. The encodings from [10, 22] are not available. The encodings from [40, 46] are available, but we decided to extend SHA1-SAT to support MD5.

In MD5, in each step 4-ary addition is performed, so the corresponding pseudo-Boolean constraint were constructed and encoded in the clausal form using ESPRESSO. As for the round functions (see Section 2), clausal forms of three of them were taken from the SHA-1 encoding, while for the remaining one the clausal form was constructed manually.

CNFs for intermediate inverse problems were constructed between steps $(27, 28)$ and $(28, 29)$ for the same 10 hashes as for SHA-1. In Table 1, the characteristics of main CNFs are shown.

5 Solving Intermediate Inverse Problems by Kissat

First, we decided to study how a state-of-the-art sequential CDCL solver behaves on intermediate inverse problems in case of MD5 and SHA-1. KISSAT of version 3.0 [5] was chosen because this solver and its modifications have showed excellent results in the last three SAT Competitions. Of course, other CDCL solver can be also tried, for example, those which are oriented on cryptanalysis [41, 32, 21]. All described experiments were performed on a PC equipped with the 16-core CPU AMD 3950X and 64 Gb of RAM.

It turned out that inverse problems for 22-step SHA-1 are simple for KISSAT, while for 23 steps the problems are quite hard, but still can be solved in reasonable time. That is why 33 inverse problems were considered: the one for 22 steps, 31 intermediate problems between steps 22 and 23, and the one for 23 steps. As stated in Section 4, we generated 10 instances for each problem – 1 for 160 1s hash, 1 for 160 0s hash, and 8 for random hashes, i.e. 330 instances in total. With the time limit of 24 hours, 303 SHA-1 instances out of 330 were

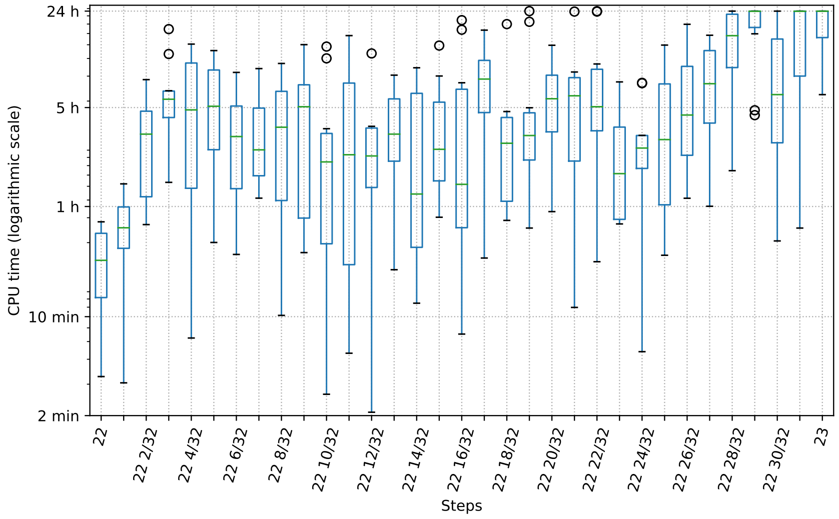


Figure 1 Boxplots of Kissat3 runtimes on intermediate inverse problems for SHA-1.

solved. Figure 1 shows the runtimes for SHA-1 in the logarithmic scale, where the unsolved instances' values were set to 24 hours. In the figure, the five-number statistics is presented in the form of boxplots: outliers are plotted as small circles, whiskers correspond to the minimum and the maximum (excluding outliers), the median is plotted in green, while a box is plotted from the first quartile to the third quartile.

As for MD5, 330 instances between steps 27 and 28 were generated in the same way. With the time limit of 24 hours, all MD5 instances were solved, see Figure 2.

According to the results, in case of MD5 the hardness of the intermediate inverse problems grows almost linearly. In case of SHA-1 there is a clear leap between steps 22 1/32 and 22 2/32, then the hardness remains more or less the same, and finally the second leap happens between steps 22 27/32 and 22 28/32. Note, that 24 out of 27 unsolved instances were parts of the last 5 series, i.e. starting from 22 28/32 steps. Similar experiments were held between steps 21-22 of SHA-1 and 26-27 of MD5 and the patterns were the same there.

Based on these results, the second take away of the approach, proposed in Section 3, is formulated: *a runtime estimation for an unreachable step can be calculated by extrapolating runtimes of the previous step and (some) intermediate problems.*

As mentioned in Section 2, it is sufficient to invert any hash to break the preimage resistance of a cryptographic hash function. However, in practice this should be a regular hash, say all 0s or all 1s, otherwise it may be hard to justify this choice and prove that it was not done for a random pair of input and output. In this paper, 128 1s and 160 1s are chosen for MD5 and SHA-1, respectively. Further they are called *1-hashes*. Instances of intermediate inverse problems between steps 28-29 of MD5 and steps 23-24 of SHA-1 were generated for 1-hashes. Recall that at most 28-step MD5 and 23-step SHA-1 are inverted in the literature. Out of 31 intermediate problems, 6 were solved by Kissat within the time limit of 24 hours in case of MD5. In particular, the intermediate problems with j of 1, 3, 5, 8, 9, and 10 were solved. Yet only one intermediate problem with $j = 20$ was solved for SHA-1. It means that (28 10/32)-step MD5 and (23 20/32)-step SHA-1 are inverted, and this is already a clear progress compared to the literature, but in the next sections we are going further.

The third take away is as follows: *by solving intermediate inverse problems, some progress can be achieved compared to the state of the art.*

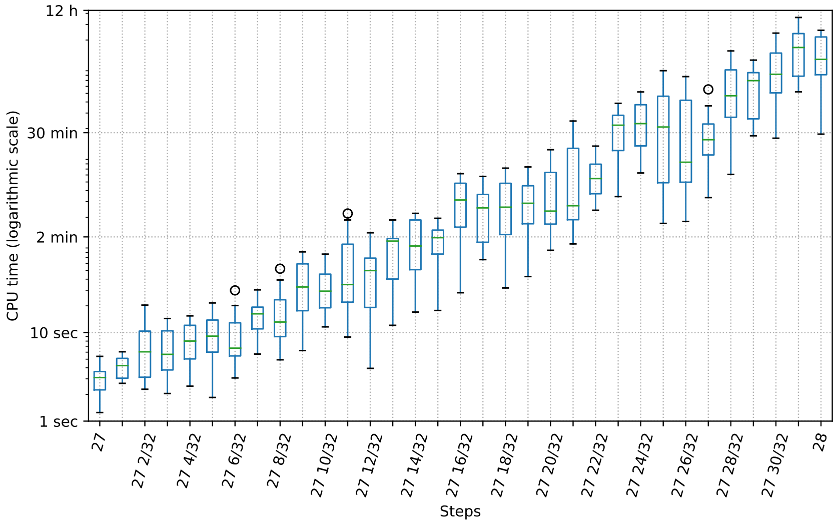


Figure 2 Boxplots of Kissat3 runtimes on intermediate inverse problems for MD5.

6 Parameterization Algorithm

A hypothesis behind the proposed method is that simple intermediate inverse problems can be leveraged to parameterize Kissat (or, in other words, tune its parameters' values) to solve hard intermediate problems faster or just solve within the time limit if it is not yet so. On the one hand, the tuning simplifies problems which are already simple, and it is not guaranteed that it will help solving hard problems. On the other hand, all corresponding CNFs have a very similar structure. We decided to test this hypothesis in practice.

Parameterization of SAT solvers is a well-developed field, see [18]. Formally, the general problem is to automatically identify a *parameter configuration* (a set of parameters' values) that maximizes the performance of a SAT solver across a set of instances [18]. Among the main parameterization algorithms for SAT are: ParamILS [20], SMAC [19], and GGA [2]. A recent implementation of GGA is PyDGGA [1] while that for SMAC is SMAC3 [23]. In this study, for this purpose an extension of a simple yet powerful (1+1) evolutionary optimization algorithm ((1+1)-EA [30]) is used to have a full control over the tuning process. Another reason is that evolutionary optimization algorithms have been successfully applied recently to speed up parallel SAT solving [39, 47].

In short, (1+1)-EA works as follows: a Boolean vector of size k is given, and a series of k independent Bernoulli trials [29] with the success probability $1/k$ is performed. If $i \in \{0, \dots, k-1\}$ corresponds to a successful trial, then the i-th value is flipped. The obtained vector is compared with the best one via an objective function, then a new vector is generated and so on. It is clear that only binary parameters can be tuned via (1+1)-EA.

We propose an integer extension of (1+1)-EA that is further called (1+1)-EA-Int. The pseudocode is presented in Algorithm 5. This algorithm operates with parameters of arbitrary finite sizes by applying the categorical distribution – a discrete probability distribution that describes the possible results of a random variable that can take on one of n possible categories, each with specified probability [31]. Assume that P is a set of parameters values. To choose a new value for the i-th parameter, the function $\mathit{categ\_dist}(x_i, P_i)$ is called, where x_i is the current value of the i-th parameter and $P_i, |P_i| = k$ is a set of possible values of the i-th parameter. In this function, a weights vector w of length k is formed. First, w_i is assigned to 0. Assume that $maxdist = max(i-1, k-i-2)$, then $w_{i-1} = w_{i+1} = 2^{maxdist}$, $w_{i-2} = w_{i+2} = 2^{maxdist-1}$ and so on. As a result, at least one of two elements w_0, w_{k-1} becomes 1. Finally, the probabilities are assigned to parameters values as follows: $(\frac{w_0}{\sum_{j=0}^{k-1} w_j}, \frac{w_1}{\sum_{j=0}^{k-1} w_j}, \ldots, \frac{w_{k-1}}{\sum_{j=0}^{k-1} w_j})$. The idea is that the closer a value to the current one, the higher the probability is, yet any value (except the current one) can be chosen with non-zero probability.

Algorithm 5 (1+1)-EA-Int.

Input: parameters values P, a start configuration x^0, an objective function f.
Output: the best found configuration x^{best} with the objective function's value f^{best}.

```
$n \leftarrow param\_num(x^0)$
$\langle x^{best}, f^{best} \rangle \leftarrow \langle x^0, f(x^0) \rangle$
$checked\_conf \leftarrow \{\}$
while $not\ termination\ criteria$ do
    $x^{cur} \leftarrow x^{best}$
    for $i \leftarrow 0$ to $n-1$ do
        with probability $\frac{1}{n}$ $x_i^{cur} \leftarrow categ\_dist(x_i^{cur}, P_i)$.
    if $x^{cur}$ in $checked\_conf$ then
        continue
    $f^{cur} \leftarrow f(x^{cur})$
    $checked\_conf.add(x^{cur})$
    if $f^{cur} < f^{best}$ then
        $\langle x^{best}, f^{best} \rangle \leftarrow \langle x^{cur}, f^{cur} \rangle$
return $\langle x^{best}, f^{best} \rangle$
```

Consider an example. Assume that a parameter has possible values $1, 2, 5, 10, 25$, while its current value is 10. Since $maxdist = max(3-1, 5-3-2) = 2$, it follows that $w = (1, 2, 4, 0, 4)$, and the probabilities are $(\frac{1}{11}, \frac{2}{11}, \frac{4}{11}, 0, \frac{4}{11})$.

Note, that additionally in Algorithm 5 all checked configurations are remembered to avoid redundant calculations since the objective function is costly.

In all experiments, given a sample of CNFs and a parameter configuration, the objective function f is the sum of Kissat's runtimes on the CNFs, when the configuration is applied to the solver. The goal is to minimize the function on the space of parameter configurations.

7 Inverting by Parameterized Solvers

In this section, the parameterization algorithm proposed in Section 6 is applied to tune Kissat on simple intermediate inverse problems to invert 24-step SHA-1 and 29-step MD5.

Experimental Setup

We implemented the parameterization algorithm from Section 6 in Python, where the objective function is calculated in parallel – 1 configuration per CPU core. The implementation is available online[2]. The extended version of sha1-sat (see Section 4) and all generated CNFs

[2] https://github.com/olegzaikin/paramsat

are available online as well[3]. The same PC as in Section 5 was used in the experiments. The termination criteria were as follows: at most 24 (or 72 in some cases) hours and 1 000 calculations of the objective function.

There are 90 parameters in KISSAT 3.0, yet 50 of them were excluded from the consideration since in preliminary experiments they did not affect the performance. The remaining 40 parameters are presented in Table 2. All these parameters are integer, and their ranges are specified in the solver's documentation [5]. As a start configuration x^0 in Algorithm 5 the default KISSAT's configuration was leveraged.

Table 2 Varied parameters of KISSAT.

backbone	eliminateclslim	reluctantlim	sweepdepth
backbonerounds	eliminateocclim	restartint	sweepfliprounds
bumpreasonslimit	eliminaterounds	restartmargin	sweepmaxclauses
bumpreasonsrate	emafast	shrink	sweepmaxdepth
chronolevels	emaslow	stable	sweepvars
compactlim	mineffort	substituteeffort	target
decay	minimizedepth	substituterounds	tier1
definitioncores	modeinit	subsumeclslim	tier2
definitionticks	reducefraction	subsumeocclim	vivifytier1
eliminatebound	reluctantint	sweepclauses	vivifytier2

Inverting 24-step SHA-1

In preliminary experiments it turned out that a training set must contain at least 10 CNFs to obtain a configuration that works better on harder intermediate inverse problems. For SHA-1, the training set consisted of 16 CNFs: the last 15 intermediate inverse problems between steps 21-22 and inverting 22-step SHA-1, all for 1-hash. These CNFs were chosen because the objective function's value on them is reasonable: 1 hour 58 minutes on 1 CPU core. For comparison, on the first 16 intermediate inverse problems between steps 22-23 the corresponding value is 44 hours.

The parameterization algorithm was run three times with seeds 0, 1, and 2 and the time limit of 24 hours. It stopped after 11, 12, and 16 hours, respectively, because the limit on the objective function calculations (1 000) had been reached. The best result was found with seed of 1: the best configuration was updated 13 times, and on the final one the objective function's value was 22 minutes (compared to 1 hour 58 minutes on the default configuration). Table 3 contains those 12 parameters which changed their values in the found configuration.

The tuned and the default KISSAT3 were run with the time limit of 24 hours on 64 CNFs: 31 intermediate ones between steps 22-23, 1 for 23 steps, 31 intermediate ones between steps 23-24, and 1 for 24 steps. The results are presented as a cactus plot in Figure 3. The default version solved 30 intermediate problems between steps 22-23, while the tuned version solved all 31 of them. Both versions inverted 23 steps. Out of 31 intermediate problems between steps 23-24, the default version solved 1 problem, while the tuned one coped with 5 problems. All in all, the tuned version solved 5 more problems, and on most problems it was significantly faster.

[3] `https://github.com/olegzaikin/sha1-sat`

Table 3 The best KISSAT's configuration found for SHA-1.

Parameter	Default value	Found value
backbonerounds	100	10
definitionticks	1 000 000	100
eliminatebound	16	32
eliminateclslim	100	10
emafast	33	10
minimizedepth	1 000	100
restartmargin	10	20
stable	1	2
sweepfliprounds	1	5
sweepmaxclauses	4 096	2 147 483 647
sweepvars	128	64
vivifytier1	3	2

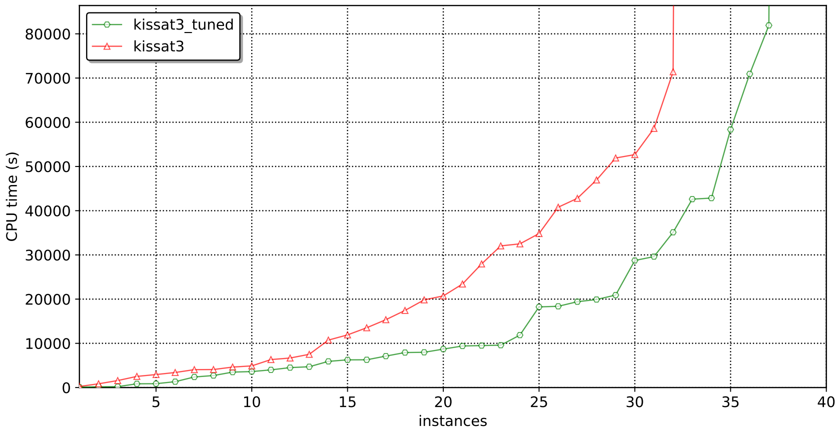


Figure 3 Comparison of the default KISSAT with its tuned version on intermediate inverse problems for steps 22-24 of SHA-1, 1-hash.

According to Figure 3, the hardness of the intermediate inverse problems increases quite smoothly in case of SHA-1, that is why it was decided not to run any additional tuning since 5 out of 31 intermediate inverse problems between steps 23-24 were already solved by the tuned solver. To invert 24-step SHA-1, experiments were run on the supercomputer "Akademik V.M. Matrosov"[4]. Each supercomputer's node is equipped with a 36-core CPU and 128 Gb RAM. At most 5 nodes (180 cores) were taken for one task.

To parallelize hard intermediate problems, the Cube-and-Conquer [13] approach was applied, where a given problem is split via lookahead into subproblems, which are solved by a CDCL solver. The lookahead solver MARCH_CU [14] was used to split the inverse problem for 24-step SHA-1 into 166 subproblems via the cutoff parameter *n=3467*. Then the same tuned KISSAT was run on the subproblems in the form of a 167-core task with the time limit of 3 days. For this purpose, the MPI program CONQUER_MPI was run[5], which used 1 core

[4] Irkutsk Supercomputer Center of SB RAS, `http://hpc.icc.ru`
[5] `https://github.com/olegzaikin/EnCnC`

for the master process and 166 cores for computing processes. Table 4 presents a solution that was found in 23 hours. The default Kissat was also run on the same subproblems on the supercomputer, yet no solution was found within 3 days.

Table 4 A preimage of 160 1s produced by 24-step SHA-1.

`0xa6c5c463`	`0x182655e0`	`0x2c5ba5f0`	`0xe0028033`
`0x8c3779b1`	`0x98635880`	`0xc5b822e`	`0x297efce7`
`0x59987038`	`0xd764eca9`	`0x7ed9801d`	`0xdde4f1e0`
`0x524e678`	`0xa8ce47dc`	`0xa813fd76`	`0x8b58e09f`

Inverting 29-step MD5

For MD5 as a training set the first 16 intermediate inverse problems between steps 27-28 were chosen. The parameterization algorithm was run with seeds 0, 1, and 2 with the time limit of 24 hours, and it stopped after 4, 3, and 2 hours, respectively. The best result was achieved on seed 0: the best configuration was updated 8 times, and on the final one the objective function's value was 4 minutes (compared to 14 minutes on the default configuration).

The tuning was run one more time starting from the found configuration. This time the training set contained the last 15 intermediate problems between steps 27-28, and the problem for 28 steps. The reason was that on these 16 CNFs on the found configuration the objective function value was 5 hours 27 minutes, that is still reasonable, while on the default configuration it was 18 hours 37 minutes. The time limit was increased to 3 days, and the algorithm was run three times with seeds 3, 4, and 5. In all cases the time limit was reached: the objective function was calculated 584, 698, and 973 times, respectively. The best result was found with seed 5: the best configuration was updated 8 times, and the final objective function's value was 1 hour 30 minutes. Table 5 contains those 16 parameters which changed their values in the found configuration compared to the default one.

Table 5 The best Kissat's configuration found for MD5.

Parameter	Default value	Found value
chronolevels	100	1 000
decay	50	32
definitionticks	1 000 000	100
eliminatebound	16	2
eliminateocclim	2 000	1 000
emaslow	100 000	75 000
minimizedepth	1 000	100
restartmargin	10	20
shrink	3	0
stable	1	2
substituterounds	2	32
subsumeclslim	1 000	10 000
sweepmaxclauses	4 096	2 048
target	1	2
tier2	6	10
vivifytier2	6	5

The tuned and the default Kissat3 were run on the PC with the time limit of 24 hours on 32 CNFs: 31 intermediate inverse problems between steps 28-29, and the inverse problem for 29 steps. The results are presented as a cactus plot in Figure 4. The default version solved 6 intermediate problems for j of 1, 3, 5, 8, 9, and 10, while the tuned one solved 10 problems for j of 1, 2, 3, 5, 6, 7, 8, 9, 11, and 13. As a result, (28 13/32)-step MD5 was inverted. The figure clearly shows that the tuned version's performance is much better.

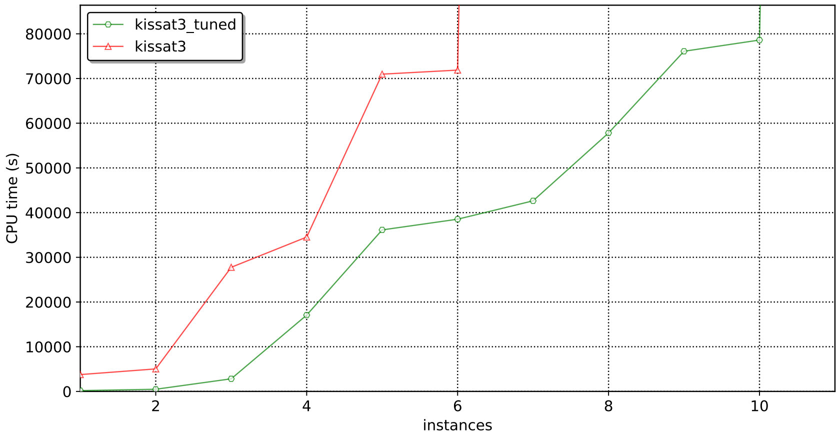


Figure 4 Comparison of the default Kissat with its tuned version on intermediate inverse problems for steps 28-29 of MD5, 1-hash.

On the next stage, intermediate inverse problems with j of 20, 24, and 28 between steps 28-29, as well as the inverse problem for 29 steps were considered. For each problem, at most 180 cubes were generated, a 180-core task was formed, and the tuned Kissat was run on each subproblem with the time limit of 24 hours on the supercomputer. On the first two problems solutions were found in 4 and 7 hours, respectively, yet nothing was found for the remaining two ones. It became clear that the inverse problem for 29 steps is quite hard, so this problem was split via march_cu into 74 470 cubes using the threshold $n = 5493$. These cubes were divided into 10 parts to form 10 7-days 180-core tasks with the time limit of 15 000 seconds for each subproblem. The first two task were completed with no found solution, yet in the third task a solution was found in 37 hours. The corresponding preimage is presented in Table 6.

Table 6 A preimage of 128 1s produced by 29-step MD5-1.

0xe1051a9e	0x48120773	0x996a5457	0xaaa1d815
0x37d8149c	0x5f999c05	0x182ba14b	0xdfff1673
0xc5db0a2f	0x44430b2a	0xa269f5a2	0x69781b85
0x2b7f0939	0xc1ff3c22	0xc55e990f	0x96ba3fb8

Discussion

According to the results, the hypothesis proposed in Section 6 is experimentally confirmed, and it gives the fourth take away: *tuning a state-of-the-art CDCL solver's parameters on simple intermediate inverse problems allows faster solving of hard intermediate inverse problems.*

Two found configurations for SHA-1 and MD5 differ quite a lot, but the following values exist in both of them: definitionticks=100, minimizedepth=100, restartmargin=20, and stable=2.

It seems realistic to invert 25-step SHA-1 via the proposed approach, but for 30-step MD5 it is likely that additional effective algorithmic techniques are required.

Reproducibility

The preimages were hard to find, but their verification via direct computations takes a fraction of a second. The correctness in case of 24-step SHA-1 was verified via the tool VERIFY-PREIMAGE from Nossum's repository SHA1-SAT. As for MD5, it was done by modifying the reference implementation from [37] written in C. First, padding (see Section 2), as well as all steps but the first 29 ones must be deleted. Then the preimage should be given as an input. As a result, 128 1s are produced.

8 Related Work

SAT-based cryptanalysis has been applied to cryptographic hash functions of the MD family as follows. In [27], practical collision attacks on MD4 and MD5 were performed. In [10], 39-step MD4 was inverted, while for steps 40-43 it was done in [45]. 26-step MD5 was inverted in [10], while for 27- and 28-step versions it was done in [22, 46]. As for the SHA family, the situation is as follows. Step-reduced versions of SHA-0, SHA-256, and SHA-3 were inverted in [22, 17, 34]. 22- and 23-step SHA-1 were inverted in [42, 22]. In [4], a weakened 24-step SHA-1 was inverted, such that the number of known hash bits was reduced from 160 to 128. In [35], full-step version of SHA-1 was inverted, but most message bits were assigned randomly, yet in [8] a similar result was achieved for SHA-256. Algebraic fault attacks on SHA-1 and SHA-256 were proposed in [33]. Collisions for SHA-1 were found in [43].

The aforementioned SAT-based preimage attacks on step-reduced MD5 and SHA-1 are practical. Also, theoretical preimage attacks on 62-step SHA-1 and full MD5 exist [12, 38].

Recently, Cube-and-Conquer has been successfully used to solve the Boolean Pythagorean Triples problem [16], the Schur number five problem [15], Lam's problem [7], and in model finding [3].

9 Conclusions and Future Work

This paper proposed a new type of intermediate inverse problems between any two steps $(i, i+1)$ of a cryptographic hash function from a wide class. First, these problems are useful to make some progress if i steps can be inverted in reasonable time while the inversion of $i+1$ steps is infeasible. Second, the simplest intermediate problems can be used to tune a CDCL solver so it can cope with previously unattainable problems. Third, if some intermediate inverse problems are solved, a runtime estimation for inverting $i+1$ steps can be calculated, so the corresponding computational resources can be allocated for this purpose. Fourth, in some cases it is not even needed to invert $i+1$ steps directly since solutions of the intermediate inverse problems can be simultaneously preimages of $i+1$ steps.

The main result of the paper is inverting 29-step MD5 and 24-step SHA-1 for the first time, thus making a clear progress in solving two hard computational problems.

In the future we are going to use more sophisticated parametrization algorithms and apply the proposed technique to analyze other cryptographic hash functions.

References

1 Carlos Ansótegui, Josep Pon, Meinolf Sellmann, and Kevin Tierney. Pydgga: Distributed GGA for automatic configuration. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 11–20. Springer, 2021. doi:10.1007/978-3-030-80223-3_2.

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, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2009. doi:10.1007/978-3-642-04244-7_14.

3 João Araújo, Choiwah Chow, and Mikolás Janota. Symmetries for cube-and-conquer in finite model finding. 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 8:1–8:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CP.2023.8.

4 Emanuele Bellini, Alessandro De Piccoli, Rusydi H. Makarim, Sergio Polese, Lorenzo Riva, and Andrea Visconti. New records of pre-image search of reduced SHA-1 using SAT solvers. In Debasis Giri, Kim-Kwang Raymond Choo, Saminathan Ponnusamy, Weizhi Meng, Sedat Akleylek, and Santi Prasad Maity, editors, *Proceedings of the Seventh International Conference on Mathematics and Computing - ICMC 2021, Shibpur, India*, volume 1412 of *Advances in Intelligent Systems and Computing*, pages 141–151. Springer, 2021. doi:10.1007/978-981-16-6890-6_11.

5 Armin Biere and Mathias Fleury. Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *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.

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

7 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, Virtual Event, February 2-9, 2021*, pages 3669–3676. AAAI Press, 2021. doi:10.1609/AAAI.V35I5.16483.

8 Davin Choo, Mate Soos, Kian Ming Adam Chai, and Kuldeep S. Meel. Bosphorus: Bridging ANF and CNF solvers. In Jürgen Teich and Franco Fummi, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, pages 468–473. IEEE, 2019. doi:10.23919/DATE.2019.8715061.

9 Ivan Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1989. doi:10.1007/0-387-34805-0_39.

10 Debapratim De, Abishek Kumarasubramanian, and Ramarathnam Venkatesan. Inversion attacks on secure hash functions using satsolvers. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 377–382. Springer, 2007. doi:10.1007/978-3-540-72788-0_36.

11 Donald E. Eastlake and Paul E. Jones. US secure hash algorithm 1 (SHA1). *RFC*, 3174:1–22, 2001. doi:10.17487/RFC3174.

12 Thomas Espitau, Pierre-Alain Fouque, and Pierre Karpman. Higher-order differential meet-in-the-middle preimage attacks on SHA-1 and BLAKE. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual*

Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I, volume 9215 of *Lecture Notes in Computer Science*, pages 683–701. Springer, 2015. doi:10.1007/978-3-662-47989-6_33.

13 Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, volume 7261 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2011. doi:10.1007/978-3-642-34188-5_8.

14 Marijn Heule and Hans van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *J. Satisf. Boolean Model. Comput.*, 2(1-4):47–59, 2006. doi:10.3233/SAT190016.

15 Marijn J. H. Heule. Schur number five. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 6598–6606. AAAI Press, 2018. doi:10.1609/AAAI.V32I1.12209.

16 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.

17 Ekawat Homsirikamol, Pawel Morawiecki, Marcin Rogawski, and Marian Srebrny. Security margin evaluation of SHA-3 contest finalists through sat-based attacks. In Agostino Cortesi, Nabendu Chaki, Khalid Saeed, and Slawomir T. Wierzchon, editors, *Computer Information Systems and Industrial Management - 11th IFIP TC 8 International Conference, CISIM 2012, Venice, Italy, September 26-28, 2012. Proceedings*, volume 7564 of *Lecture Notes in Computer Science*, pages 56–67. Springer, 2012. doi:10.1007/978-3-642-33260-9_4.

18 Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Automated configuration and selection of SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 481–507. IOS Press, 2021. doi:10.3233/FAIA200995.

19 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 - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer, 2011. doi:10.1007/978-3-642-25566-3_40.

20 Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 1152–1157. AAAI Press, 2007. URL: http://www.aaai.org/Library/AAAI/2007/aaai07-183.php.

21 Stepan Kochemazov. Exploring the limits of problem-specific adaptations of SAT solvers in SAT-based cryptanalysis. In Leonid Sokolinsky and Mikhail Zymbler, editors, *Parallel Computational Technologies*, pages 149–163. Springer International Publishing, 2021. doi:10.1007/978-3-030-81691-9_11.

22 Florian Legendre, Gilles Dequen, and Michaël Krajecki. Encoding hash functions as a SAT problem. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, pages 916–921. IEEE Computer Society, 2012. doi:10.1109/ICTAI.2012.128.

23 Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *J. Mach. Learn. Res.*, 23:54:1–54:9, 2022. URL: http://jmlr.org/papers/v23/21-0888.html.

24 João Marques-Silva and Karem Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999. doi:10.1109/12.769433.

25 Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. doi:10.1201/9781439821916.

26 Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989. doi:10.1007/0-387-34805-0_21.

27 Ilya Mironov and Lintao Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 2006. doi:10.1007/11814948_13.

28 Pawel Morawiecki and Marian Srebrny. A SAT-based preimage analysis of reduced Keccak hash functions. *Inf. Process. Lett.*, 113(10-11):392–397, 2013. doi:10.1016/J.IPL.2013.03.004.

29 Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. doi:10.1017/CBO9780511814075.

30 Heinz Mühlenbein. How genetic algorithms really work: Mutation and hillclimbing. In Reinhard Männer and Bernard Manderick, editors, *Parallel Problem Solving from Nature 2, PPSN-II, Brussels, Belgium, September 28-30, 1992*, pages 15–26. Elsevier, 1992.

31 Kevin P. Murphy. *Machine learning - a probabilistic perspective*. Adaptive computation and machine learning series. MIT Press, 2012.

32 Saeed Nejati and Vijay Ganesh. CDCL(Crypto) SAT solvers for cryptanalysis. In Tima Pakfetrat, Guy-Vincent Jourdan, Kostas Kontogiannis, and Robert F. Enenkel, editors, *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON 2019, Markham, Ontario, Canada, November 4-6, 2019*, pages 311–316. ACM, 2019. doi:10.5555/3370272.3370307.

33 Saeed Nejati, Jan Horácek, Catherine H. Gebotys, and Vijay Ganesh. Algebraic fault attack on SHA hash functions using programmatic SAT 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 737–754. Springer, 2018. doi:10.1007/978-3-319-98334-9_47.

34 Saeed Nejati, Jia Hui Liang, Catherine H. Gebotys, Krzysztof Czarnecki, and Vijay Ganesh. Adaptive restart and CEGAR-based solver for inverting cryptographic hash functions. In Andrei Paskevich and Thomas Wies, editors, *Verified Software. Theories, Tools, and Experiments - 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers*, volume 10712 of *Lecture Notes in Computer Science*, pages 120–131. Springer, 2017. doi:10.1007/978-3-319-72308-2_8.

35 Vegard Nossum. SAT-based preimage attacks on SHA-1. Master's thesis, University of Oslo, Department of Informatics, 2012.

36 Ronald L. Rivest. The MD4 message digest algorithm. In Alfred Menezes and Scott A. Vanstone, editors, *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 303–311. Springer, 1990. doi:10.1007/3-540-38424-3_22.

37 Ronald L. Rivest. The MD5 message-digest algorithm. *RFC*, 1321:1–21, 1992. doi:10.17487/RFC1321.

38 Yu Sasaki, Wataru Komatsubara, Yasuhide Sakai, Lei Wang, Mitsugu Iwamoto, Kazuo Sakiyama, and Kazuo Ohta. Meet-in-the-middle preimage attacks revisited - new results on MD5 and HAVAL. In Pierangela Samarati, editor, *SECRYPT 2013 - Proceedings of the 10th International Conference on Security and Cryptography, Reykjavík, Iceland, 29-31 July, 2013*, pages 111–122. SciTePress, 2013. URL: https://ieeexplore.ieee.org/document/7223160/.

39 Alexander A. Semenov, Daniil Chivilikhin, Artem Pavlenko, Ilya V. Otpuschennikov, Vladimir Ulyantsev, and Alexey Ignatiev. Evaluating the hardness of SAT instances using evolutionary optimization algorithms. 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 47:1–47:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CP.2021.47.

40 Alexander A. Semenov, Ilya V. Otpuschennikov, Irina Gribanova, Oleg Zaikin, and Stepan Kochemazov. Translation of algorithmic descriptions of discrete functions to SAT with applications to cryptanalysis problems. *Log. Methods Comput. Sci.*, 16(1), 2020. doi:10.23638/LMCS-16(1:29)2020.

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

42 Marian Srebrny, Mateusz Srebrny, and Lidia Stepien. SAT as a programming environment for linear algebra and cryptanalysis. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008. URL: http://isaim2008.unl.edu/PAPERS/SS1-AI+Logic/MSrebrny-ss1.pdf.

43 Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 570–596. Springer, 2017. doi:10.1007/978-3-319-63688-7_19.

44 Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In Ronald Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005. doi:10.1007/11426639_2.

45 Oleg Zaikin. Inverting 43-step MD4 via Cube-and-Conquer. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 1894–1900. ijcai.org, 2022. doi:10.24963/IJCAI.2022/263.

46 Oleg Zaikin. Inverting cryptographic hash functions via Cube-and-Conquer. *J. Artif. Intell. Res.*, in press.

47 Oleg Zaikin and Stepan Kochemazov. On black-box optimization in divide-and-conquer SAT solving. *Optim. Methods Softw.*, 36(4):672–696, 2021. doi:10.1080/10556788.2019.1685993.

Solving LBBD Master Problems with Constraint Programming and Domain-Independent Dynamic Programming

Jiachen Zhang
Department of Mechanical and Industrial Engineering, University of Toronto, Canada

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

Abstract

We investigate using Constraint Programming (CP) and Domain-Independent Dynamic Programming (DIDP) to solve the master problem in Logic-based Benders Decomposition (LBBD) models, in particular addressing the challenge of feasibility cut formulation. For CP, we exploit key variable manipulation, constraint-based expressions, and global constraints to construct three combinatorial cut encodings. For the state-based DIDP model, we propose two cut encoding approaches: using additional preconditions of state transitions or adding state constraints. Each of these approaches can be modeled using integer numeric variables or set variables, resulting in four novel encodings. We apply the three CP variants and four DIDP variants to simple assembly line balancing problems with sequence-dependent setup times type-1 (SUALBP-1). Experimental results show all approaches outperform a mixed-integer programming (MIP) based master problem and the state-of-the-art monolithic MIP model, with the three CP variants being superior to all of the DIDP approaches.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization

Keywords and phrases constraint programming, domain-independent dynamic programming, logic-based Benders decomposition, assembly line balancing, sequence-dependent setup

Digital Object Identifier 10.4230/LIPIcs.CP.2024.32

Funding *J. Christopher Beck*: Natural Sciences and Engineering Research Council of Canada.

1 Introduction

Logic-Based Benders Decomposition (LBBD) is one of the most powerful and convenient patterns of problem decomposition for solving combinatorial optimization problems [15]. While the most common combination within the Constraint Programming (CP) literature uses Mixed Integer Programming (MIP) for master problems and CP for subproblems [14], LBBD is compatible with various modeling and solving techniques. For example, subproblems have been modeled and solved with Satisfiability Modulo Theories (SMT) [22], Binary Decision Diagrams [11], and problem-specific algorithms [10, 29]. However, work investigating modeling and solution methods other than MIP for master problems in LBBD is sporadic [8]. In this paper, we explore the modeling and solving LBBD master problems with methods different from MIP.

As a constraint-based formalism, CP can readily accept cuts encoded as linear constraints. However, linear constraints tend to propagate weakly, resulting in poor master problem performance. The encoding methods proposed in this paper are more combinatorial and focus on key decision variables in the global constraints of the master problem CP model. As CP is competitive with MIP across a number of optimization problems [21], when the master problem is of the form that is better solved with CP, a CP-based master problem may outperform a corresponding MIP master problem if a good cut formulation can be achieved.

Domain-Independent Dynamic Programming (DIDP) is a recent exact framework to model and solve combinatorial optimization problems [19, 20]. Its success on well-known problems motivates us to investigate using DIDP for master problems in the LBBD framework. Since a DIDP model is defined as a state-transition system, encoding Benders cuts in DIDP differs fundamentally from the constraint-based encoding in MIP and CP.

As a case study, we use assembly line balancing problems with sequence-dependent setup times type-1 (SUALBP-1) [9]. The natural decomposition for this problem is to solve the Simple Assembly Line Balancing Problem type-1 (SALBP-1) as the master problem and to solve a traveling salesman problem with precedence constraints as a subproblem. Previous work shows that both CP and DIDP can outperform MIP for SALBP-1 [21], thus this choice allows us to test whether cuts can be formulated to maintain this advantage.

Our contributions are summarized as follows.

1. We formulate three alternative representations of feasibility cuts for SUALBP-1 for a CP-based master problem.
2. We propose four approaches to encode Benders feasibility cuts in a DIDP model of LBBD master problems based on using integer or set variables to encode preconditions or state constraints. We apply these approaches to SUALBP-1 and develop four feasibility cut encodings for a DIDP-based master problem.
3. We obtain superior results for SUALBP-1 in solving master problems with CP and DIDP rather than MIP, with CP outperforming DIDP. We provide statistical analysis and insights on our seven novel cut formulations.

This paper is organized as follows. The background is covered in Section 2. The three novel CP feasibility cut formulations for SUALBP-1 are introduced in Section 3. The four encoding methods of Benders feasibility cuts in DIDP and their instantiations for SUALBP-1 are presented in Section 4. The experimental results are presented in Section 5. We discuss the proposed approaches and results in Section 6, followed by our conclusions.

2 Background

2.1 Logic-Based Benders Decomposition

Logic-Based Benders Decomposition (LBBD) applies to problems that can be formulated as

$$\min_{\mathbf{x},\mathbf{y}}\{f(\mathbf{x},\mathbf{y})|C(\mathbf{x},\mathbf{y}),\mathbf{x}\in D_x,\mathbf{y}\in D_y\} \tag{1}$$

where $\mathbf{x}$ and $\mathbf{y}$ are decision variables in the domains D_x and D_y, while $f(\mathbf{x},\mathbf{y})$ and $C(\mathbf{x},\mathbf{y})$ represent the objective function and a set of constraints for these variables, respectively [13]. The variables are divided into two groups and, once some of the variables are fixed by solving a master problem and setting $\mathbf{x}=\overline{\mathbf{x}}$, the remaining subproblem is defined, often in the form of multiple independent subproblems. The subproblem (SP) has the form

$$SP(\overline{\mathbf{x}})=\min_{\mathbf{y}}\{f(\overline{\mathbf{x}},\mathbf{y})|C(\overline{\mathbf{x}},\mathbf{y}),\mathbf{y}\in D_y\}. \tag{2}$$

LBBD analyzes the SP solution to infer a function $B_{\overline{\mathbf{x}}}(\mathbf{x})$ that provides a lower bound on $f(\mathbf{x},\mathbf{y})$ for any given $\mathbf{x}\in D_x$. The bound is sharp for $\mathbf{x}=\overline{\mathbf{x}}$, i.e., $B_{\overline{\mathbf{x}}}(\mathbf{x})=SP(\overline{\mathbf{x}})$ [15].

Each iteration of LBBD begins by solving a Master Problem (MP):

$$MP(\overline{\mathbf{X}})=\min_{x,\beta}\{\beta|\beta\geq B_{\overline{\mathbf{x}}}(\mathbf{x}),\forall\overline{\mathbf{x}}\in\overline{\mathbf{X}},\mathbf{x}\in D_x\} \tag{3}$$

where the inequalities $\beta\geq B_{\overline{\mathbf{x}}}(\mathbf{x})$ are Benders cuts obtained from the subproblem solutions given $\mathbf{x}=\overline{\mathbf{x}}$. $\overline{\mathbf{X}}$ is the set of master problem solutions and is usually empty initially.

Defining ϕ^* as the optimal value of the original problem (1), the optimal MP value $MP(\overline{\mathbf{X}})$ is a lower bound on ϕ^*. If $\overline{\mathbf{x}}$ is an optimal MP solution, the corresponding subproblem is then solved to obtain $SP(\overline{\mathbf{x}})$ as an upper bound on ϕ^*, and a Benders cut $\beta \geq B_{\overline{\mathbf{x}}}(\mathbf{x})$ for the master problem, with $\overline{\mathbf{x}}$ added to $\overline{\mathbf{X}}$. The process repeats until the lower and upper bounds converge, i.e., until $MP(\overline{\mathbf{X}}) = \min_{\overline{\mathbf{x}} \in \overline{\mathbf{X}}} SP(\overline{\mathbf{x}})$. The convergence is guaranteed after a finite number of iterations, if D_x is finite [13].

In general, there are two LBBD variants, distinguished by subproblem types. When a subproblem is an optimization problem, we deduce a lower bound on ϕ^* in the form of a Benders optimality cut [31]. When a subproblem is a feasibility problem, a set of MP solutions are pruned by the corresponding Benders feasibility cut [1] according to the SP solution associated with $\overline{\mathbf{x}}$. In this work, we focus on encoding *Benders feasibility cuts.*

2.2 Domain-Independent Dynamic Programming

A DIDP model is described by Dynamic Programming Description Language (DyPDL), a solver-independent formalism to define a dynamic programming (DP) model [20]. In DyPDL, a problem is represented by states and transitions between states. A solution of the problem corresponds to a sequence of transitions satisfying particular conditions.

A DyPDL model is a tuple $\langle \mathcal{V}, S^0, \mathcal{T}, \mathcal{B}, \mathcal{C}, h \rangle$, where $\mathcal{V}$ is the set of state variables, S^0 is a state called the target state, $\mathcal{T}$ is the set of transitions, $\mathcal{B}$ is the set of base cases, $\mathcal{C}$ is the set of state constraints, and h is the set of dual bounds. A state variable is either an element, set, or numeric variable. A numeric state variable v may have a preference such as *less* (*more*), i.e., a state having smaller (larger) v dominates another state if the other state variables have the same value in the two states. Such a variable is called a resource variable.

Given a set of state variables $\mathcal{V} = \{v_1, ..., v_n\}$, a state is a tuple of values $S = (d_1, ..., d_n)$ where $d_i \in D_{v_i}$ for $i = 1, ..., n$, i.e., a state is a complete assignment to state variables. We denote the value d_i of variable v_i in state S by $S[v_i]$. Intuitively, the target state is the start of the state transition system and a base state is a goal, i.e., the end of the state transition system. State constraints are relations on state variables that must be satisfied by *all* states.

A transition τ is a 4-tuple $\langle eff_\tau, cost_\tau, pre_\tau, forced_\tau \rangle$ where eff_τ is the set of effects, $cost_\tau$ is the cost, pre_τ is the set of preconditions, and $forced_\tau \in \{\top, \bot\}$, where $\top$ represents *true* and $\bot$ represents *false.* The preconditions of a transition define when we can use it while the effects of a transition define what the state variables become if the transition fires. For detailed DIDP models of various optimization problems, please see existing DIDP papers [20, 21].

2.3 SUALBP-1

The Simple Assembly Line Balancing Problem (SALBP) is a well-studied production planning problem [5]. As setup operations such as tool changes, curing, or cooling processes are often required between consecutive tasks in real production lines [18], SUALBP incorporates setup times into SALBP [2], as shown in Fig. 1.

Problem Definition

SUALBP-1 consists of n assembly tasks, partially ordered with precedence constraints, that require processing on m ordered assembly stations. The tasks on a machine must all sequentially execute within the cycle time c. In SUALBP-1, the cycle time c is fixed and the objective is to minimize the number of stations m. Though all stations can perform all assembly tasks, if a task is assigned to station j, all its successors as defined by the precedence

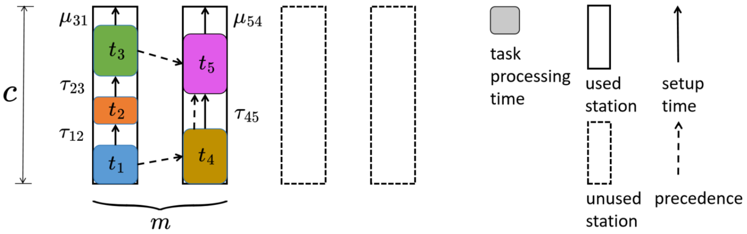


Figure 1 Example of SUALBP-1.

constraints must be assigned to the same or subsequent stations (i.e., $j, j+1, j+2, ..., m$). Tasks assigned to the same station must also be sequenced to satisfy the precedence constraints, if any. The deterministic processing time of a task is provided a priori. However, the setup before a task (*forward setup*) is dependent upon the previous task in the processing sequence of the station it is assigned to. There is also a sequence-dependent setup (*backward setup*) from the last task on a machine to the first task on the same machine to model the setup required between the end of a cycle and the start of the next one.

The setups are not symmetric, i.e., the setup time from task i to j might be different from that from task j to i. Nevertheless, the setups satisfy the triangle inequality. The decisions to be made for SUALBP-1 are (i) the assignment of tasks to stations; and (ii) the sequence of the tasks assigned to each station. We use the notation proposed by Esmaeilbeigi et al. [9], as shown in the Table 1 for SUALBP-1. To obtain all the parameters in the table, we adapt the preprocessing techniques in the literature [20, 9, 31].

SUALBP-1 has been solved with a number of approaches including MIP [9] and heuristics [25]. The state-of-the-art MIP model is the Second Station-Based Formulation (SSBF) [9] defined in Appendix A. The model uses two-indexed binary variables to encode task assignment, three-indexed binary variables to represent the precedence relations of pairs of tasks on a station, and auxiliary variables to help express the objective and constraints.

There is no existing LBBD approach specifically designed for SUALBP-1. The closest work is an LBBD algorithm for mixed-model assembly line balancing problem with sequence-dependent setups [1] that can be adapted (with significant simplification) to SUALBP-1. We discuss this model in Section 5.

In our parallel work currently under review [30], new state-of-the-art results are found with a monolithic DIDP model. Since our focus is on cut encoding in LBBD, we return to these results in the discussion.

Table 1 Notation and definition for SUALBP-1 [9].

Notation	Definition
$i, j \in V$	index and set of tasks
$k \in K$	index and set of stations
t_i	execution time for task $i \in V$
P_i (P_i^*)	set of direct (all) predecessors of task $i \in V$
S_i (S_i^*)	set of direct (all) successors of task $i \in V$
c	the cycle time
$\overline{m}$ ($\underline{m}$)	upper (lower) bound on the number of stations
τ_{ij} (μ_{ij})	forward (backward) setup times from task $i \in V$ to task j
$\underline{\tau}_i$ ($\underline{\mu}_i$)	the smallest forward (backward) setup time from any task to task $i \in V$
$\underline{t}_i$	a lower bound of the time contribution by task i, i.e., $\underline{t}_i = t_i + \min(\underline{\tau}_i, \underline{\mu}_i)$

3 CP-LBBD for SUALBP-1

In this section, we present three LBBD formulations for SUALBP-1 with CP master problems and Benders feasibility cuts.

3.1 CP Master Problem

SUALBP-1 fixes the cycle time (maximum station time) and seeks to minimize the number of stations used. In the LBBD framework, we decompose the problem to an assignment master problem and a scheduling subproblem for each station.

In all our approaches, the master problem assigns tasks to stations, minimizing the number of stations used, and ensuring that the precedence constraints between tasks and the cycle time limit are not violated. Without any Benders cuts, this master problem is identical to the Simple Assembly Line Balancing Problem type-1 (SALBP-1) [4].

For SALBP-1, Kuroiwa and Beck [20] improved the CP model proposed by Bukchin and Raviv [6] by using `Pack` global constraint. Our models differ from theirs in two ways: (1) t_i is replaced by $\underline{t}_i$ for task i to model a subproblem relaxation in the master problem and (2) three different combinatorial formulations of Benders feasibility cuts are used, one formulation in each model.

We define E_i as a lower bound on the number of stations required to schedule task i, L_i as a lower bound on the number of stations between the station of task i and the last station, inclusive, and d_{ij} as a lower bound on the number of stations between the stations of tasks i and j, inclusive:

$$E_i = \left\lceil \frac{\underline{t}_i + \sum_{j \in P_i^*} \underline{t}_j}{c} \right\rceil, \quad L_i = \left\lfloor \frac{\underline{t}_i - 1 + \sum_{j \in S_i^*} \underline{t}_j}{c} \right\rfloor, \quad d_{ij} = \left\lceil \frac{\underline{t}_i + \underline{t}_j - 1 + \sum_{v \in S_i^* \cap P_j^*} \underline{t}_v}{c} \right\rceil.$$

Let z be an integer decision variable representing the number of stations, x_i be an integer decision variable for the station that task i is assigned to, and y_k be an integer decision variable for the sum of the lower bound time contribution of tasks scheduled in station k. Then the CP model for the master problem, CP-MP, is as follows:

$$\begin{aligned}
&\min z && && \text{(4a)}\\
&\text{s.t. } \texttt{Pack}(\{y_k | k \in K\}, \{x_i | i \in V\}, \{\underline{t}_i | i \in V\}), && && \text{(4b)}\\
&\quad 0 \le y_k \le c, && \forall k \in K, && \text{(4c)}\\
&\quad E_i - 1 \le x_i \le z - 1 - L_i, && \forall i \in V, && \text{(4d)}\\
&\quad x_i + d_{ij} \le x_j, && \forall j \in V, \forall i \in P_j^*, \nexists v \in S_i^* \cap P_j^* : d_{ij} \le d_{iv} + d_{vj}. && \text{(4e)}
\end{aligned}$$

The `Pack` global constraint [27] ensures that for tasks "packed" onto stations, $y_k = \sum_{i \in V, x_i = k} \underline{t}_i$. Constraints (4c) and (4d) state the domains of y_k and x_i. Constraint (4b) and (4c) together ensure that the total task time on each station does not exceed the cycle time. Constraint (4e) is an enhanced version of the precedence constraint using d_{ij}.

3.2 CP Formulations for Benders Feasibility Cuts

For SUALBP-1, we develop three combinatorial CP formulations for Benders feasibility cuts by using key variable manipulation, a `Count_Different` expression, and a `Pack` constraint.

Let $\mathcal{J}$ be the set of subproblems leading to Benders cuts. Consider subproblem $j \in \mathcal{J}$ corresponding to station k, let $\mathcal{I}^j$ be the set of tasks assigned to the station that cannot all be scheduled within the cycle time, then the j-th Benders feasibility cut based on manipulation of the key decision variables, i.e., the station assignment specified by x_i, is as follows:

$$\sum_{i \in \mathcal{I}^j} (x_i = k) \leq |\mathcal{I}^j| - 1, \ \forall k \in K. \tag{5}$$

Chu and Xia defined a valid Benders cut as a logical expression having two properties [7]:

- Property 1: The cut must exclude the current MP solution if it is not globally feasible.
- Property 2: The cut must not remove any globally feasible solutions.

Property 1 ensures finite convergence if the MP variables have finite domains. Property 2 assures optimality since the cut never removes globally feasible solutions.

▶ **Proposition 1.** *Cut (5) is valid.*

Proof. As $x_i = k$ specifies the station assignment and there are $|\mathcal{I}^j|$ tasks in $\mathcal{I}^j$, the cut prevents the tasks in $\mathcal{I}^j$ from being all assigned to the same station and satisfies Property 1. Since the solutions removed by this encoding are all infeasible globally with the set of tasks $\mathcal{I}^j$ assigned to any station, Property 2 is satisfied. ◀

The constraint-based expression `Count_Different` takes a list of (more than one) variables as input and returns the number of distinct values of these variables [17]. The j-th cut based on `Count_Different` is as follows:

$$\texttt{Count\_Different}(\{x_i | i \in \mathcal{I}^j\}) \geq 2. \tag{6}$$

▶ **Proposition 2.** *Cut (6) is valid.*

Proof. This constraint guarantees that the number of distinct values in $\{x_i | i \in \mathcal{I}^j\}$ is at least 2 and implies (5). Thus, Properties 1 and 2 are satisfied. ◀

The j-th cut based on the global constraint `Pack` is as follows:

$$\texttt{Pack}(\{w_k | k \in K\}, \{x_i | i \in \mathcal{I}^j\}, \{\mathbf{1}_i | i \in \mathcal{I}^j\}), \tag{7}$$

where $0 \leq w_k \leq |\mathcal{I}^j| - 1$ and $\mathbf{1}_i = 1, \forall i \in \mathcal{I}^j$.

▶ **Proposition 3.** *Cut (7) is valid.*

Proof. Since $\mathbf{1}_i$ has unit length and $w_k \leq |\mathcal{I}^j| - 1$, this cut assures that no more than $|\mathcal{I}^j| - 1$ tasks in $\mathcal{I}^j$ are assigned to any station and satisfies Property 1. Similar to the proof for Proposition 1, Property 2 is satisfied. ◀

The CP-LBBD models with cut (5), (6), and (7) are referred to as CP-LBBD$_a$, CP-LBBD$_c$, and CP-LBBD$_p$, corresponding to “assignment”, “count”, and “pack”, respectively.

4 DIDP-LBBD for SUALBP-1

In this section, we present the DIDP model for the master problem for SUALBP-1, four general encoding methods for Benders feasibility cuts, and their instantiation to the Benders cuts for SUALBP-1.

4.1 Master Problem

As stated in Section 3.1, the master problem is equivalent to the SALBP-1. Our DIDP formulations for the master problem (with Benders cuts) of SUALBP-1 are inspired by an existing DIDP model for SALBP-1 [20], which is defined as follows:

State variables.

- U: set variable for unscheduled tasks. In the target state (i.e., the initial state), $U = V$.
- r: integer resource variable for the remaining time (cycle time minus used time) of the current station. In the target state, $r = 0$. A larger r is better.

Base case. A base case is a set of conditions to terminate the recursion. The base case of the DIDP model is $U = \emptyset$.

Transitions.

- $Assign_i = \langle U \to U \backslash \{i\} \ \wedge \ r \to r - \underline{t}_i,\ 0,\ i \in U \ \wedge \ \underline{t}_i \leq r \ \wedge \ U \cap P_i^* = \emptyset,\ \perp \rangle$: assign task i to the current station.
- $Open = \langle r \to c,\ 1,\ (i \notin U \ \vee \ r < \underline{t}_i \ \vee \ U \cap P_i^* \neq \emptyset) \mid \forall i \in V,\ \perp \rangle$: open a new station.

Note that we use $\underline{t}_i$ instead of t_i in the master problem to estimate the setup times that are exactly calculated in the subproblems.

Theoretically, the transition *Open* can be used at any state. However, a state with a closed station that can accommodate an unscheduled task is dominated by an otherwise identical one that schedules such a task. Thus, a dominance rule, stating that a station can only be opened if no task can be assigned to the current station, is encoded in the preconditions for transition *Open*. This dominance rule plays an important role in the efficiency of the DIDP model [20] but presents a complication for our cut formulations (see Section 4.3.2).

Recursive function. We use $f(U, r)$ to represent the cost of a state. Let $U_1 = \{i \in U \mid r \geq \underline{t}_i \ \wedge \ U \cap P_i^* = \emptyset\}$ be the set of tasks with all their predecessors scheduled that can fit on the current station. The recursive function of the DIDP model is as follows:

$$\texttt{compute } f(V, 0) \tag{8a}$$

$$f(U, r) = \begin{cases} 0 & \text{if } U = \emptyset, & \text{(i)} \\ 1 + f(U, c) & \text{else if } U_1 = \emptyset, & \text{(ii)} \\ \min_{i \in U_1} f(U \backslash \{i\}, r - \underline{t}_i) & \text{else,} & \text{(iii)} \end{cases} \tag{8b}$$

$$f(U, r) \leq f(U, r^{'}), \quad \text{if } r \geq r^{'}, \tag{8c}$$

$$f(U, r) \geq \max \begin{cases} \lceil \frac{\sum_{i \in U} \underline{t}_i - r}{c} \rceil, & \text{(i)} \\ \sum_{i \in U} w_i^2 + \lceil \sum_{i \in U} w_i^{'2} - l^2 \rceil, & \text{(ii)} \\ \lceil \sum_{i \in U} w_i^3 - l^3 \rceil. & \text{(iii)} \end{cases} \tag{8d}$$

The term (8a) is to compute the cost of the target state. Equation (8b) is the main recursion of the DIDP model. Specifically, (8b-i) refers to the base case, while (8b-ii) corresponds to opening a new station and (8b-iii) refers to assigning task i to the current station. Inequality (8c) formulates state domination due to the resource variable: if other

Table 2 Numeric constants for calculating a knapsack-based dual bound.

$\underline{t}_i$	(0, c/2)	c/2	(c/2, c]	$\underline{t}_i$	(0, c/3)	c/3	(c/3, c/2)	2c/3	(2c/3, c]
w_i^2	0	0	1	w_i^3	0	1/3	1/2	2/3	1
$w_i^{'2}$	0	1/2	0						

variables are equal, a state with a larger remaining time dominates. (8d-i), (8d-ii), and (8d-iii) are valid dual bounds proposed by Scholl and Klein [26] with numeric constants w^2, w'^2, w^3 indexed by a task i and depending on $\underline{t}_i$, as shown in Table 2.

4.2 Feasibility Cut Encoding in DIDP-LBBD

Let $\mathbf{x}$ be the decision variables in the master problem and let $\overline{\mathbf{x}}$ be the optimal solution of the latest MP iteration. Let $\mathcal{I}^j$ be the set of MP variable indices that appear in the j-th subproblem, then the Benders feasibility cut obtained from this subproblem is of the following form:

$$\sum_{i \in \mathcal{I}^j} (x_i = \overline{x}_i) \leq |\mathcal{I}^j| - 1. \tag{9}$$

This form is often formulated as a linear constraint in the MIP master problem and we call it the j-th cut.

In DIDP, however, a cut of form (9) cannot be directly represented with state variables. Thus, instead of adding only a constraint to the DIDP model, we add a new state variable for each cut, with relevant transitions updating the variable value. New preconditions or state constraints are also added.

4.2.1 Counting-based Encoding

Our first two encoding methods are based on integer numeric variables in DIDP. Let g^j be an integer numeric variable that counts the active variable-value pairs in the left-hand side (LHS) of the cut (9), i.e., $g^j = \sum_{i \in \mathcal{I}^j} (x_i = \overline{x}_i)$. In the target state, the value of g^j is 0. Let $\mathcal{F}^j$ be the function that updates the value of g^j according to transitions. If the effects eff_τ of transition τ imply that $x_i = \overline{x}_i$ for some $i \in \mathcal{I}^j$ and $x_k \neq \overline{x}_k$ for some $k \in \mathcal{I}^j$, we have $\mathcal{F}^j(\tau) = |\mathcal{U}^j_\tau| - |\mathcal{D}^j_\tau|$, where $\mathcal{U}^j_\tau$ ($\mathcal{D}^j_\tau$) is the set of the variable indices of the variable-value pairs that are changed from inactive (active) to active (inactive) by transition τ with respect to the j-th cut, with $i \in \mathcal{U}^j_\tau$ and $k \in \mathcal{D}^j_\tau$. Let S be the state where the preconditions of transition τ are satisfied, and let $S' = S[[\tau]]$ be the state reachable from S by τ, we have $S'[g^j] = S[g^j] + \mathcal{F}^j(\tau)$.

In practice, the implementation of $\mathcal{F}$ depends on the problem and we define the encoding for SUALBP-1 later in Section 4.3. With the LHS of cut (9) modeled, we use preconditions or state constraints to model the right-hand side (RHS).

Precondition-based Encoding. Our first method for modeling the RHS of (9) is based on preconditions. Specifically, for the cut with the form (9), we add a precondition for each transition in the DIDP model that can modify the LHS variables as follows:

$$S[g^j] + \mathcal{F}^j(\tau) \leq |\mathcal{I}^j| - 1, \tag{10}$$

where τ is the transition. If the precondition is violated, the transition τ is not permitted.

State Constraint-based Encoding. Our second method for modeling the RHS of (9) is based on state constraints that need to be satisfied by all states. The state constraint for the j-th cut is as follows:

$$S[g^j] \leq |\mathcal{I}^j| - 1, \tag{11}$$

where S is any state. A state constraint is evaluated after a state is created but a precondition would prevent the state from being created.

4.2.2 Set-based Encoding

Our second two encoding methods are based on set variables in DIDP. Let Ω^j be a set variable that keeps track of the active variable-value pairs in the LHS of the cut (9). More specifically, the set variable Ω^j contains an element e_i iff $x_i = \overline{x}_i$ is satisfied in a state. In the target state, $\Omega^j = \emptyset$. Let $\mathcal{O}^j$ be the function that updates the value of Ω^j according to transitions. If the effects eff_τ of transition τ imply that $x_i = \overline{x}_i$ for some $i \in \mathcal{I}^j$ and $x_k \neq \overline{x}_k$ for some $k \in \mathcal{I}^j$, let $\mathcal{U}^j_\tau$ be the set containing all such i and $\mathcal{D}^j_\tau$ be the set containing all such k, we have $\mathcal{O}^j(\tau) = (S[\Omega^j] \cup \mathcal{U}^j_\tau) \backslash \mathcal{D}^j_\tau$. Let S be a state and $S' = S[[\tau]]$ be the state reachable from S by τ, we have $S'[\Omega^j] = \mathcal{O}^j(\tau)$. Similar to the counting-based encoding, we use preconditions or state constraints to model the RHS.

Precondition-based Encoding. For the cut (9), we add a precondition for each transition that can modify $\mathcal{O}^j(\tau)$ in the DIDP model as follows:

$$\mathcal{I}^j \nsubseteq \mathcal{O}^j(\tau), \tag{12}$$

where τ is the transition. $\mathcal{O}^j(\tau)$ gives the value of Ω^j after the transition and may contain items that are not in $\mathcal{I}^j$. The precondition prevents Ω^j from including all the items in $\mathcal{I}^j$.

State Constraint-based Encoding. The state constraint for the j-th cut is as follows:

$$\mathcal{I}^j \nsubseteq S[\Omega^j], \tag{13}$$

where S is any state.

4.2.3 Weakness of the DIDP Encoding

There is a fundamental weakness in the aforementioned DIDP encodings compared to constraint-based models: adding a cut expands the search space. All four DIDP encoding methods rely on adding a new state variable to the MP to keep track of the changes to the LHS of (9) caused by transitions. After adding a new state variable corresponding to the j-th cut, the original state space size is multiplied by the cardinality of the $\mathcal{I}^j$. We return to this point in Section 6.

4.3 Encoding DIDP-LBBD Cuts for SUALBP-1

The formulations above can be used for any cut of the form (9). Here we formally present four cut formulations for SUALBP-1.

4.3.1 Counting-based Precondition Encoding

For cut $j \in \mathcal{J}$, recall that $\mathcal{I}^j$ is the set of tasks assigned to the station that cannot be scheduled within the cycle time. Define function $\mathcal{F}^j$ such that $\mathcal{F}^j(i) = 1$ if $i \in \mathcal{I}^j$ and 0 otherwise. In order to encode this cut, we add a new state variable g^j with its value being 0 at the target state. We then modify the recursive formulation (8b) as follows.

$$f(U, r, \{g^j \mid \forall j \in J\}) =$$
$$\begin{cases} 0 & \text{if } U = \emptyset, & \text{(i)} \\ 1 + f(U, c, \{0 \mid \forall j \in J\}) & \text{else if } U_2 = \emptyset, & \text{(ii)} \\ \min_{i \in U_2} f(U \backslash \{i\}, r - \underline{t}_i, \{g^j + \mathcal{F}^j(i) \mid \forall j \in J\}) & \text{else.} & \text{(iii)} \end{cases} \tag{14}$$

where $U_2 = \{i \in U \mid r \geq \underline{t}_i \ \wedge \ U \cap P^*_i = \emptyset \ \wedge \ (\forall j \in \mathcal{J}, g^j + \mathcal{F}^j(i) \leq |\mathcal{I}^j| - 1)\}$.

▶ **Proposition 4.** *The counting-based precondition encoding is valid.*

Proof. For any cut $j \in \mathcal{J}$, g^j counts the number of variable-value pairs that appear in the current station. With transition *Open*, the current station changes to the next station and $g^j = 0$, as shown in (14-ii). As shown in (14-iii), with transition $Assign_i$ for any i, since $\mathcal{F}^j$ is non-negative and $g^j + \mathcal{F}^j(i) \leq |\mathcal{I}^j| - 1$ is the precondition stated in U_2, we have $S[g^j] \leq |\mathcal{I}^j| - 1$ at any state S of the DIDP model. This guarantees that the same set of tasks are never assigned to the same station and satisfies Property 1. Since the solutions removed by this encoding are the solutions with the set of tasks $\mathcal{I}^j$ assigned to any station, they are all infeasible globally as the task processing times and setup times are independent of stations, and thus Property 2 is satisfied. ◀

4.3.2 Counting-based State Constraint Encoding

We keep the modified effects and use state constraints instead of preconditions to enforce the logic of feasibility cuts. The recursive formulation (8b) becomes:

$$f(U, r, \{g^j \mid \forall j \in J\}) =$$
$$\begin{cases} 0 & \text{if } U = \emptyset, & \text{(i)} \\ 1 + f(U, c, \{0 \mid \forall j \in J\}) & \text{else if } U_2 = \emptyset, & \text{(ii)} \\ \min_{i \in U_1} f(U \backslash \{i\}, r - \underline{t}_i, \{g^j + \mathcal{F}^j(i) \mid \forall j \in J\}) & \text{else if } U_1 \neq \emptyset. & \text{(iii)} \end{cases} \tag{15}$$

In (15-iii), there is no precondition preventing a task assignment that violates Benders cut. Instead, state constraints are added to prune the resulting states as follows:

$$g^j \leq |\mathcal{I}^j| - 1, \forall j \in \mathcal{J}. \tag{16}$$

However, as noted, there is an interaction between the cut and the dominance rule associated with the preconditions of transition *Open*: if we maintain the original precondition on *Open* (i.e., $U_1 = \emptyset$), then a state where only tasks that violate the cut can be scheduled will result in a dead-end. The transitions satisfying (15-iii) will fire and the resulting states will all violate the state constraints. Thus, no state is reachable from the current state. However, a new station should be opened in the state when no tasks can be scheduled. To ensure the correctness of the model, either we remove the dominance and allow *Open* at any time, or we maintain it by allowing *Open* when no tasks, including those violating cuts, can be scheduled (the new preconditions become $U_2 = \emptyset$). We select the latter option to maintain the efficiency of the proposed DIDP model.

▶ **Proposition 5.** *The counting-based state constraint encoding is valid.*

Proof. Similar to the proof for Proposition 4, we have $S[g^j] \leq |\mathcal{I}^j| - 1$ at any state S of the DIDP model. Property 1 and Property 2 are hence satisfied. ◀

4.3.3 Set-based Precondition Encoding

To encode this cut, we add a new state variable Ω^j with its value being $\emptyset$ at the target state. We then modify the recursive formulation (8b) in the DIDP model of the master problem to address all the Benders feasibility cuts:

$$f(U, r, \{\Omega^j \mid \forall j \in J\}) =$$
$$\begin{cases} 0 & \text{if } U = \emptyset, & \text{(i)} \\ 1 + f(U, c, \{\emptyset \mid \forall j \in J\}) & \text{else if } U_3 = \emptyset, & \text{(ii)} \\ \min_{i \in U_3} f(U \backslash \{i\}, r - \underline{t}_i, \{\Omega^j \cup \{i\} \mid \forall j \in J\}) & \text{else.} & \text{(iii)} \end{cases} \tag{17}$$

where $U_3 = \{i \in U \mid r \geq \underline{t}_i \ \wedge \ U \cap P_i^* = \emptyset \ \wedge \ (\forall j \in \mathcal{J}, \mathcal{I}^j \not\subseteq \Omega^j \cup \{i\})\}$.

▶ **Proposition 6.** *The set-based precondition encoding is valid.*

Proof. For any cut $j \in \mathcal{J}$, Ω^j keeps track of the variable-value pairs that appear in the current station. With transition *Open*, the current station changes to the next station and $\Omega^j = \emptyset$, as shown in (17-ii). As shown in (17-iii), with transition $Assign_i$ for any i, since the effects never remove any element from Ω^j and $\mathcal{I}^j \not\subseteq \Omega^j \cup \{i\}$ is the precondition stated in U_3, we have $\mathcal{I}^j \not\subseteq S[\Omega^j]$ at any state S of the DIDP model. This guarantees that the same set of tasks would never appear in the same station and satisfies Property 1. Similar to the proof for Proposition 4, Property 2 is satisfied. ◀

4.3.4 Set-based State Constraint Encoding

The recursive formulation (8b) becomes:

$$f(U, r, \{\Omega^j \mid \forall j \in J\}) =$$
$$\begin{cases} 0 & \text{if } U = \emptyset, & \text{(i)} \\ 1 + f(U, c, \{\emptyset \mid \forall j \in J\}) & \text{else if } U_3 = \emptyset, & \text{(ii)} \\ \min_{i \in U_1} f(U \backslash \{i\}, r - \underline{t}_i, \{\Omega^j \cup \{i\} \mid \forall j \in J\}) & \text{else if } U_1 \neq \emptyset. & \text{(iii)} \end{cases} \tag{18}$$

The added state constraint is:

$$\mathcal{I}^j \not\subseteq \Omega^j, \forall j \in \mathcal{J}. \tag{19}$$

Similar to (15), we maintain the dominance specified by the preconditions of the transition *Open* by inserting the case violating state constraints (19) into the preconditions (the new preconditions become $U_3 = \emptyset$).

▶ **Proposition 7.** *The set-based state constraint encoding is valid.*

Proof. Similar to the proof for Proposition 6, Property 1 and Property 2 are satisfied. ◀

The DIDP-LBBD models with recursive formulation (14), (15), (17), and (18) replacing (8b) are referred as DIDP-LBBD$_{cPre}$, DIDP-LBBD$_{cCon}$, DIDP-LBBD$_{sPre}$, and DIDP-LBBD$_{sCon}$, respectively, where "c" and "s" correspond to "count" and "set" and "Pre" and "Con" map to "precondition" and "constraint".

5 Experimental Evaluation

In this section, we compare the performance of our CP-LBBD, DIDP-LBBD, and MIP-LBBD models against the state-of-the-art MIP model [9] (see Appendix A) on the 788 instances of the SBF2 data set [25].[1]

5.1 MIP-LBBD Master Problem

We use a MIP-LBBD model as the baseline LBBD approach. For the master problem, instead of a simplified MIP formulation proposed by Akpinar et. al [1] we use the state-of-the-art NF4 MIP formulation [23] for SALBP-1 and replace t_i by $\underline{t}_i$ to express the subproblem

[1] https://assembly-line-balancing.de/sualbsp/data-set-of-scholl-et-al-2013/

relaxation. For the Benders cuts, linear constraints [1] are directly applied. As $\mathcal{I}^j$ is the set of MP variable indices that appear in the j-th subproblem, the corresponding Benders feasibility cut in the MIP form is as follows:

$$\sum_{i \in \mathcal{I}^j} x_{ik} \leq |\mathcal{I}^j| - 1, \; \forall k \in K, \tag{20}$$

where x_{ik} is the decision variable used in the MP MIP formulation and $x_{ik} = 1$ if task i is assigned to station k and 0 otherwise.

5.2 Solving the Subproblem

In the LBBD framework for SUALBP-1, the MP solution assigns tasks to each station. Thus, each subproblem is a constraint satisfaction problem to find a schedule of the tasks, considering the precedence relation between tasks, the sequence-dependent setup times, and the cycle time. The task processing times are not included in the subproblem as they are constant after the task assignment is given; the sum of processing times is therefore subtracted from the cycle time when evaluating feasibility. The subproblem has the structure of the Travelling Salesman Problem (TSP) with precedence constraints. For this constrained TSP variant, our preliminary investigations showed that DIDP outperforms CP and MIP and we hence use DIDP as the sole subproblem solver. The state variables, base cases, and the recursive function are as follows.

State variables. For station j, the DIDP model has the following state variables:

- U: set variable for unscheduled tasks. In the target state, $U = \mathcal{I}^j$.
- s: element variable for the current task, with its value in $\mathcal{I}^j$. In the target state, $s = d_s$, where d_s is a dummy task with setup times from and to any other tasks set to zero.
- f: element variable for the first task, with its value in $\mathcal{I}^j$. In the target state, $f = d_s$.

Base cases. The base case of the DIDP model is: $U = \emptyset \, \wedge \, s = d_s$.

Recursive function. We use $\mathcal{V}(U, s, f)$ to represent the cost of a state. Let P_i^{j*} be the set of predecessors of task i on station j. Let $U_4 = \{i \in \mathcal{I}^j \mid \mathcal{I}^j \cap P_i^{j*} = \emptyset\}$.

$$\texttt{compute } \mathcal{V}(I^j, d_s, d_s) \tag{21a}$$

$$\mathcal{V}(U, s, f) = \begin{cases} 0 & \text{if } U = \emptyset \, \wedge \, s = d_s, & \text{(i)} \\ \mu_{sf} + \mathcal{V}(U, d_s, d_s) & \text{else if } U = \emptyset \, \wedge \, s \neq d_s, & \text{(ii)} \\ \mu_{si} + \min_{i \in U_4} \mathcal{V}(U \backslash \{i\}, i, f) & \text{else if } U_4 \neq \emptyset \, \wedge \, s \neq d_s, & \text{(iii)} \\ \min_{i \in U_4} \mathcal{V}(U \backslash \{i\}, i, i) & \text{else,} & \text{(iv)} \end{cases} \tag{21b}$$

$$\mathcal{V}(U, s, f) \geq \max \begin{cases} \underline{\mu}_f + \sum_{i \in U} \underline{\mathcal{I}}_i, & \text{if } s = d_s, & \text{(i)} \\ 0, & \text{else.} & \text{(ii)} \end{cases} \tag{21c}$$

Case (21b-i) refers to the base case, while (21b-iv) corresponds to assigning the first task to the current empty station. Case (21b-iii) represents assigning the next task to the current station and adding the corresponding setup time. (21b-ii) represents closing the station and adding the setup time to the first task. (21c) is the dual bound [20].

Although this DIDP model is designed for optimization problems, since some DIDP solvers support anytime solving [21], by setting a primal bound, the search can be stopped after a solution satisfying all the constraints and having a total cost no greater than the cycle time minus the total processing time is found.

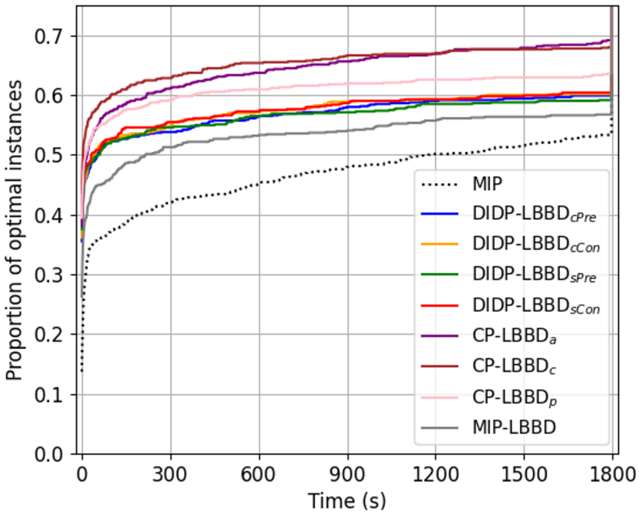


Figure 2 Ratio of instances solved and proved optimal over time for SUALBP-1.

5.3 Experiment Setting

We use the SBF2 data set proposed by Zohali et al. [31] and follow their clustering of the instances into four classes:

- Data set A: small (132 instances) with up to 25 tasks.
- Data set B: medium (140 instances) with 28 to 35 tasks.
- Data set C: large (188 instances) with 45 to 70 tasks.
- Data set D: extra-large (328 instances) with 75 to 111 tasks.

Each class has four different settings according to a parameter α that specifies the ratio of the average setup time to the average task processing time: 0.25, 0.50, 0.75, and 1.00.

For the DIDP models, we use the state-of-the-art solver based on CABS [21] in didp-rs v0.7.3.[2] For the CP models, we use CP Optimizer 22.1.1 [17]. For the MIP models, we use Gurobi 11.0.1 [12]. All the experiments are implemented in Python 3.10.11. Each instance is run for 1800 seconds on a single thread on a Ubuntu 22.04.2 LTS machine with Intel Core i7 CPU and 16 GB memory.

5.4 Experiment Results

The results on SUALBP-1 are shown in Fig. 2.[3] Better performance is indicated by curves closer to the top left corner of the graph. First note that all of our proposed techniques outperform the current state of the art. CP-LBBD$_a$ achieves the best performance at the time limit with 69% of instances proved to optimality. CP-LBBD$_c$ performs best before 1500 seconds. In particular, CP-LBBD$_c$ achieves 63% in 300 seconds while CP-LBBD$_a$ is two times slower to achieve that level. This performance difference indicates the speedup brought by the constraint-based expression `Count_Different`. CP-LBBD$_p$, though trailing the other two CP-LBBD models significantly, performs better than DIDP-LBBD, MIP-LBBD, and MIP approaches. These results imply that direct manipulation of core decision variables x_i in the CP model is advantageous compared to global constraints, especially when using a global constraint requires extra variables such as w_k in the `Pack` constraint.

[2] `https://didp.ai/`

[3] Disaggregated results for datasets A, B, C, and D are presented in Fig. 7-10 in Appendix B.

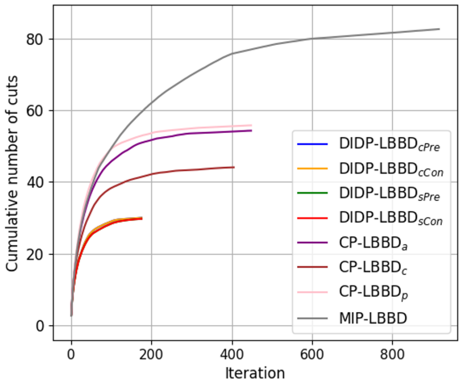


Figure 3 Mean cumulative number of cuts added over iterations.

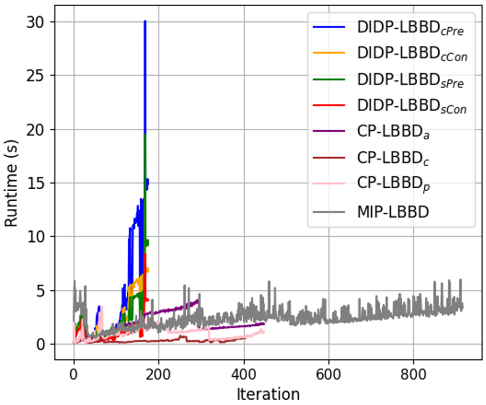


Figure 4 Mean MP runtime over iterations.

The DIDP-LBBD models find and prove optimal solutions for more instances in a shorter computation time than MIP-LBBD and MIP. In 60 seconds, all four DIDP-LBBD models find and prove optimality on 50% of the instances. MIP cannot achieve the same performance in 1100 seconds. At 1800 seconds, DIDP-LBBD has found and proved optimality for around 60% of the problem instances compared to 57% and 54% for MIP-LBBD and MIP, respectively.

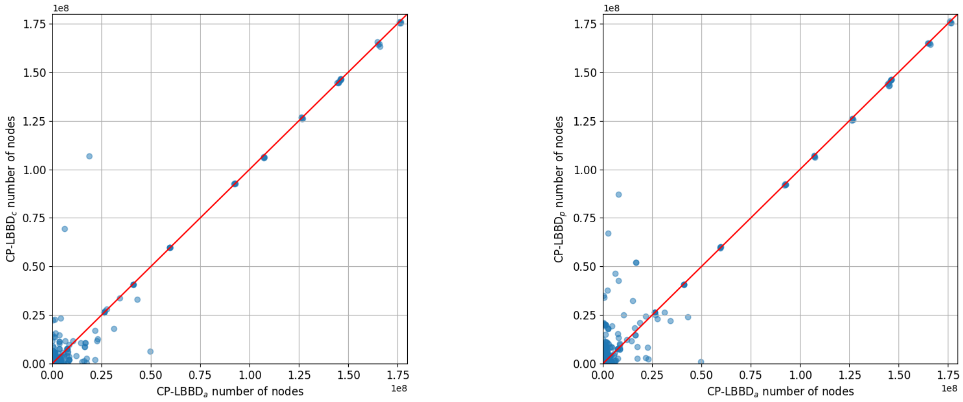
Focusing on the LBBD models, the relative rankings are: CP-LBBD, DIDP-LBBD, and MIP-LBBD, which demonstrates the promise of CP-LBBD and DIDP-LBBD. Though the three CP-LBBD variants differ substantially in Fig. 2, there is no significant performance difference among the four DIDP-LBBD variants. Note that the subproblem solve time is very short, e.g., 0.001s.

5.5 Algorithm Analysis

For the SBF2 data set, 394 of the 788 instances are proved optimal by each of the eight LBBD models. The mean cumulative numbers of cuts added for the 394 instances are shown in Fig. 3.[4] We can see that DIDP-LBBD models have significantly fewer iterations and cuts than CP-LBBD and MIP-LBBD. We believe that this difference is due to the existence of multiple optimal solutions of the master problem: different models find different optimal solutions and different Benders cuts, leading to different numbers of MP runs. While CP and DIDP models require many fewer iterations on average, we found no evidence that this is a systematic difference but rather the arbitrary impact of which optimal solutions are found.

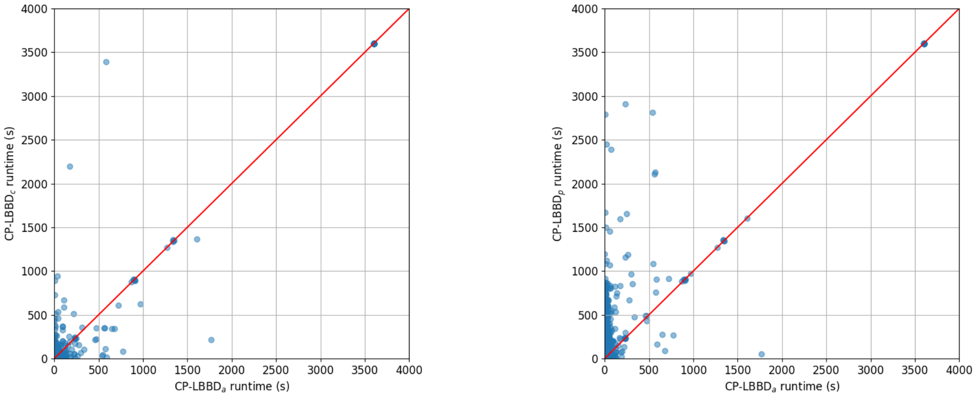
The mean MP runtimes of the 394 instances over iterations for all the eight LBBD models are shown in Fig. 4. CP-LBBD and MIP-LBBD have relatively consistent MP runtime across different iterations. For DIDP-LBBD models, although starting from small magnitude, the MP runtimes increase drastically as the iterations increase. As discussed in Section 4.2.3, with more state variables added to the DIDP model of the master problem, the state space of the model is enlarged and needs more search effort to find and prove optimality, hence the MPs become more time-consuming to solve. This performance degradation can partially explain the worse results of DIDP-LBBD compared to CP-LBBD.

[4] The behaviors of DIDP-LBBD$_{cPre}$ and DIDP-LBBD$_{cCon}$ are exactly the same in terms of cuts added. The behaviors of DIDP-LBBD$_{sPre}$ and DIDP-LBBD$_{sCon}$ are the same, too. Thus, their plots overlap.



(a) CP-LBBD$_c$ and CP-LBBD$_a$. **(b)** CP-LBBD$_p$ and CP-LBBD$_a$.

Figure 5 Number of nodes of the MPs in CP-LBBD models for the SBF2 dataset.



(a) CP-LBBD$_c$ and CP-LBBD$_a$. **(b)** CP-LBBD$_p$ and CP-LBBD$_a$.

Figure 6 Runtime of the MPs in CP-LBBD models for the SBF2 dataset.

In order to investigate the differences among the three CP-LBBD models, for all 788 instances in the SBF2 dataset, we added the cuts generated by CP-LBBD$_a$ model at each MP iteration to all models, in the corresponding cut forms, with a time limit of 3600 seconds. Thus for each MP iteration, the three models solve identical problems except for the differences in the form of the cuts.

Fig. 5 and 6 show scatter plots for the number of nodes and the runtimes. All four graphs show a substantial cluster in the lower-left corner demonstrating broadly similar performance. However, both CP-LBBD$_c$ and, to a greater extent, CP-LBBD$_p$ exhibit a number of instances with a large number of nodes and large runtimes when CP-LBBD$_a$ has relatively small values of these measures.

These graphs are consistent with the overall results of the CP models in Figure 2. In terms of the number of nodes generated, the graphs suggest that the difference comes less from a systematic performance difference among the models and more from a small number of outliers with large node counts for CP-LBBD$_c$ and CP-LBBD$_p$. In contrast, the runtime graphs for CP-LBBD$_p$ and, to a lesser extent, CP-LBBD$_c$ show vertical clusters of instances with relatively low CP-LBBD$_a$ runtimes implying that the higher computational effort of the global constraint based models does not pay off in terms of performance.

A different perspective on the results in Fig. 5 and 6, is shown by the runtime vs. number of nodes of the MPs in three CP-LBBD models in Fig. 11 in Appendix C. Since the three models solve identical problems except for cut forms, the results reflect the runtime each of the three CP-LBBD models needs for exploring the same number of nodes and also coincide with the performance rankings of CP-LBBD models from a regression perspective.

6 Discussion

Global constraints in CP can increase domain propagation and the overall solving performance but have a limit, after which the improved propagation, if any, is not worth the effort required [24]. This dynamic may be observed by the worse results of CP-LBBD$_p$ compared to CP-LBBD$_a$ and CP-LBBD$_c$. By contrast, CP-LBBD$_a$ and CP-LBBD$_c$ manipulate the main decision variables more directly while not inducing much larger constraint models.

The validity of the proposed four DIDP cut encoding methods depends on the effective extraction of the useful information, i.e., the change of the variable-value pairs in the Benders cuts. Such information is often hidden in the transitions of DIDP models. Thus, it is difficult to create a cut encoding using the existing state variables. An important question is to understand if this state-space expansion is an inherent weakness for DIDP and, indeed, state-based models in general. There exists similar work examining the addition of trajectory constraints to AI planning problems which similarly expand the state space [16, 3].

In a parallel work, a monolithic DIDP model for SUALBP-1 performs better than all the LBBD models presented here [30]. This is a surprising result as the state of the art for similar problems with sequence-dependent setup times is typically based on decomposition [31, 28]. Further research is required to understand why DIDP models for SUALBP-1 do not follow this pattern. We speculate that the relaxation of the setup time in the MP hurts performance compared to the monolithic DIDP model because setup time can be directly accounted for in the transitions.

7 Conclusions

In this paper, we proposed novel logic-based Benders decomposition (LBBD) models with master problems modeled and solved with constraint programming (CP) and domain-independent dynamic programming (DIDP), using simple assembly line balancing problem with sequence-dependent setup times type-1 (SUALBP-1) as a testbed. We developed three CP-based master problem formulations with Benders feasibility cuts formulated as key variable manipulation, constraint-based expressions, and global constraints. In the state transition system of DIDP, we proposed four encoding methods for Benders feasibility cuts by exploiting the integer or set variables and preconditions or state constraints. Experimental results on SUALBP-1 show superior performance for the CP-LBBD models and good performance of the four DIDP-LBBD models, compared to MIP-LBBD and monolithic MIP models. This work demonstrates the promise of decomposition-based approaches employing CP and DIDP approaches.

References

1 Sener Akpinar, Atabak Elmi, and Tolga Bektaş. Combinatorial benders cuts for assembly line balancing problems with setups. *European Journal of Operational Research*, 259(2):527–537, 2017.

2 Carlos Andres, Cristobal Miralles, and Rafael Pastor. Balancing and scheduling tasks in assembly lines with sequence-dependent setup times. *European Journal of Operational Research*, 187(3):1212–1223, 2008.

3 Jorge A Baier, Fahiem Bacchus, and Sheila A McIlraith. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence*, 173(5-6):593–618, 2009.

4 Ilker Baybars. A survey of exact algorithms for the simple assembly line balancing problem. *Management science*, 32(8):909–932, 1986.

5 Christian Becker and Armin Scholl. A survey on problems and methods in generalized assembly line balancing. *European journal of operational research*, 168(3):694–715, 2006.

6 Yossi Bukchin and Tal Raviv. Constraint programming for solving various assembly line balancing problems. *Omega*, 78:57–68, 2018.

7 Yingyi Chu and Quanshi Xia. Generating benders cuts for a general class of integer programming problems. In Jean-Charles Régin and Michel Rueher, editors, *Proceedings of the First International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2004)*, volume 3011, pages 127–136. Springer, Berlin Heidelberg, 2004.

8 Maryam Daryalal, Hamed Pouya, and Marc Antoine DeSantis. Network migration problem: A hybrid logic-based benders decomposition approach. *INFORMS Journal on Computing*, 2023.

9 Rasul Esmaeilbeigi, Bahman Naderi, and Parisa Charkhgard. New formulations for the setup assembly line balancing and scheduling problem. *OR spectrum*, 38:493–518, 2016.

10 Michael Forbes, Mitchell Harris, Marijn Jansen, Femke van der Schoot, and Thomas Taimre. Combining optimisation and simulation using logic-based benders decomposition. *arXiv preprint arXiv:2107.08390*, 2021.

11 Cheng Guo, Merve Bodur, Dionne M Aleman, and David R Urbach. Logic-based benders decomposition and binary decision diagram based approaches for stochastic distributed operating room scheduling. *INFORMS Journal on Computing*, 33(4):1551–1569, 2021.

12 LLC Gurobi Optimization. Gurobi optimizer reference manual, 2021. Accessed on 2024-04-10. URL: `http://www.gurobi.com`.

13 John Hooker. *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*. John Wiley & Sons, Inc., New York, 2000.

14 John N Hooker. Planning and scheduling by logic-based benders decomposition. *Operations research*, 55(3):588–602, 2007.

15 John N Hooker and Greger Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96(1):33–60, 2003.

16 Chih-Wei Hsu, Benjamin W Wah, Ruoyun Huang, and Yixin Chen. Constraint partitioning for solving planning problems with trajectory constraints and goal preferences. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 1924–1929, 2007.

17 IBM. IBM ILOG CPLEX Optimizer. Accessed on 2024-04-20. URL: `https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-cp-optimizer`.

18 Naveen Kumar and Dalgobind Mahto. Assembly line balancing: a review of developments and trends in approach to industrial application. *Global Journal of Researches in Engineering Industrial Engineering*, 13(2):29–50, 2013.

19 Ryo Kuroiwa and J. C. Beck. Domain-independent dynamic programming. *arXiv preprint arXiv:2401.13883*, 2024.

20 Ryo Kuroiwa and J Christopher Beck. Domain-independent dynamic programming: Generic state space search for combinatorial optimization. In *the 33rd International Conference on Automated Planning and Scheduling (ICAPS), 236–244.*, 2023.

21 Ryo Kuroiwa and J Christopher Beck. Solving domain-independent dynamic programming problems with anytime heuristic search. In *the 33rd International Conference on Automated Planning and Scheduling (ICAPS), 245–253.*, 2023.

22 Florin Leutwiler and Francesco Corman. A logic-based benders decomposition for microscopic railway timetable planning. *European Journal of Operational Research*, 303(2):525–540, 2022.

23 Marcus Ritt and Alysson M Costa. Improved integer programming models for simple assembly line balancing and related problems. *International Transactions in Operational Research*, 25(4):1345–1359, 2018.

24 Francesca Rossi, Peter Van Beek, and Toby Walsh. Constraint programming. *Foundations of Artificial Intelligence*, 3:181–211, 2008.

25 Armin Scholl, Nils Boysen, and Malte Fliedner. The assembly line balancing and scheduling problem with sequence-dependent setup times: problem extension, model formulation and efficient heuristics. *OR spectrum*, 35:291–320, 2013.

26 Armin Scholl and Robert Klein. Salome: A bidirectional branch-and-bound procedure for assembly line balancing. *INFORMS journal on Computing*, 9(4):319–334, 1997.

27 Paul Shaw. A constraint for bin packing. In *International conference on principles and practice of constraint programming*, pages 648–662. Springer, 2004.

28 Tony T Tran, Arthur Araujo, and J Christopher Beck. Decomposition methods for the parallel machine scheduling problem with setups. *INFORMS Journal on Computing*, 28(1):83–95, 2016.

29 Tony T Tran and J Christopher Beck. Logic-based benders decomposition for alternative resource scheduling with sequence dependent setups. In *ECAI 2012*, pages 774–779. IOS Press, 2012.

30 Jiachen Zhang and J. C. Beck. Domain-independent dynamic programming and constraint programming approaches for assembly line balancing problems with setups. *arXiv preprint arXiv:2403.06780*, 2024.

31 Hassan Zohali, Bahman Naderi, and Vahid Roshanaei. Solving the type-2 assembly line balancing with setups using logic-based benders decomposition. *INFORMS Journal on Computing*, 34(1):315–332, 2022.

A Monolithic MIP Model of SUALBP-1

Table 3 Additional parameters for SUALBP-1 [9].

Notation	Definition
$\mathcal{E}$	set of all precedence relations
E_i	earliest station for task $i \in V$, e.g., $E_i = \lceil \frac{t_i + \sum_{j \in P_i^*} t_j}{\bar{c}} \rceil$
L_i	latest station for task $i \in V$, e.g., $L_i = \overline{m} + 1 - \lceil \frac{t_i + \sum_{j \in F_i^*} t_j}{\bar{c}} \rceil$
$KD(KP)$	set of definite (possible) stations, i.e., $KD = \{1, ..., \underline{m}\}$, $KP = \{\underline{m}+1, ..., \overline{m}\}$, and $K = KD \cup KP$
FS_i	set of stations to which task $i \in V$ can be assigned, i.e., $FS_i = \{E_i, E_i + 1, ..., L_i\}$
FT_k	set of tasks which can be assigned to station $k \in K$, i.e., $FT_k = \{i \in V \vert k \in FS_i\}$
A_i	set of tasks that cannot be assigned to the station to which task i is assigned, e.g., $A_i = \{j \in V \vert FS_j \cap FS_i = \emptyset\}$
$F_i^F(P_i^F)$	set of tasks which may directly follow (precede) task i in forward direction, i.e., $F_i^F = \{j \in V - (F_i^* - F_i) - P_i^* - A_i - \{i\}\}$ and $P_i^F = \{j \in V \vert i \in F_j^F\}$
$F_i^B(P_i^B)$	set of tasks which may directly follow (precede) task i in backward direction, i.e., $F_i^B = \{j \in V - F_i^* - A_i\}$ and $P_i^B = \{j \in V \vert i \in F_j^B\}$

To present the monolithic MIP model of SUALBP-1, additional parameters are required, as shown in Table 3. Since the SSBF model can be adapted to both SUALBP-1 and SUALBP-2, we name it SSBF-1 [9]. The decision variables are:

- x_{ik}: binary variable with value 1, iff task $i \in V$ is assigned to station $k \in FS_i$.
- z_i: integer variable for encoding the index of the station task $i \in V$ is assigned to.
- u_k: binary variable with value 1, iff any task is assigned to station k.
- g_{ijk}: binary variable $= 1$, iff task i is performed immediately before task j on station k.
- h_{ijk}: binary variable $= 1$, iff task i is the last and task j is the first task on station k.
- r_i: integer variable representing the rank of task i in a sequence of all tasks. The sequence is composed of the task sequences on all the active stations.

The SSBF-1 MIP model proposed by Esmaeilbeigi et al. [9] is as follows.

$$\min \sum_{k \in KP} u_k + \underline{m} \tag{22a}$$

$$\text{s.t.} \sum_{k \in FS_i} x_{ik} = 1, \quad \forall i \in V, \tag{22b}$$

$$\sum_{k \in FS_i} k \cdot x_{ik} = z_i, \quad \forall i \in V, \tag{22c}$$

$$\sum_{i \in FT_k \cap F_i^F} g_{ijk} + \sum_{i \in FT_k \cap F_i^B} h_{ijk} = x_{ik}, \quad \forall i \in V, \forall k \in FS_i, \tag{22d}$$

$$\sum_{i \in FT_k \cap P_j^F} g_{ijk} + \sum_{i \in FT_k \cap P_j^B} h_{ijk} = x_{jk}, \quad \forall j \in V, \forall k \in FS_j, \tag{22e}$$

$$\sum_{i \in FT_k} \sum_{j \in (FT_k \cap F_i^B)} h_{ijk} = 1, \quad \forall k \in KD, \tag{22f}$$

$$\sum_{i \in FT_k} \sum_{j \in (FT_k \cap F_i^B)} h_{ijk} = u_k, \quad \forall k \in KP, \tag{22g}$$

$$r_i + 1 + (n - |F_i^*| - |P_j^*|) \cdot (\sum_{k \in (FS_i \cap FS_j)} g_{ijk} - 1) \leq r_j, \quad \forall i \in V, \forall j \in F_i^F, \tag{22h}$$

$$r_i + 1 \leq r_j, \quad \forall (i,j) \in \mathcal{E}, \tag{22i}$$

$$z_i \leq z_j, \quad \forall (i,j) \in \mathcal{E}, \tag{22j}$$

$$\sum_{i \in FT_k} t_i x_{ik} + \sum_{i \in FT_k} \sum_{j \in (FT_k \cap F_i^F)} \tau_{ij} g_{ijk} + \sum_{i \in FT_k \cap P_i^B} \mu_{ij} h_{ijk} \leq c, \quad \forall k \in KD, \tag{22k}$$

$$\sum_{i \in FT_k} t_i x_{ik} + \sum_{i \in FT_k} \sum_{j \in (FT_k \cap F_i^F)} \tau_{ij} g_{ijk} + \sum_{i \in FT_k \cap P_i^B} \mu_{ij} h_{ijk} \leq c \cdot u_k, \quad \forall k \in KP, \tag{22l}$$

$$\sum_{i \in FT_k \backslash \{j\}} x_{ik} \leq (n - \underline{m} + 1) \cdot (1 - h_{jjk}), \quad \forall k \in K, \forall j \in FT_k, \tag{22m}$$

$$u_{k+1} \leq u_k, \quad \forall k \in KP \backslash \{\overline{m}\}. \tag{22n}$$

$$g_{ijk} \in \{0,1\}, \quad \forall k \in K, \forall i \in FT_k, \forall j \in (FT_k \cap F_i^F), \tag{22o}$$

$$h_{ijk} \in \{0,1\}, \quad \forall k \in K, \forall i \in FT_k, \forall j \in (FT_k \cap F_i^B), \tag{22p}$$

$$|P_i^*| + 1 \leq r_i \leq n - |F_i^*|, \quad \forall i \in V, \tag{22q}$$

$$x_{ik} \in \{0,1\}, \quad \forall i \in V, \forall k \in FS_i, \tag{22r}$$

$$r_i, z_i \in \mathbb{Z}^+, \quad \forall i \in V, \tag{22s}$$

The objective (22a) minimizes the number of stations. Constraint (22b) ensures that a task is assigned to a station. Constraint (22c) links x_{ik} and z_i. Constraints (22d) and (22e) assure that a task on station k is followed and preceded by exactly one other task in the cyclic sequence of this station. According to constraints (22f) and (22g), in each cycle exactly one of the relations is a backward setup. Constraints (22h) and (22i) establish the precedence relations among the tasks within each station. Note that the constraint (22h) is inactive if tasks i and j are assigned to different stations. We add the constraint (22j) to make sure that the precedence relations among the tasks of different stations are satisfied. Knapsack constraints (22k) and (22l) ensure that no station time exceeds the cycle time. Constraint (22m) guarantees that only task j is allocated to station k when $h_{jjk} = 1$. Constraint (22n) guarantees that stations are used in the correct order and no empty station is in the middle of used stations. Constraints (22o) to (22s) specify the domain of the variables.

Note that the decision variables r_i and z_i are set to continuous in [9]. However, doing so results in infeasible solutions being labeled as feasible for some problem instances. In addition to the MIP model, Esmaeilbeigi et al. [9] developed pre-processing techniques to reduce the number of variables and constraints. We implement all these techniques, as well.

B Approach Performances for Separate Datasets

The performance of each approach on datasets A, B, C, and D separately are presented in Fig. 7 - 10, respectively. As shown in Fig. 7, all approaches except MIP solve all problems in dataset A to proved optimality in a few seconds. For dataset B (Fig. 8), all approaches, including MIP, are competitive and behave similarly. For dataset C, MIP-LBBD has the worst performance while surprisingly it outperforms all DIDP-LBBD approaches and MIP for dataset D, as shown in Fig. 9 and 10. We can also see the performance degradation of DIDP-LBBD when solving larger problems.

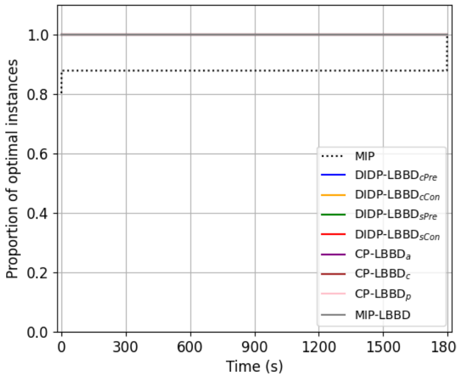


Figure 7 Ratio of instances solved and proved optimal over time for dataset A.

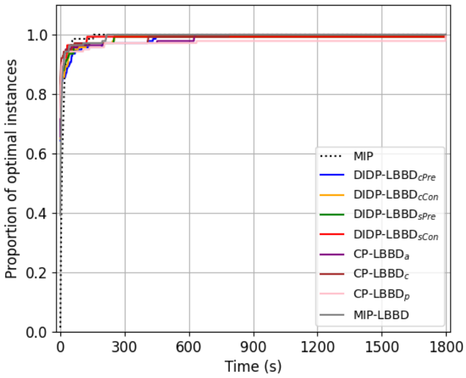


Figure 8 Ratio of instances solved and proved optimal over time for dataset B.

C Analysis of CP-LBBD

In Section 5.5, for all 788 instances in the SBF2 dataset, we added the cuts generated by CP-LBBD$_a$ model at each MP iteration to all models, in the corresponding cut forms, with a time limit of 3600 seconds. The runtime over the number of nodes of the MPs in CP-LBBD

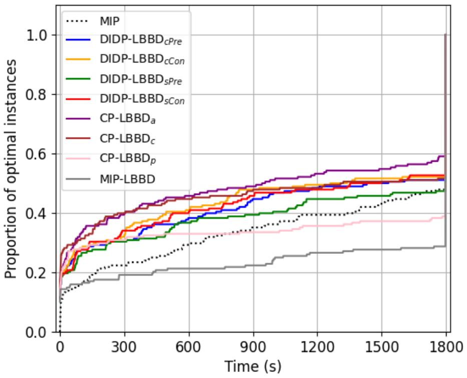


Figure 9 Ratio of instances solved and proved optimal over time for dataset C.

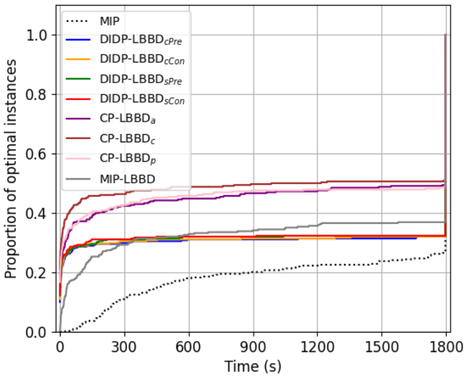


Figure 10 Ratio of instances solved and proved optimal over time for dataset D.

models for the SBF2 dataset is shown in Fig. 11. The regression lines demonstrate the performance rankings of the three CP-LBBD models in terms of the runtime required to explore the same number of nodes.

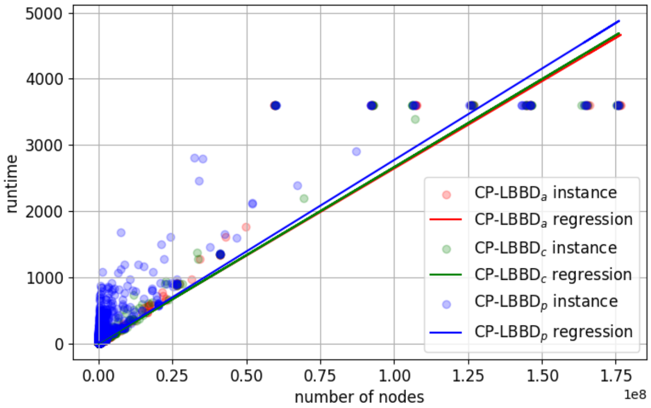


Figure 11 Runtime vs. number of nodes of the MPs in CP-LBBD models for the SBF2 dataset.

Ex-Ante Constraint Elicitation in Incomplete DCOPs

Roie Zivan
Ben-Gurion University of the Negev, Beer-Sheva, Israel

Shiraz Regev
Ben-Gurion University of the Negev, Beer-Sheva, Israel

William Yeoh
Washington University in St. Louis, MO, USA

Abstract

Distributed Constraint Optimization Problems (DCOPs) is a framework for representing and solving distributed combinatorial problems, where agents exchange messages to assign variables they own, such that the sum of constraint costs is minimized. When agents represent people (e.g., in meeting scheduling problems), the constraint information that the agents hold may be incomplete. For such scenarios, researchers proposed Incomplete DCOPs (I-DCOPs), which allow agents to elicit from their human users some of the missing information. Existing I-DCOP approaches evaluate solutions not only by their quality, but also the elicitation costs spent to find them (*ex-post*). Unfortunately, this may result in the agents spending a lot of effort (in terms of elicitation costs) to find high-quality solutions, and then ignoring them because previous lower-quality solutions were found with less effort.

Therefore, we propose a different approach for solving I-DCOPs by evaluating solutions based on their quality and considering the elicitation cost beforehand (*ex-ante*). Agents are limited in the amount of information that they can elicit and, therefore, need to make smart decisions on choosing which missing information to elicit. We propose several heuristics for making these decisions. Our results indicate that some of the heuristics designed produce high-quality solutions, which significantly outperform the previously proposed ex-post heuristics.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed Constraint Optimization Problems, Preference Elicitation, Multi-Agent Optimization

Digital Object Identifier 10.4230/LIPIcs.CP.2024.33

Funding This research is partially supported by US-Israel Binational Science Foundation (BSF) grant #2022189 and by a J.P. Morgan Faculty Research Award.

1 Introduction

The Distributed Constraint Optimization Problem (DCOP) formulation is widely used for representing and solving combinatorial optimization problems that are distributed by nature [5, 7, 15]. It includes agents holding variables, which are constrained with variables held by other agents (their neighbors) and attempt to find an optimal assignment to their variables that minimizes constraint costs, while exchanging messages with their neighbors.

When agents represent humans, such as in meeting scheduling problems [4, 1], the information held by agents regarding the preferences of the humans that they represent may be incomplete. Agents can elicit information from the humans by introducing queries to their human users, However, humans might find that answering these queries is a tedious task and may abandon the use of the system if the burden is too heavy. Thus, there is a clear need to limit the amount of queries that the human users need to answer.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 33; pp. 33:1–33:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In order to represent such situations and allow agents to select high-quality assignments to their variables, while taking into consideration incomplete information, and make a limited use of elicitation queries, the Incomplete DCOP (I-DCOP) model was proposed [10, 11, 12]. I-DCOP enables the representation of partial information by having agents hold constraint tables in which some entries include the costs for the corresponding combination of assignments and some do not. The agents can use elicitation queries to fill some of the empty entries and then use the information available to them in order to select the solution to the problem.

Tabakhi *et al.* [11, 12] proposed limiting the use of elicitation queries by evaluating the outcome of the I-DCOP solving process as a weighted sum of the quality of the selected solution and the effort (e.g., number of queries asked) for producing it. Thus, the agents aimed to find a solution that has not only a high quality, but also a low effort to find it. While this evaluation of outcomes incentivizes the algorithm to make efficient use of the human query resources, from a practical point of view, this method for evaluating possible outcomes does not make sense.

For example, imagine that an agent is searching for a hotel for the next trip of the person it represents. After a small search effort c_1, the agent finds a decent hotel with solution quality q_1. Then, the agent decides to spend more effort, searching for a better hotel and, after a costly effort $c_2 >> c_1$, it manages to find one that is slightly better $q_2 > q_1$. According to the evaluation method proposed [11, 12], the agent will choose the first hotel because $c_1 - q_1 < c_2 - q_2$.[1] In other words, the second hotel is not as good because the marginal increase in quality is not worth the large amount of effort spent for it. However, intuitively, since the search effort was already spent, it does not make sense to not use the better solution found.

The key issue with the prior approach is that the search effort considered is done ex-post – *after* the effort was spent – when it should be done ex-ante – *before* the effort was spent. With this insight in mind, we propose a different approach for solving incomplete DCOPs. Inspired by others [3], we limit the amount of queries that agents can use (i.e., a query "budget") and propose different heuristic strategies for the agents to follow when they decide what information to elicit. We compare the success of the proposed strategies in comparison with the existing ex-post approach, in combination with a complete SyncBB algorithm [2] and two incomplete DSA and MGM [15, 17] algorithms.

Our results indicate that all the ex-ante heuristic strategies we proposed outperformed the existing ex-post heuristic. Moreover, the heuristics that spend effort in identifying parts of the search space that have higher probability to be part of a high-quality solution are more successful.

2 Background

In this section, we present DCOPs and three algorithms for solving them: SyncBB, DSA, and MGM.

2.1 Distributed Constraint Optimization Problems

Without loss of generality, in the rest of this paper, we will assume that all problems are minimization problems, as it is common in the DCOP literature [1]. Thus, we assume that all constraints define costs and not utilities.

[1] We assume that we are minimizing costs in this paper.

A DCOP is defined by a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$. $\mathcal{A}$ is a finite set of agents $\{A_1, A_2, \ldots, A_n\}$. $\mathcal{X}$ is a finite set of variables $\{X_1, X_2, \ldots, X_m\}$. Each variable is held by a single agent, and an agent may hold more than one variable. $\mathcal{D}$ is a set of domains $\{D_1, D_2, \ldots, D_m\}$. Each domain D_i contains the finite set of values that can be assigned to variable X_i. We denote an assignment of value $x \in D_i$ to X_i by an ordered pair $\langle X_i, x \rangle$. $\mathcal{R}$ is a set of relations (constraints). Each constraint $R_j \in \mathcal{R}$ defines a non-negative *cost* for every possible value combination of a set of variables, and is of the form $R_j : D_{j_1} \times D_{j_2} \times \ldots \times D_{j_k} \to \mathbb{R}^+ \cup \{0\}$. A *binary constraint* refers to exactly two variables and is of the form $R_{ij} : D_i \times D_j \to \mathbb{R}^+ \cup \{0\}$. We say that a variable is *involved* in a constraint if it is one of the variables the constraint refers to and that an agent is involved in a constraint if one of its variables is involved in the constraint. We assume that agents hold all constraints that they are involved in. For each binary constraint R_{ij}, there is a corresponding cost table T_{ij} with dimensions $|D_i| \times |D_j|$ in which the cost in every entry e_{xy} is the cost incurred when x is assigned to X_i and y is assigned to X_j. A *binary DCOP* is a DCOP in which all constraints are binary. A *partial assignment* is a set of value assignments to variables, in which each variable appears at most once. $vars(PA)$ is the set of all variables that appear in partial assignment PA (i.e., $vars(PA) = \{X_i \mid \exists x \in D_i \wedge \langle X_i, x \rangle \in PA\}$). A constraint $R_j \in \mathcal{R}$ of the form $R_j : D_{j_1} \times D_{j_2} \times \ldots \times D_{j_k} \to \mathbb{R}^+ \cup \{0\}$ is *applicable* to PA if each of the variables $X_{j_1}, X_{j_2}, \ldots, X_{j_k}$ is included in $vars(PA)$. The set of constraints that are applicable to a partial assignment PA will be denoted by R_{PA}. $R_j(PA)$ is the cost of incurred which corresponds to R_j with respect to PA. When R_j does not apply to (PA), $R_j(PA) = 0$. The *cost of a partial assignment* $C(PA)$ is the sum of costs of all constraints that are applicable to PA, i.e., $C(PA) = \sum_{R_j \in R_{PA}} R_j(PA)$. A *complete assignment* (or a *solution*) is a partial assignment that includes all the DCOP's variables (i.e., $vars(PA) = \mathcal{X}$). An *optimal solution* is a complete assignment with minimal cost.

For simplicity, we make the common assumption that each agent holds exactly one variable (i.e., $n = m$) and we concentrate on binary DCOPs. These assumptions are common in the DCOP literature [7, 13]. That being said, we emphasize that all methods and heuristics we propose in this paper apply to k-ary constraints as well, for $2 \leq k \leq n$.

2.2 Synchronous Branch-and-Bound (SyncBB)

Synchronous Branch-and-Bound (SyncBB) [2] is a complete, synchronous, search-based algorithm that can be considered as a distributed version of a standard branch-and-bound algorithm. It uses a complete ordering of the agents to extend a *Current Partial Assignment* (CPA) via a synchronous communication process. The CPA is exchanged by the agents according to the order. Agents add the assignments to their variables before sending the CPA forward and remove their assignments before sending it backwards. The CPA also functions as a mechanism to propagate bound information. The algorithm prunes those parts of the search space whose solution quality is sub-optimal by exploiting the bounds that are updated at each step of the algorithm. In other words, an agent backtracks when the cost of the CPA is not smaller than the cost of the best complete solution found so far.

The algorithm begins by the first agent in the order, which generates the CPA, assigns it a value and forwards it to the next in the order. The CPA includes a lower bound, which is the current cost of the partial assignment carried by the CPA and an upper bound (UB), which is the cost of the best solution found so far by the algorithm (initially infinity), When an agent A_i receives a CPA, it attempts to assign its variable X_i with one of the values $x \in D_i$ and send it forward. When it is received back from the agent following it in the order (A_{i+1}, it attempts to reassign X_i with a different value from D_i. A CPA is sent back when the agent

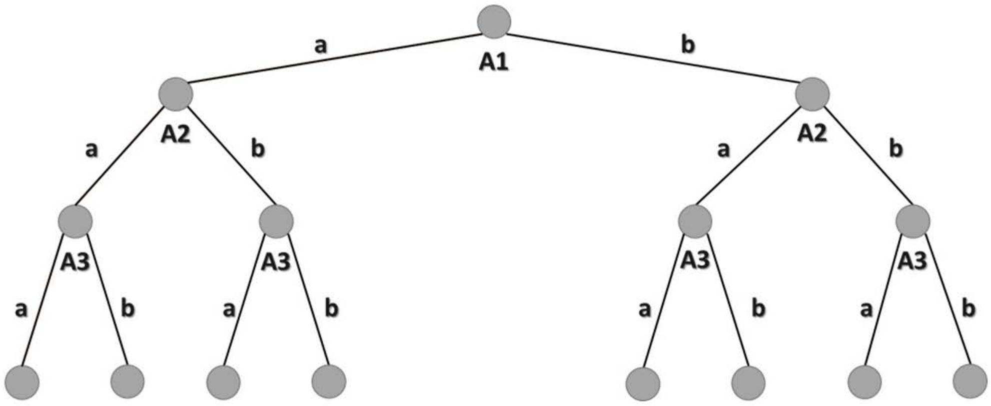


Figure 1 Example of a search tree.

cannot assign a value to the CPA that has not been assigned to the CPA with the specific context (the partial assignment) before, or that does not cause a breach of UB. When the last agent in the order manages to assign its variable, without breaching UB, a new solution is generated and stored, and UB is updated with its cost. The algorithm terminates when the first agent sends the CPA back. The solution reported is the last complete assignment (solution) that caused an update of UB.

In order to analyze the performance of complete search algorithms, such as SyncBB, when solving constraint reasoning problems, such as DCOPs, it is common to use a search tree. The search tree is a tool that allows one to follow the advancement of the search process and analyze its properties. The root of the search tree is the first variable in the order, and each of the edges connecting it to its children represents a possible value assignment. Similarly the second layer represents the possible assignments of the second variable in the order and so forth, until the leaves of the tree, which represent the value assignments of the last variable in the order [14]. Thus, each value assignment is the root of a sub-tree in this search tree.

Figure 1 presents an example of a search tree with three agents A_1, A_2, and A_3, each holding one variable with two values in its domain a and b.

2.3 Distributed Stochastic Algorithm (DSA)

The Distributed Stochastic Algorithm (DSA) [15] is a simple distributed local search algorithm in which, following an initial step where agents (randomly) choose an initial value for their variable, the agents perform a series of steps (looped iteratively) until some termination condition is met. In every step, an agent sends its value assignment to its neighbors in the constraint graph and collects the value assignments of its neighbors. Once the value assignments of all its neighbors have been collected, an agent decides whether to keep its value assignment or to modify it. This decision has a significant effect on the performance of the algorithm. If an agent in DSA cannot upgrade its current state by substituting its present value, it does not do so. On the other hand, if the agent can improve (or maintain, depending on the version used) its current state, it decides whether to replace its value assignment using a stochastic strategy.

2.4 Maximum Gain Message (MGM)

Like DSA, Maximum Gain Message (MGM) is a distributed synchronous local search algorithm, in which agents perform in iterations. In each iteration the agents send messages to all their neighbors, receive messages from all of them and perform computation. The main difference from DSA is that, for each decision whether to replace an assignment, two iterations are performed. In the first, like in DSA, the agents exchange their value assignments. In the second, the agents exchange the maximal improvement they can achieve by replacing assignments. Only agents that suggested a positive improvement that is greater than all their neighbors (ties are broken deterministically according to the agents' identifying indexes), replace their assignments.

3 Ex-Ante Incomplete DCOP

An Ex-Ante Incomplete DCOP (EAI-DCOP) is defined by a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R}, \tilde{\mathcal{R}}, \mathcal{E}, \mathcal{B} \rangle$, where $\mathcal{A}$, $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{R}$ are defined the same as in DCOP. For each constraint $R_j \in \mathcal{R}$, there is a corresponding incomplete constraint $\tilde{R}_j \in \tilde{\mathcal{R}}$, where $\tilde{R}_j \in \tilde{\mathcal{R}} : D_{j_1} \times D_{j_2} \times \ldots \times D_{j_k} \to \mathbb{R}^+ \cup \{0, ?\}$, where each of the D_{j_q} in $\tilde{R}_j$ is also a member in R_j and ? is a special element denoting that the cost for a given combination of value assignments is not known to the agent. In I-DCOP, it is assumed that an agent does not hold the set of constraints that it is involved in, but rather the set of incomplete constraints that it is involved in.

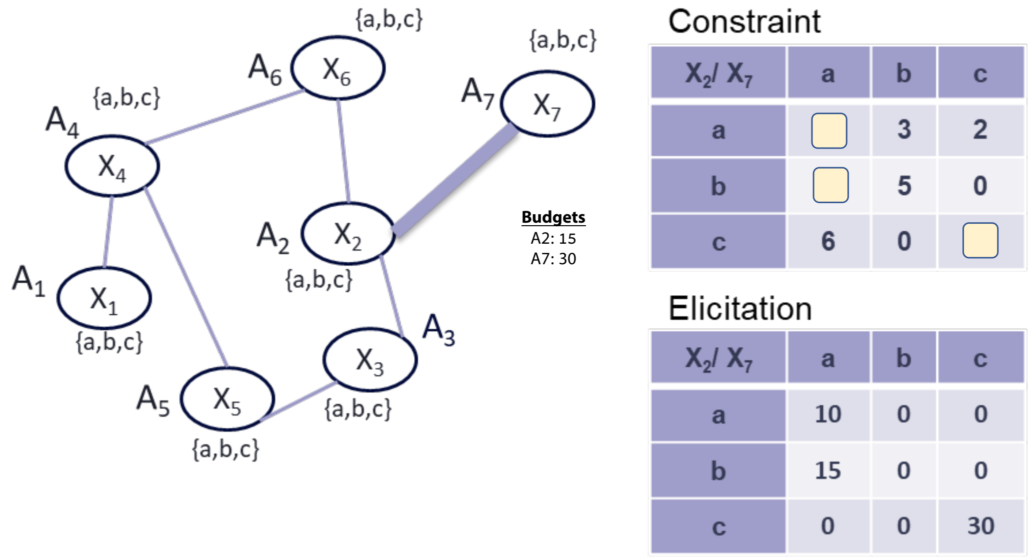
For every incomplete constraint $\tilde{R}_j$, there is an elicitation cost function $E_j \in \mathcal{E}$, such that for each unknown cost of a combination of assignments $r \in R_j$ there is a positive elicitation cost in $e(r) \in E_j$ that the agent will need to "pay" for eliciting this constraint. An explored solution space $\tilde{\mathbf{x}}$ is the union of all solutions explored so far by a particular algorithm. $\tilde{\mathbf{x}}_{PA}$ is the explored solution space at the time that PA was generated. The cumulative elicitation cost $\mathcal{E}(\tilde{\mathbf{x}})$ (and $\mathcal{E}(\tilde{\mathbf{x}}_{PA})$ respectively) is $\sum_{r \in R} e(r)$ such that r is an unknown constraint in $\tilde{\mathcal{R}}$, but it is not an unknown constraint in $\tilde{\mathbf{x}}$. In other words, it is the sum of the elicitation costs of all elicitation queries conducted while exploring $\tilde{\mathbf{x}}$.

In standard (Ex-Post) I-DCOP [12], the cost $C(PA)$ of a partial assignment is calculated as follows: $C(PA) = \sum_{R_j \in R_{PA}} C(R_j) + \mathcal{E}(\tilde{\mathbf{x}}_{PA})$, where R_{PA} is the set of constraints whose variables are in $vars(PA)$. In an Ex-Ante I-DCOP, the solution cost, like in standard DCOP, is $C(PA) = \sum_{R_j \in R_{PA}} C(R_j)$. However, agents are limited in the amount of information they can elicit. We formulate this limitation using a budget $\mathcal{B} = \{B_1, B_2, ..., B_n\}$, where B_i is the amount of elicitation cost agent A_i may spend. These are taken into consideration during the search process and, thus, the agents take the budget limitations into consideration **before** they decide whether to elicit some information.

Figure 2 includes an example of an EAI-DCOP with seven agents. Each agent holds one variable with three values in its domain, and has an elicitation budget (we only present the budgets of A_2 and A_7, which are relevant to the example). On the right hand side the cost table and the elicitation cost table of constraint $R_{2,7}$ are presented. There are three unknown costs in the cost table. Agent A_2's budget allows it to elicit the cost for $\langle X_2 = a, X_7 = a \rangle$ or $\langle X_2 = b, X_7 = a \rangle$. Agent X_7 can afford to elicit any of the three missing costs, and even to elicit the costs for both $\langle X_2 = a, X_7 = a \rangle$ and $\langle X_2 = b, X_7 = a \rangle$.

4 Solving EAI-DCOPs

We propose ex-ante elicitation heuristics for three algorithms solving EAI-DCOPs – SyncBB [2], DSA [15], and MGM [4]. These well-known algorithms were selected for their simplicity, in order to emphasize the effect of the selected elicitation heuristic on the search process. SyncBB and MGM were also used in the previous I-DCOP studies [11, 12].



Constraint

X_2/ X_7	a	b	c
a	□	3	2
b	□	5	0
c	6	0	□

Elicitation

X_2/ X_7	a	b	c
a	10	0	0
b	15	0	0
c	0	0	30

Figure 2 Example of an EAI-DCOP.

4.1 Solving EAI-DCOPs with SyncBB

The main difference between agents performing SyncBB to solve EAI-DCOPs from the agents performing SyncBB to solve standard DCOPs is that, in EAI-DCOPs, agents do not attempt to assign all values to their variables. Instead, when an agent that has partial information regarding the constraint costs of its variable receives a CPA, it needs to decide whether to elicit missing information and which missing information to elicit. We will assume that regardless of the heuristic being used, an agent will first attempt to assign values to its variable, for which it knows all costs of constraints with the value assignments included in the CPA. For the values in its domain for which it does not know all the constraint costs, the agent can decide either to elicit this information, and pay the corresponding cost (which is deducted from its budget), or to avoid eliciting this information. Obviously, if its budget is smaller than the elicitation cost, the first option is ruled out. After eliciting the constraint costs corresponding to a value $x \in D_i$, agent A_i treats x as any other value in its domain for which it has complete knowledge regarding its constraints, that is, it tries to assign x to X_i and send the CPA forward. On the other hand, if A_i decides not to elicit the costs that correspond to x, the subtree rooted by x (in the search tree) will not be explored. We propose the following heuristics for deciding whether to elicit the cost information by agents solving EAI-DCOPs with SyncBB.

Depth Dependent (DD). The decision whether to elicit the cost information for a value is decided stochastically. The probability for an agent A_i to elicit the costs of a value in its domain is calculated using the Sigmoid function: $p(i) = \frac{e^i}{e^{n/2}+e^i}$, where i is the depth of the agent's variable in the search tree and n is the number of agents.[2] Note that this function does not distinguish between the values within a domain of a variable held by some agent and, thus, if the decision is to elicit, the agent will elicit the constraint cost information for all its values, until the budget is exhausted.

[2] Sigmoid functions are used as activation functions in neural networks [9], but are not related to their use here.

The intuition that led to the design of this heuristic was that the deeper an agent is in the search tree, the larger is the chance that a solution improving on former solutions will be found. This is because every layer in the search tree can require additional elicitation.

Distance from Bound (DB). The decision whether to elicit the cost information for a value $x \in D_i$ is based on its distance Δ_x of the cost of the CPA from the upper bound (UB) maintained by SyncBB and a threshold $t(i)$ of the agent a_i. Specifically, the costs for x are elicited if $\Delta_x > t(i)$.

The distance from the upper bound Δ_x for each value $x \in D_i$ is calculated as follows: $\Delta_x = UB - (C(PA) + \delta_x)$, where $C(PA)$ is the cost of the current partial assignment and δ_x is the lowest cost that was generated from constraints with assignments of variables held by agents that come after A_i in the order, in previous attempts to assign x to X_i. If there were no previous attempts, $\delta_x = 0$.

The threshold $t(i)$ is calculated as follows: $t(i) = g \cdot \left(1 - \frac{e^i}{e^{n/2}+e^i}\right)$, where g is a constant that is dependent on the distribution of constraint costs in the problem.

The intuition is that when the distance from the upper bound is larger, there is more chance that this part of the search tree will include relevant solutions, since for a small distance an additional cost is expected to breach the bound and cause a backtrack.

Elicitation Required in Subtree (ERS). The decision whether to elicit the costs corresponding to a value assignment is done according to the number of unknown constraints in the subtree (of the search tree) rooted by this value. The intuition is that there is a larger chance that in a subtree with a small number of unknown constraints, complete solutions will be found with low elicitation cost.

In EAI-DCOPs, it is possible to count the number of unknown constraints in each such subtree. The process is performed bottom up, having the last agent in the order count the unknown constraints for each of the values in its domain, and sending this information up to the agent preceding it in the order. This agent adds the amount of constraints for each of its value and sends it up to the agent preceding it, and so forth. The process ends when the first agent in the order, holding the root variable of the subtree, updates the number of unknown constraints in each of the subtrees rooted by the values in its domain.

In more details, after an agent A_i receives a message that includes a number C_{i+1} of unknown constraints from the agent following it in the order, it can calculate, for each value $x \in D_i$, the number of constraints it is involved in among them. Thus, in order to calculate the number of unknown constraints in the subtree rooted by x, it reduces from C_{i+1} the number of constraints that all values $x' \in D_i, x' \neq x$ are involved in. To this number, it adds the number of unknown constraints that x is involved in with variables held by agents that are before A_i in the order. The total number of unknown constraints that A_i sends to A_{i-1} is then calculated as follows: $C_i = \sum_{x \in D_i} C_x$.

Each agent performs the initial calculations of C_x for every value x in the domain of its variables before the algorithm begins as a prepossessing procedure. After the algorithm starts, these parameters are updated following each elicitation as follows: When agent A_i elicits the unknown constraints for a value $x \in D_i$, it updates the number of unknown constraints for this value, and updates each of the agents involved in the constraints that were elicited. These agents reduce the corresponding number of unknown constraints for the subtrees of the corresponding values.

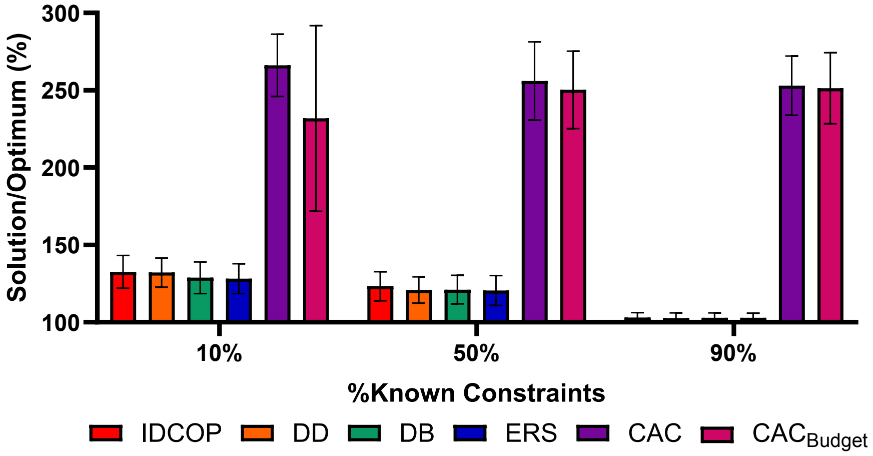


Figure 3 Solution costs when agents have a total budget of 105.

4.2 Solving EAI-DCOPs with DSA and MGM

Like in the case of SyncBB, the main challenge when solving EAI-DCOPs using distributed local search algorithms, such as the DSA and MGM, is in deciding which constraints to elicit. However, in contrast to SyncBB, DSA and MGM are incomplete and, thus, not all values with known constraints need to be assigned. Therefore, the decision whether to elicit needs to take into consideration the amount of uncertainty regarding the cost of a possible value assignment and the potential improvement it offers. In more details, we propose a heuristic that we incorporated in both DSA and MGM, according to which the elicitation decision for value $x \in D_i$ is performed if the following two conditions hold:

1. The number of unknown constraints that x is involved in is smaller than $q \cdot CN_x$, where CN_x is the total number of constraints that x is involved in and $0 \leq q \leq 1$.
2. The difference between the sum of the known constraints that x is involved in and the cost of the current partial assignment is larger than $g \cdot \frac{e^i}{e^{n/2}+e^i}$. Here, g is a constant as defined above for the formula in SyncBB and i is the iteration number.

5 Experimental Evaluation

In order to evaluate the success of our approach for generating high-quality I-DCOP solutions, under different elicitation "budget" restrictions, we performed experiments in which we compared the different heuristics that we proposed for solving EAI-DCOPs incorporated in DCOP algorithms, with the previous proposed approach for solving I-DCOP [12]. All our experiments were performed on a simulator, implemented in Python, on a Lenovo Carbon X1 Gen 9 computer with an 11th Generation Intel(R) Core (TM) i7-1165G7 @ 2.80GHz 2.80 GHz processor.

The complete algorithms solved I-DCOPs including seven agents, each holding one variable with four values in its domain. The average number of neighbors that agents had was 3. Constraint costs for combinations of assignments of neighboring agents were selected uniformly between 2 and 5. The percentage of unknown constraints varied. We generated problems in which the fraction of known constraints was 10%, 50%, and 90%. For each unknown cost, an elicitation cost was selected uniformly from the range $[0, 20]$.

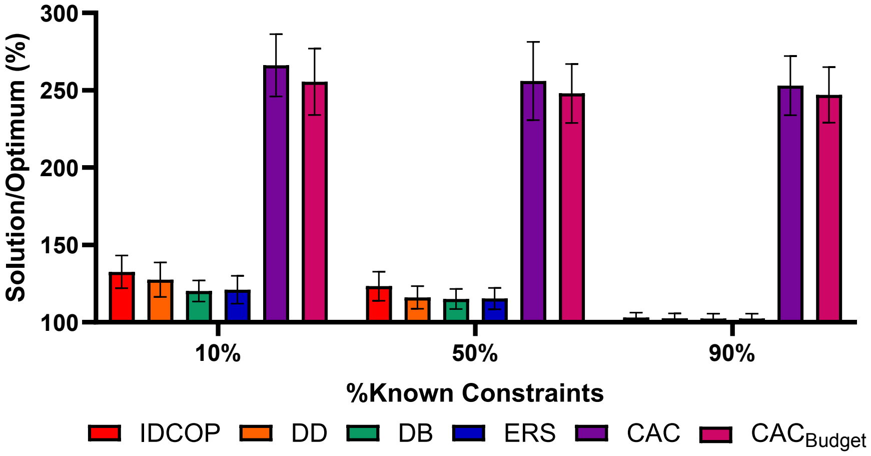


Figure 4 Solution costs when agents have a total budget of 525.

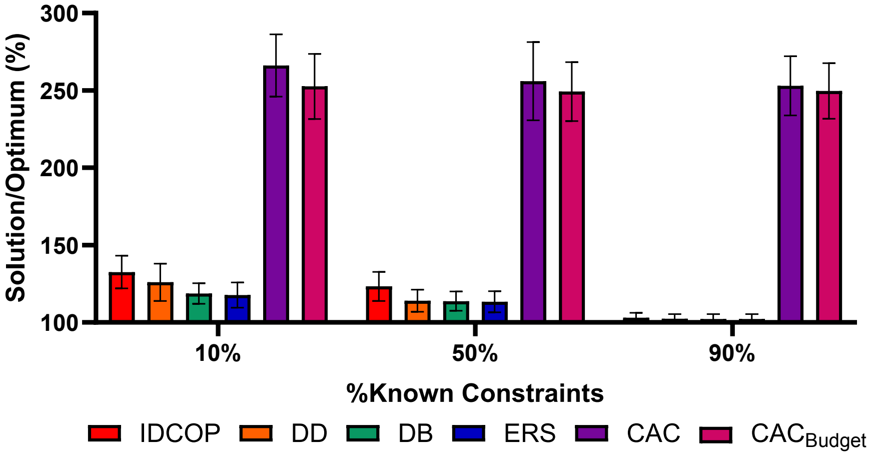


Figure 5 Solution costs when agents have a total budget of 1050.

5.1 SyncBB with Global Budgets

Previous I-DCOP approaches [12] with SyncBB assumed agents considered global elicitation costs, which are summations of elicitation costs over all agents. Therefore, to fairly compare against them, we also consider a variant, where agents have access to a global budget that can be accessed by all agents. Specifically, we set the global budget to either 105, 525, or 1050, so that the highest budget was an order of magnitude more than the lowest and they are easy to split among the agents in the personal budget version, which will be presented next. (We consider the variant where agents have personal budgets in the next section.)

Figure 3 depicts the average overhead in the solution quality with respect to the optimal solution (when all information is known), of solutions that the different algorithms produced when solving the problems with SyncBB, when agents were allocated the smallest global elicitation budget (105). The figure includes three batches of bars, one for each percentage of knowledge known to the agents in the beginning of the run of the algorithm (from left

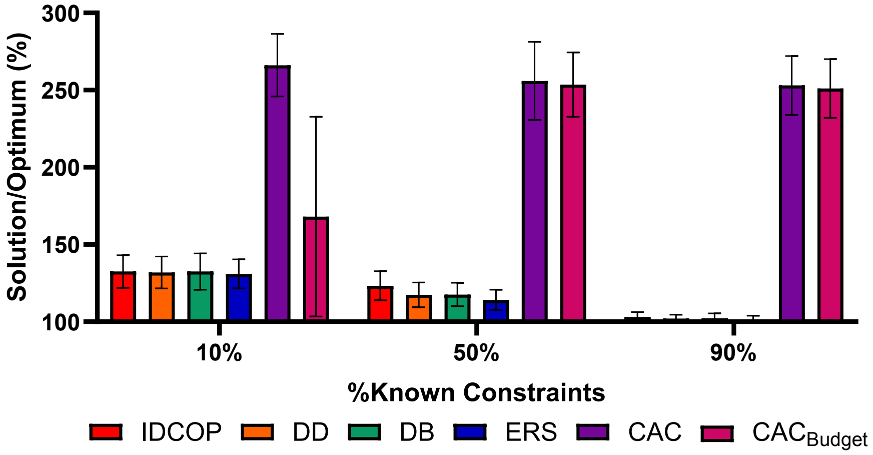


Figure 6 Solution costs when agents have a personal budget of 15.

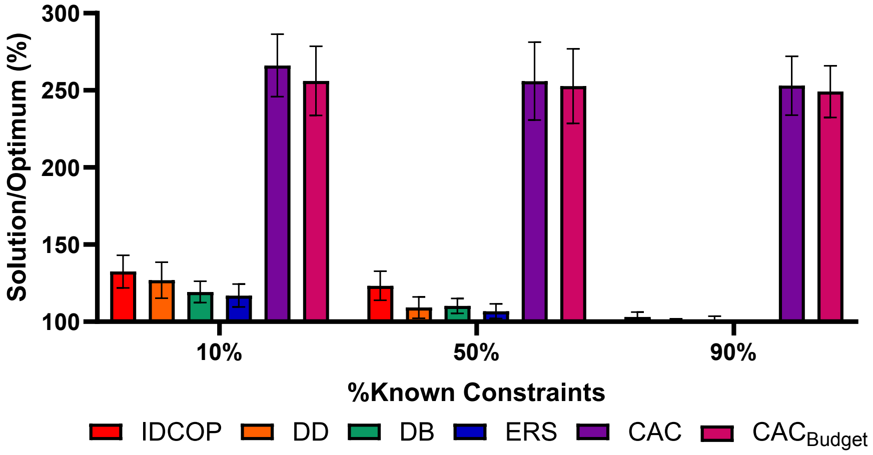


Figure 7 Solution costs when agents have a personal budget of 75.

to right: 10%, 50%, and 90%). The two bars on the right in each batch are the results of the I-DCOP algorithm using the *CAC* heuristic proposed previously [12]. In one version (labeled $\text{CAC}_{\text{budget}}$), the algorithm was limited by a budget, similar to the amount used by the EAI-DCOP heuristics (i.e., 105). In the second (labeled CAC), the algorithm was not bounded by a budget. Surprisingly, the version that was limited by a budget produced better results on average. We assume that a limited budget limits also the elicitation cost that is taken into consideration in the lower bound of the algorithm and, thus, the agents explore more solutions when they have a limited budget. It is apparent that both of these versions produce solutions with much greater costs than the solutions produced by the other versions. Among them, the bar on the left (labeled IDCOP) is a version of the algorithm that does not perform elicitation at all, while the three others present the average results of the algorithm using the limited budget according to the three heuristics described in Section 4.1.

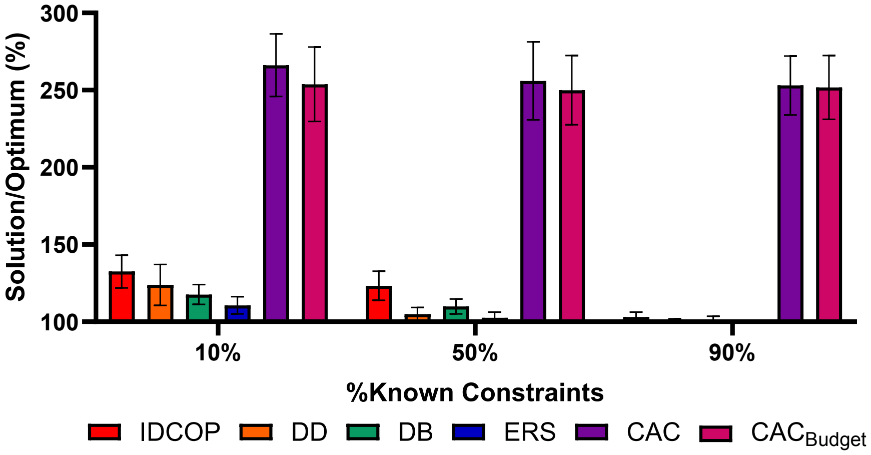


Figure 8 Solution costs when agents have a personal budget of 150.

The results clearly indicate that it is enough to solve the I-DCOP using SyncBB with no elicitation, in order to get a much better solution in comparison with the solutions produced by the algorithm implementing the ex-post approach and heuristic suggested by the literature [12]. Yet, performing elicitation using the allocated budget can reduce costs further. The best result is achieved by the ERS heuristic. However, its advantage over DD and DB is not significant. Figures 4 and 5 present similar results produced by scenarios in which agents had larger budgets (525 and 1050 respectively). While the trends seem similar, it is clear that the advantage of the proposed heuristics over the I-DCOP version without elicitation, is more apparent (as expected).

5.2 SyncBB with Personal Budgets

Figures 6, 7, and 8 present results for the same algorithms solving the same problems, only in this case the EAI-DCOP heuristics, that is, DD, DB, and ERS use personal budgets instead of a global budget as used in the experiments presented above. We divided the global costs such that there will be no difference in the total budget used by the agents. However, these scenarios present the more realistic case where an agent represents a user, and the budget limits the effort a user must spend in replying to queries during search. In these personal budget scenarios, the versions using the EAI-DCOP heuristics produced solution with a more significant advantage in general over the vanilla I-DCOP version. The difference was most apparent when the agents had medium or high budgets, and the initial knowledge available was 50%. The EAI-DCOP heuristics were able to produce solutions with a significant advantage over the vanilla I-DCOP version, and for the 150 budget per agent, the DD and ERS heuristics produce solutions that their quality was close to optimal. It seems that, besides being a more realistic scenario in a multi-agent environment, the budget per agent settings allows the algorithm to use elicitation in different parts of the search space and explore high quality solutions.

Figures 9, 10, and 11 present the runtimes in a logarithmic scale, of the algorithms in terms of NCLOs [16, 6]. In each figure, the budget allocated to the agents were different. The differences between the complete DCOP version and the I-DCOP version is identical in all figures because it is not affected by the budget. It is however affected by the amount

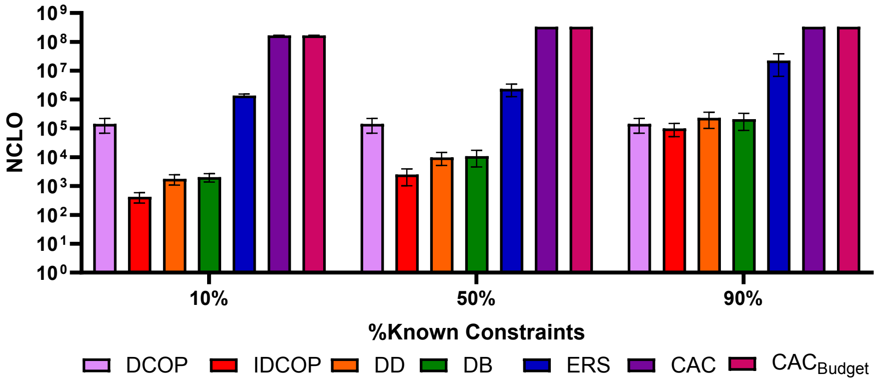


Figure 9 Runtime in terms of NCLOs with a budget of 105.

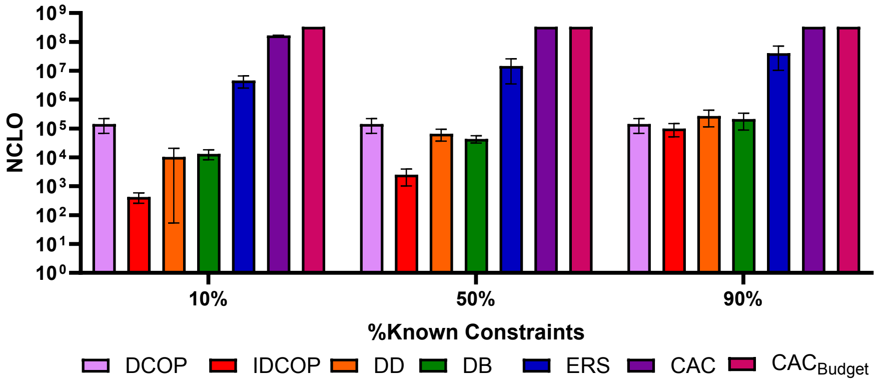


Figure 10 Runtime in terms of NCLOs with a budget of 525.

of knowledge known to agents: In the 10% scenario, the algorithm solving I-DCOP is much faster than the algorithm solving the complete DCOP while, in the 90% scenario, the runtimes are much closer. It is also apparent that the CAC and ERS versions of the algorithms are much slower than all other versions. The reason is that these heuristics require exponential computation before the algorithm begins its search (preparing the heuristic data in preprocessing). Moreover, unlike the SyncBB algorithms that performs some level of pruning, the preprocessing heuristics aggregate information from the entire search space and thus, their runtime is orders of magnitude larger. On the other hand, The DD and DB heuristics are much faster. Additionally, their advantage over the ERS heuristic in runtime is much more significant than the advantage that ERS provides in solution quality. Similar results were obtained when agents used personal budget as presented in Figures 12, 13 and 14.

5.3 MGM and DSA with Personal Budgets

In the second set of experiments we performed, we compared incomplete algorithms, that is, MGM and DSA, which were implemented in the ALS framework [17], in order to produce the anytime solutions as was done previously in the literature [12]. In this set of experiments, the problems included 50 agents, each holding a single variable with 10 values in its domain. Agents had 20 neighbors in average. The costs of constraints were randomly selected between 2 and 5. We present the solution costs of the algorithms in the first 50 iterations, because

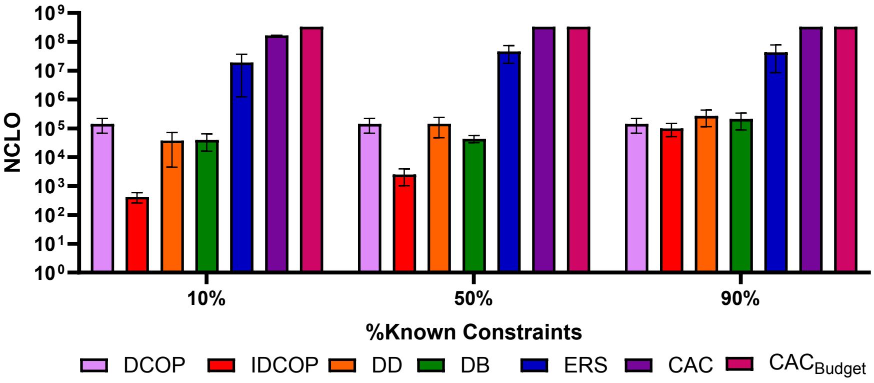


Figure 11 Runtime in terms of NCLOs with a budget of 1050.

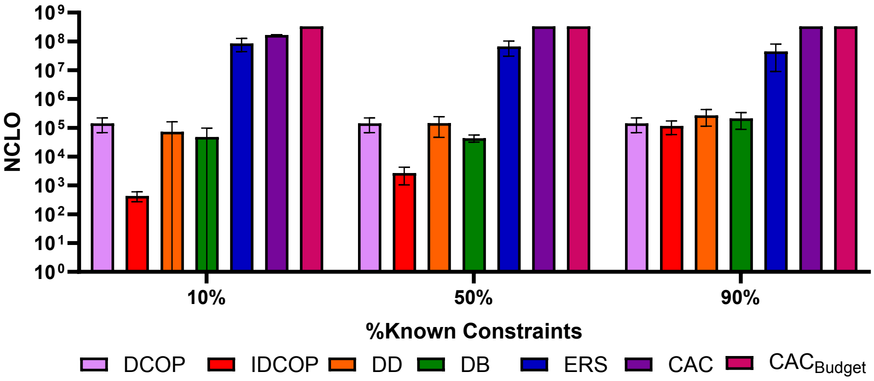


Figure 12 Runtime in terms of NCLOs when each agent had a personal budget of 15.

that was the number of iterations that were required for the algorithms to converge.

We first discuss the MGM results, since this was the algorithm that was implemented in the literature [12]. Figure 15 presents the solution cost as a function of the number of iterations performed by the algorithms. The DCOP version is the omniscient algorithm that knows all constraints. The dashed lines are the versions implementing the NHC heuristic [12]. The different lines represent the amount of initial knowledge. The other solid lines represent the results of the EAI-DCOP version in which the elicitation was performed according to the heuristic presented in Section 4.2. Each agent was allocated 180 elicitation queries. The significant advantage of the EAI-DCOP version is apparent regardless of the amount of initial knowledge the agents had.

Figure 16 presents the results of MGM and DSA performing the EAI-DCOP heuristic for deciding on elicitation, with an allocation 180 queries per agent (with 50% initial knowledge available). Surprisingly, the EAI-DCOP version of MGM is more successful than the DSA version. This is in contrast to the well-known advantage that DSA has over MGM when solving standard DCOPs.

Figure 17 presents the EAI-DCOP versions of MGM and DSA, when allocated fewer (120) queries per agent. Again, it is apparent that the MGM versions outperform the DSA versions, except for the versions solving problems with 10% initial knowledge, where both

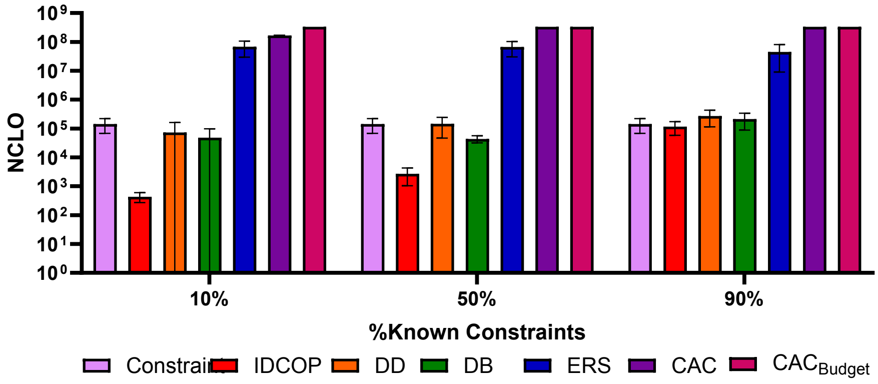


Figure 13 Runtime in terms of NCLOs when each agent had a personal budget of 70.

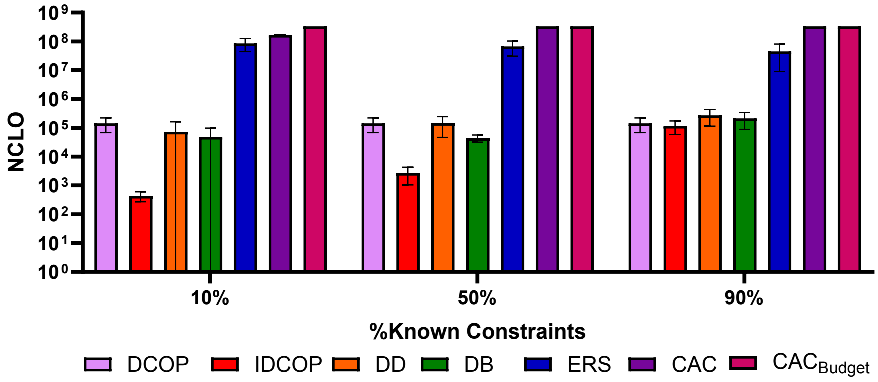


Figure 14 Runtime in terms of NCLOs when each agent had a personal budget of 150.

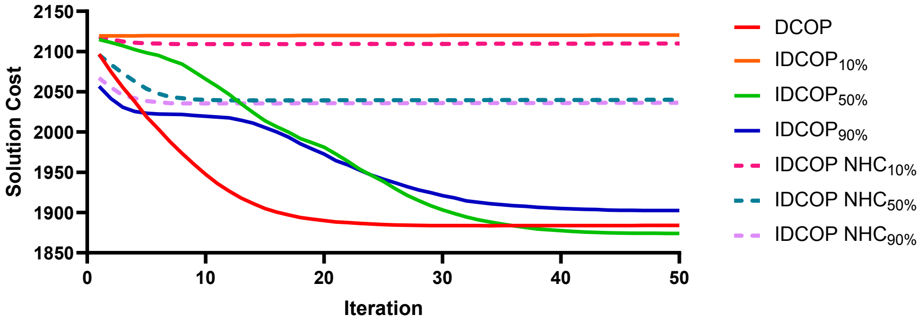


Figure 15 Solution cost as a function of the number of iterations.

algorithms struggle. Moreover, the versions that solved problems with 50% initial knowledge outperform the versions that solved problems with 90% knowledge. We assume that less knowledge resulted in a positive exploration effect, as was reported for environments with imperfect communication [8].

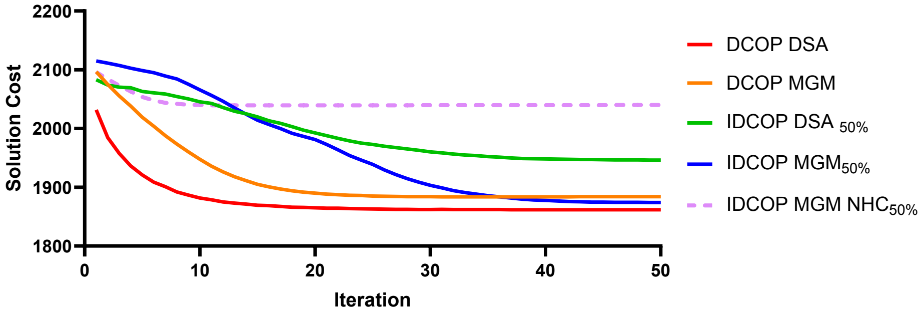


Figure 16 Solution cost as a function of the number of iterations.

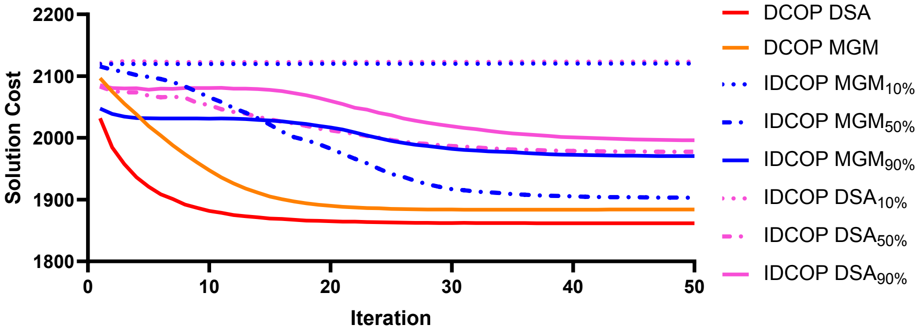


Figure 17 Solution cost as a function of the number of iterations.

6 Conclusions

We introduced a novel approach for solving I-DCOPs in this paper. In contrast to previous studies on I-DCOPs in which elicitation costs were considered *after* elicitations were made, we consider the costs *before* the elicitations in EAI-DCOPs.

The EAI-DCOP approach is not only more realistic, as it is not reasonable that agents will not use a high-quality solution after spending much effort to find it, it is also better in finding higher quality solutions *and* finding them with less runtime. These empirical results were shown on both complete and incomplete algorithms that are commonly used in the literature. Therefore, this seems to be one of the rare occasions where the new approach outperforms prior work on all three key relevant dimensions – practicality, solution quality, and runtime.

References

1 Ferdinando Fioretto, Enrico Pontelli, and William Yeoh. Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research*, 61:623–698, 2018.

2 Katsutoshi Hirayama and Makoto Yokoo. Distributed partial constraint satisfaction problem. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 222–236, 1997.

3 Tiep Le, Atena M. Tabakhi, Long Tran-Thanh, William Yeoh, and Tran Cao Son. Preference elicitation with interdependency and user bother cost. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1459–1467, 2018.

4 Rajiv Maheswaran, Jonathan Pearce, and Milind Tambe. Distributed algorithms for DCOP: A graphical game-based approach. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 432–439, 2004.

5 Pragnesh J. Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1–2):149–180, 2005.

6 Arnon Netzer, Alon Grubshtein, and Amnon Meisels. Concurrent forward bounding for distributed constraint optimization problems. *Artificial Intelligence*, 193:186–216, 2012.

7 Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1413–1420, 2005.

8 Ben Rachmut, Roie Zivan, and William Yeoh. Communication-aware local search for distributed constraint optimization. *Journal of Artificial Intelligence Research*, 75:637–675, 2022.

9 Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology*, 4(12):310–316, 2020.

10 Atena M. Tabakhi, Tiep Le, Ferdinando Fioretto, and William Yeoh. Preference elicitation for DCOPs. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 278–296, 2017.

11 Atena M. Tabakhi, Yuanming Xiao, William Yeoh, and Roie Zivan. Branch-and-bound heuristics for incomplete DCOPs. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1677–1679, 2021.

12 Atena M. Tabakhi, William Yeoh, and Roie Zivan. Incomplete distributed constraint optimization problems: Model, algorithms, and heuristics. In *Proceedings of the International Conference on Distributed Artificial Intelligence (DAI)*, pages 64–78, 2021.

13 William Yeoh, Ariel Felner, and Sven Koenig. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research*, 38:85–133, 2010.

14 William Yeoh, Pradeep Varakantham, Xiaoxun Sun, and Sven Koenig. Incremental DCOP search algorithms for solving dynamic DCOP problems. In *Proceedings of the International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, pages 257–264, 2015.

15 Weixiong Zhang, Guandong Wang, Zhao Xing, and Lars Wittenburg. Distributed stochastic search and distributed breakout: Properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence*, 161(1–2):55–87, 2005.

16 Roie Zivan and Amnon Meisels. Message delay and DisCSP search algorithms. *Annals of Mathematics and Artificial Intelligence*, 46:415–439, October 2006.

17 Roie Zivan, Steven Okamoto, and Hilla Peled. Explorative anytime local search for distributed constraint optimization. *Artificial Intelligence*, 212:1–26, 2014.

Minimizing Working-Group Conflicts in Conference Session Scheduling Through Maximum Satisfiability

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

Heythem Sattoutah
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Chu-Min Li
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Corinne Lucet
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Laure Brisoux-Devendeville
MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

Abstract

This paper explores the application of Maximum Satisfiability (Max-SAT) to the complex problem of conference session scheduling, with a particular focus on minimizing working-group conflicts within the context of the ROADEF conference, the largest French-speaking event aimed at bringing together researchers from various fields such as combinatorial optimization and operational research. A Max-SAT model is introduced then enhanced with new variables, and solved through state-of-the-art solvers. The results of applying our formulation to data from ROADEF demonstrate its ability to effectively compute session schedules, while enabling to reduce the number of conflicts and the maximum number of parallel sessions compared to the handmade solutions proposed by the organizing committees. These findings underscore the potential of Max-SAT as a valuable tool for optimizing conference scheduling processes, offering a systematic and efficient solution that ensures a smoother and more productive experience for attendees and organizers alike.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling; Computing methodologies → Logic programming and answer set programming

Keywords and phrases Maximum Satisfiability, Scheduling, Modeling

Digital Object Identifier 10.4230/LIPIcs.CP.2024.34

Category Short Paper

Supplementary Material *Other (Source Code, Data, Benchmark)*: `https://github.com/satoutahhaithem/ROADEF_SCHEDULING` [12]
archived at `swh:1:dir:7083377094f69163d37d30b77d740c72c562139d`

1 Introduction

The Maximum Satisfiability (Max-SAT) problem is a natural optimization extension of the well-known satisfiability (SAT) problem [9]. While SAT consists in verifying whether all the clausal constraints in a given propositional formula can be satisfied by an assignment of the variables, the goal in Max-SAT shifts to determining the maximum number of clausal constraints that can be satisfied. Although Max-SAT is NP-hard, the last decade has known major breakthroughs in Max-SAT theory and solving. Indeed, SAT-based algorithms [2, 18, 27] are able to harness the power of modern SAT solvers and particularly their ability to effectively compute cores, i.e. unsatisfiable subsets of the formula. More

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 34; pp. 34:1–34:11
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

recently, Branch and Bound algorithms for Max-SAT have also gained a huge leap in competitiveness with respect to other approaches through the introduction of a foundational learning process tailored to Max-SAT [11, 22, 23]. This was coupled with new insights and understanding of Max-SAT proof theory, paving the way for efficient solver certification for Max-SAT [7, 29].

These recent advancements have made Max-SAT a powerful formalism that can be used to model and effectively solve many real-world problems. In particular, Max-SAT has been used to tackle applications in various domains such as security [31], hardware and software debugging [25], data analysis [8] and bio-informatics [19] among many others. In particular, in the context of scheduling, Max-SAT has been used for tackling Curriculum-based Course Timetabling in [4] and [14], mainly for optimizing isolated lectures or resources in the schedule. It was also used in [10] to avoid idle time periods in the scheduling of bilateral meetings as well as in [15] to solve staff scheduling problems while taking into account requested and unpreferred shifts. A Max-SAT formulation for scheduling tasks in overloaded real-time systems by enforcing task deadlines as soft constraints is introduced in [24]. More recently, an iterative and incremental Max-SAT approach has been introduced to solve train scheduling optimization problems in [20]. These successful applications show the high potential of Max-SAT as an effective paradigm to solve problems in the context of scheduling and beyond.

In this paper, we address a conference scheduling problem through the lens of Max-SAT. In particular, we aim to tackle a problem which has arisen during the organization of the ROADEF conference, the largest French-speaking event aimed at bringing together researchers from various domains, mainly the combinatorial optimization and operational research communities. Our work addresses a crucial gap in the organization of the esteemed conference where, despite its status as a gathering of premier scheduling experts, the planning process remains reliant on manual efforts, requiring significant human and time resources, even after 25 editions. Our objective is thus to provide an effective way for optimizing the conference scheduling process, thus enhancing the overall experience for conference attendees and organizers alike. To this end, we introduce a Max-SAT model to minimize the working-group conflicts in the scheduling of parallel sessions during the conference. Furthermore, we show how our model can be enhanced by introducing new variables. Our experimental evaluation shows that our formulation offers a systematic and efficient solution to the intricate challenges associated with our problem, ensuring a smoother and more productive experience for attendees and organizers as it enables to reduce the number of conflicts as well as the maximum number of parallel sessions compared to the handmade solutions proposed by the organizing committees.

This paper is organized as follows. Section 2 includes the necessary definitions and notations. The problem context and parameters are presented in Section 3. Our dedicated Max-SAT model and its enhanced formulation are introduced in Section 4. An experimental evaluation on data from ROADEF is presented in Section 5. Finally, we conclude and discuss future work in Section 6.

2 Preliminaries

Let V be the set of propositional variables. A literal l is a variable $x \in V$ 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 $\alpha : V \rightarrow \{true, false\}$ maps each variable to a Boolean value and can be represented as a set of literals. A literal l is satisfied by an assignment α if $l \in \alpha$, else it is falsified by α. A clause is satisfied by an assignment α if at least one of its literals is satisfied by α, otherwise it is falsified by α. The cost of an assignment α, denoted $cost_\alpha(\phi)$, is the number of clauses in the formula ϕ falsified by α. Solving the (plain) Max-SAT problem [6] consists in determining the maximum number of clauses that can be satisfied by an assignment of a given CNF formula ϕ or, equivalently, the minimum number of clauses that each assignment must falsify, i.e. $opt(\phi) = \min_\alpha cost_\alpha(\phi)$. In our formulation, we will particularly rely on the partial Max-SAT problem [21], which is a well-known variant taking as input a partial CNF formula, i.e. a bipartite set of clauses $\phi = H \cup S$ where H is the set of hard clauses that must be satisfied and S is the set of soft clauses to be optimized as in plain Max-SAT. The goal thus shifts to computing $opt(S)$ such that all the clauses in H are satisfied. Finally, we note that, in the subsequent Max-SAT formulation, we will use Pseudo-Boolean (PB) constraints which enforce a constant bound on a weighted sum of literals as in $(\sum_{i=1}^{h} a_i * l_i) \circ k$ where $a_i, k \in \mathbb{N}$ and $\circ \in \{\leq, =, \geq\}$. In the case where all literals in the sum are variables in V and all the weights are set to 1 (and thus can be discarded), such constraints are more commonly referred to as cardinality constraints. PB and cardinality constraints can be efficiently encoded in CNF form [30] and are typically used in Max-SAT solvers to enforce relevant bounds during the search [21]. For instance, the sorting network encoding [16] enables to encode PB and cardinality constraints using $O(\sum_{i=1}^{h} a_i \, log^2(\sum_{i=1}^{h} a_i))$ new variables and clauses, i.e. $O(h \, log^2(h))$ for cardinality constraints as $m = h$. Independently, the cardinality network encoding [3] can be more suited for cardinality constraints as it is in $O(h \, log^2(k))$ while the BDD encoding [1], which is in $O(h^3 \, log(max_{1 \leq i \leq h} a_i))$, can be more suitable for PB constraints.

3 ROADEF Scheduling

The ROADEF[1] conference is the largest French-speaking event aimed at bringing together researchers from various domains, including combinatorial optimization, operational research, constraint programming and industrial engineering. This event is organized annually and welcomes around 600 participants, with the aim to foster exchanges and collaborations between researchers and industry professionals, as well as to contribute to the training of young researchers and encourage them to present their work to the national community. ROADEF includes plenary sessions, tutorials in semi-plenary sessions, multiple parallel sessions, industrial experience feedback sessions and the ROADEF general assembly. The conference also involves many working groups consisting of researchers collaborating on a national and potentially international level on specific themes covered by the conference, with each parallel session usually being organized by one or more of these working groups. In this paper, we focus on the scheduling of ROADEF parallel sessions into available time slots while avoiding clashes among research working groups. This problem arises each year during the event organization as each parallel session is usually scheduled to occur simultaneously with others, potentially causing overlaps and conflicts. Indeed, such overlaps not only prevent researchers from assisting to the sessions organized by their respective working groups but they may also pose constraints regarding session chair assignments.

Let S be the set of conference sessions, G be the set of working groups and C be the set of available slots[2]. A parallel session can be allocated to multiple slots with a different number of papers allocated to each slot. We denote L the set of authorized quantity of papers that can be allocated to a specific slot for each parallel session. We set n as the maximum number

[1] https://www.roadef.org/roadef-le-congres-annuel

[2] To simplify, each session, slot, or working group is referred to by a unique natural number identifier.

Table 1 Summary of ROADEF Data for the last four years. #X denotes the number of X.

Year	2024	2023	2022	2021
#Sessions	40	47	42	27
#Working-groups	20	24	24	17
#Slots	7	7	8	11
Paper range *L*	{3,4,5,6}		{3,4,5}	
#Papers	307	358	311	182

of parallel sessions for the available slots, which is fixed each year depending on the available resources. Note that, in general, a higher value for this parameter entails more complex logistics and more resources to manage the parallel sessions. For each session $s \in S$, we denote $np(s)$ and $W(s) \subseteq G$ respectively the number of papers and the set of working groups associated to session s. Finally, for each slot $c \in C$, we denote $npMax(c)$ the maximum number of authorized papers for each parallel session in slot c.

For instance, in 2024, the 25^{th} edition of ROADEF[3] was organized over three days in the city of Amiens. The conference comprised 40 programmed sessions[4]. It also involved the participation of 20 distinct working groups. Over the course of the three-day conference, there were seven parallel session slots. The organizing committee has decided that each parallel session was required to host at least three papers and at most six. Indeed, since the allocated presentation and question time per paper is 20 minutes, a session with less than 3 papers, i.e. less than one hour, would be too short while, one with more than 6 would be too long since a break would be expected after 2 hours. The maximum number of parallel sessions per slot can go up to 15 with respect to the available resources and mainly the available rooms in the conference site. For the lack of space, we do not report the maximum number of authorized papers for each slot nor the values of $np(s)$ and $W(s)$ for $s \in S$. However, we mention that there were a total of 307 papers accepted in the 2024 edition, split over the different sessions. To give an example, for session $s = 40$ dedicated to constraint programming, we have $np(s) = 8$ and $W(s) = \{7, 15\}$. The summary of Data from ROADEF over the last four years is reported in Table 1. Finally, we note that additional constraints for specific sessions may arise. In particular, the chairs of session 34 in the 2024 edition requested to assign their session presentations on the last day of the conference covered by slots 5, 6 and 7.

4 Maximum Satisfiability for Conference Session Scheduling

4.1 Basic Model

Our aim is to minimize the conflicts among working groups in the conference schedule while respecting all the schedule-related constraints. We start first by defining our main variables. More specifically, we set two types of variables to represent the allocation of parallel sessions and to compute the conflicts as follows:

- $x_{(s,c,l)}$ is set to true if session $s \in S$ is allocated to slot $c \in C$ with $l \in L$ papers
- $y_{(s_1,s_2,c,g)}$ is set to true if there is a conflict in slot $c \in C$ for working group $g \in G$ associated to the pair of sessions $(s_1, s_2) \in S^2$ such that $s_1 < s_2$ and $g \in W(s_1) \cap W(s_2)$

[3] https://roadef2024.sciencesconf.org/

[4] 50 parallel sessions were initially programmed in 2024 but 6 were canceled for lack of submissions and 4 were merged with other sessions for insufficient number of submissions.

Since we want to reduce the number of group conflicts in the conference, we clearly need to minimize the number of satisfied y variables. We can therefore define the set of soft clauses accordingly as follows:

$$\phi_{soft} = \bigwedge_{\substack{(s_1,s_2,c,g)\in S\times S\times C\times G\\ s1<s2\\ g\in W(s_1)\cap W(s_2)}} \overline{y_{(s_1,s_2,c,g)}}$$

Next, we state the constraints that will be added to the hard clause set ϕ_{hard} of the formula:

- At most one quantity of papers can be chosen for each (session , slot) pair:

$$\sum_{l\in L} x_{(s,c,l)} \leq 1 \qquad \forall (s,c)\in S\times C \tag{1}$$

- The subdivision of a session into slots covers all the papers in the session:

$$\sum_{\substack{c\in C\\ l\in L}} x_{(s,c,l)} * l = np(s) \qquad \forall s\in S \tag{2}$$

- The allocation of a session to a slot c is valid if $npMax(c)$ is not exceeded:

$$\bigwedge_{\substack{l\in L\\ l>npMax(c)}} \overline{x(s,c,l)} \qquad \forall (s,c)\in S\times C \tag{3}$$

- The maximum number of parallel sessions per slot is not exceeded:

$$\sum_{\substack{s\in S\\ l\in L}} x_{(s,c,l)} \leq n \qquad \forall c\in C \tag{4}$$

- Two sessions associated to the same group and allocated to the same slot generate a conflict:

$$\bigwedge_{(l_1,l_2)\in L^2} \overline{x_{(s_1,c,l_1)}} \vee \overline{x_{(s2,c,l_2)}} \vee y_{(s_1,s_2,c,g)} \tag{5}$$

$$\forall (s_1,s_2,c,g)\in S^2\times C\times G \text{ s.t } s_1<s_2 \text{ and } g\in WG(s_1)\cap WG(s_2)$$

- Session-specific constraints such as the one which arose for session 34 in the 2024 edition can be written as follows:

$$\bigwedge_{\substack{c\in C\setminus\{5,6,7\}\\ l\in L}} \overline{x_{(34,c,l)}} \tag{6}$$

4.2 Enhanced Model

In order to enhance our model, we first remark that Constraint (5) can be written differently as stated in the following proposition which can be easily proven by the distributive properties of logical conjunction and disjunction.

▶ **Proposition.** *Constraint (5) is logically equivalent to the following constraint:*

$$(\bigwedge_{l\in L} \overline{x_{(s_1,c,l)}}) \vee (\bigwedge_{l\in L} \overline{x_{(s_2,c,l)}}) \vee y_{(s_1,s_2,c,g)}$$

$$\forall (s_1,s_2,c,g)\in S^2\times C\times G \; s.t \; s_1<s_2 \; and \; g\in WG(s_1)\cap WG(s_2)$$

As such, we can clearly observe that the information represented by the conjunctions in the clauses may occur redundantly in the constraint. Note that, for a given (session,slot) pair, these conjunctions represent the absence of a session from a given slot. Using these observations, we introduce new variables to our model as follows:

- $z_{(s,c)}$ is set to *true* if session $s \in S$ is not allocated to slot $c \in C$

To enforce the semantic meaning of these variables, we add the following constraint which can be easily rewritten in CNF form using the CNF rewriting of logical equivalence and the distributivity laws:

- When a session is not allocated to a given slot, the corresponding z variable is set to *true*, otherwise to $false$:

$$z_{(s,c)} \Leftrightarrow \bigwedge_{l \in L} \overline{x_{(s,c,l)}} \qquad \forall (s,c) \in S \times C \tag{7}$$

Now, we can rewrite some of the previous constraints as shown below.

- We can optimize the left-hand side of constraint (4) through the z variables as follows:

$$\sum_{s \in S} \overline{z_{(s,c)}} \leq n \qquad \forall c \in C \tag{4*}$$

- We can also simply remodel constraint (5) based on the rewriting established in the proposition above and by relying on the newly introduced z variables:

$$z_{(s_1,c)} \vee z_{(s_2,c)} \vee y_{(s_1,s_2,c,g)} \quad \forall (s_1,s_2,c,g) \in S^2 \times C \times G, s_1 < s_2 \text{ and } g \in W(s_1) \cap W(s_2) \tag{5*}$$

- Finally, the z variables can be also used to effectively encode session-specific constraints as follows:

$$\bigwedge_{c \in C \setminus \{5,6,7\}} z_{(34,c)} \tag{6*}$$

5 Experimental Evaluation

We have encoded our Max-SAT models[5] using the PySAT[6] library [17]. To compare our enhanced formulation to the basic model, we used the Sorting Network (SortN) encoding which is available for both PB and cardinality constraints in PySAT. Furthermore, to show that our enhanced model can be further improved, we also test a CardN+PBbest combination, with a cardinality encoding for cardinality constraints and the “best” option set for PB constraints in PySAT, which typically invokes the BDD encoding. Using data extracted from the last four years of ROADEF, we have generated a total of 72 instances, by choosing 7 varying values of n for each year. This enables us to not only minimize the number of working-group conflicts for a given n value but to also seek an optimal value for the maximum number of parallel sessions. The features of the generated instances in terms of number of variables and clauses are reported in Table 2. We have solved the generated instances using different state-of-the-art Max-SAT solvers including RC2 [18], OpenWBO (WBO) [26], MaxCDCL [23] and MaxHS [13]; all first-ranked solvers in previous Max-SAT evaluations.

[5] Our code, data and benchmark are available in the following **GitHub repository**

[6] `https://pysathq.github.io/`

Table 2 Summary of instance features and computed optimal values for the whole benchmark. # X denotes number of X. An interval is indicated when there are several values possibles. For the sake of comparison, we report the values obtained through the handmade scheduling by the organizing committees between "()" in the last two rows.

Year	2024		2023		2022		2021	
Encoding	SortN	CardN+PBbest	SortN	CardN+PBbest	SortN	CardN+PBbest	SortN	CardN+PBbest
# Var	45112	11007	49917	[12309-15235]	44212	[15810-12466]	44612	[9337-10173]
# Hard	67754	20208	74926	[22640-27029]	65950	[26122-21106]	66807	[16209-17463]
# Soft	308		518		880		264	
# Conflicts	4 (6)		9 (35)		29 (38)		0 (3)	
Optimum n	10 (11)		13 (14)		11 (11)		5 (5)	

For the sake of comparison, we have also implemented our models using OR-Tools to perform tests with the constraint programming solver CP-SAT [28]. The tests were performed on a machine equipped with an Intel Core i7 processor clocked at 3.80GHz, under Ubuntu 22.04. A timeout of 3600s was set for each instance. The results in terms of solving time are described in Table 3.

First, it is crucial to highlight the substantial enhancements our solutions have achieved over the manually crafted schedules by the organizing committees in recent years. More specifically, our models have remarkably enabled to improve both the number of conflicts and the management of parallel sessions. Indeed, for the years 2024 to 2021, our automated solutions reduced the number of working-group conflicts respectively by 2, 26, 9, and 3. This reduction emphasizes the relevance of our computational approach over traditional manual methods. Furthermore, our formulation has successfully decreased the maximum number of parallel sessions required in the 2024 and 2023 schedules, by reducing it from 11 to 10 in 2024 and from 14 to 13 in 2023. These results enable to significantly lessen the operational complexity and the resources required for organizing the conference and to ensure a smoother and more productive experience for attendees and organizers alike.

Next, we compare our enhanced formulation to the basic one. As showcased in Table 3, the comparison between the two models with Max-SAT solvers for a similar encoding (SortN) clearly supports the relevance of introducing z variables into the model. Across all years, the enhanced model consistently outperforms the basic one in terms of solving times. Notably, in terms of total solving time across all MaxSAT solvers, it achieves a gain of 62.9%, 67.2% and 44.7% respectively for years 2024, 2023 and 2021, with a total gain of 66.3% over the three years. In particular, the solving times were significantly improved for specific solvers as, for instance, more than 93% gain in solving time is achieved for the solver RC2 in 2023 with our enhanced formulation. Furthermore, the enhanced model's ability to produce better feasible solutions is particularly pronounced. Indeed, for the instances of 2023, the model not only enables the MaxCDCL solver to solve some instances within the timeout but also to achieve a feasible solution (which is optimal) for the remaining ones, unlike the basic model where it fails on all the instances. For the instances of 2022, although the optimal solutions are not found within the timeout[7], it is crucial to emphasize that the enhanced model consistently improves the best feasible solutions compared to the basic model. In fact, the optimal values for the number of conflicts is in [27,29] with $n = 11$ showing that the enhanced model is able to achieve a tight optimal solution interval for 2022, with a feasible solution of 29 achieved by

[7] Note that we did not change the internal behavior of the solvers and, in particular, MaxHS which may stop and return the best feasible solution before the timeout if it finds the problem hard to solve and does not manage to achieve optimality

Table 3 Solving times in seconds for the Basic Model with SortN and the Enhanced Model with SortN and CardN+PBbest. For each one, the best solving time for each instance is highlighted in bold. The best overall results for each instance are also underlined. T denotes the occurrence of a timeout. If the solver is not able to compute the optimal solution, the best obtained feasible solution is reported between "()" and, if a lower bound is provided, it is indicated between "[]".

		Basic Model					Enhanced Model					Enhanced Model with CardN+PBbest			
Year	**n**	**RC2**	**WBO**	**MaxCDCL**	**MaxHS**	**CP-SAT**	**RC2**	**WBO**	**MaxCDCL**	**MaxHS**	**CP-SAT**	**RC2**	**WBO**	**MaxCDCL**	**MaxHS**
2024	15	**3.428**	7.811	5.358	16.086	26.800	1.259	1.180	**0.989**	17.933	7.683	0.767	**0.300**	0.467	9.925
	14	**3.054**	12.095	5.716	46.308	26.291	1.300	1.130	**0.818**	14.633	6.677	0.716	0.301	**0.290**	11.236
	13	**3.288**	6.989	5.591	46.206	20.622	1.199	1.404	**0.992**	18.585	7.713	0.770	**0.279**	0.353	12.476
	12	**3.540**	6.036	5.147	95.241	22.691	1.315	1.742	**0.805**	20.607	10.056	0.718	**0.309**	0.313	5.972
	11	**3.652**	12.384	8.746	153.335	22.594	1.287	2.472	**1.100**	71.655	10.390	0.707	0.392	0.395	26.609
	10	**9.902**	1398.111	511.043	869.119	88.831	**9.707**	341.225	150.416	271.440	35.460	1.544	411.766	213.257	500.964
2023	17	197.515	**119.674**	T (-)	1187.325	T (-)	**9.687**	17.769	T (9)	357.906	T (9)	10.370	**6.136**	3371.304	353.515
	16	355.627	**71.472**	T (-)	755.975	T (9)	**12.602**	20.701	3058.418	395.716	T (9)	12.145	**4.792**	2866.853	426.965
	15	139.625	**99.332**	T (-)	1106.281	T (9)	**23.467**	41.042	T (9)	408.854	T (9)	15.247	**6.696**	2686.484	405.923
	14	107.202	**102.947**	T (-)	1036.948	T (9)	**14.011**	22.683	T (9)	413.160	T (9)	19.751	**6.706**	T (9)	417.693
	13	250.308	**199.549**	T (-)	853.876	T (9)	**8.006**	14.962	3136.008	439.812	T (9)	8.733	**4.162**	T (9)	438.877
	12	T [10]	T (-)	T (-)	T (28)[8]	T (-)	T (-)[10]	T (-)	T (-)	T (23)[8]	T (10)	T [10]	T (-)	T (-)	T (30)[9]
2022	16	T (-)[24]	T (127)	**T (29)**	385.625 (46)[18]	**T (29)**	T (-)[25]	T (146)	**T (29)**	496.061 (42)[22]	**T (29)**	T (-)[22]	T (83)	**T (29)**	1024.785 (40)[25]
	15	T (-)[24]	T (161)	**T (29)**	380.048 (37)[26]	**T (29)**	T (-)[24]	T (134)	**T (29)**	322.627 (51)[18]	**T (29)**	T (-)[24]	T (58)	**T (29)**	1059.200 (33)[25]
	14	T (-)[25]	T (196)	**T (29)**	T (37)[26]	**T (29)**	T (-)[27]	T (132)	**T (29)**	473.504 (38)[23]	**T (29)**	T (-)[24]	T (47)	**T (29)**	**448.613 (29)[23]**
	13	T (-)[24]	T (222)	**T (29)**	392.299 (53)[18]	**T (29)**	T (-)[27]	T (146)	**T (29)**	348.256 (56)[17]	**T (29)**	T (-)[25]	T (50)	**T (29)**	330.284 (59)[18]
	12	T (-)[24]	T (324)	**T (29)**	856.923 (50)[24]	**T (29)**	T (-)[26]	T (135)	**T (29)**	980.692 (30)[25]	**T (29)**	T (-)[24]	T (71)	**T (29)**	328.658 (59)[18]
	11	T (-)[25]	T (543)	**T (29)**	T (40)[27]	**T (29)**	T (-)[27]	T (436)	**T (29)**	713.466 (56)[18]	**T (29)**	T (-)[24]	T (287)	**T (29)**	650.517 (48)[18]
2021	10	0.670	1.397	**0.503**	0.653	1.356	0.656	0.748	0.179	**0.132**	1.307	0.178	0.139	0.050	**0.027**
	9	1.210	1.380	**0.276**	0.685	1.324	0.309	0.812	0.170	**0.155**	1.316	0.133	0.136	0.038	**0.026**
	8	1.232	1.352	**0.437**	0.682	1.327	0.654	0.775	**0.278**	0.454	1.272	0.132	0.148	0.039	**0.030**
	7	1.189	1.364	**0.262**	0.619	1.333	0.649	0.791	**0.256**	0.448	1.276	0.126	0.132	0.035	**0.030**
	6	0.646	1.384	**0.414**	0.695	1.307	0.350	0.776	**0.281**	0.470	1.274	0.135	0.146	0.035	**0.031**
	5	1.142	1.724	**0.425**	0.696	1.347	0.661	0.885	**0.279**	0.459	1.280	0.142	0.200	0.035	**0.027**

MaxCDCL. Overall, the results provide strong empirical evidence supporting the importance of introducing z variables in the scheduling model. This observation remains true for the CP-SAT solver which also shows superior performance with the enhanced formulation.

Note how CP-SAT achieves a timeout on all the instances of 2022 and 2023 both with the basic and enhanced models and does not manage to outperform the best MaxSAT solver in 2021 and 2024 for all the instances, which highlights the relevance of our MaxSAT approach in managing large-scale scheduling tasks. In addition, for the sake of enhancing MaxSAT solver performance, we have tested the CardN+PBbest combination with the enhanced model and we have obtained promising outcomes. Specifically, this combination demonstrates notable improvements in the best solving times compared to SortN, with a gain of 67.5%, 59.9% and 88.2% respectively for 2024, 2023 and 2021, thus averaging to more than 70% gain over the three years. Finally, note how the instances become particularly difficult for the smallest n within each year. In fact, the choices of the n values were not arbitrary as a formal lower bound can be established by calculating an upper bound of the number of papers for a given n as follows: $UBP_n = \sum_{c \in C} n * npMax(c)$. For instance, in 2024, we have $UBP_9 = 288 < \sum_{s \in S} np(s) = 307$ and, therefore, the the set of hard constraints becomes unsatisfiable for $n = 9$ as it is impossible to schedule all the sessions so as to achieve the total amount of papers. In fact, we have also generated the instances for $n = 9$ in 2024, $n = 11$ in 2023, $n = 10$ in 2022 and $n = 4$ in 2021 totaling to 12 unsatisfiable instances. Unsurprisingly, while CP-SAT is able to prove the infeasibility of these instances, all the MaxSAT solvers failed within the timeout as these solvers are scarcely confronted to instances where the set of hard clauses is unsatisfiable [5]. This underscores the importance of enlarging the benchmarks in Max-SAT evaluations with such instances and, specifically, ones extracted from real-life scenarios such as ours to keep up the driving force in improving modern solvers.

6 Conclusion

In this paper, we addressed an optimization problem in conference session scheduling. More specifically, we presented a Max-SAT formulation for the minimization of working-group conflicts in the ROADEF conference, which we then enhanced by introducing new variables. Our results indicate that the number of conflicts and the maximum number of parallel sessions could be reduced and a scheduling could be computed in an effective time frame. Our work thus holds great promise in simplifying the scheduling task, streamlining logistics and optimizing resources in subsequent editions of the conference. As future work, we plan to expand our research by conducting experiments on additional datasets, both within and beyond the scope of ROADEF. Additionally, we aim to enhance our scheduling approach by incorporating considerations for author conflicts so as to develop a more comprehensive optimization framework that accounts for the intricate relationships among participants.

References

1 Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. BDDs for Pseudo-Boolean Constraints - Revisited. In Karem A. Sakallah and Laurent Simon, editors, *14th International Conference on Theory and Applications of Satisfiability Testing - SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2011. doi:10.1007/978-3-642-21581-0_7.

2 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. SAT-based MaxSAT algorithms. *Artif. Intell.*, 196:77–105, 2013. doi:10.1016/J.ARTINT.2013.01.002.

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 Roberto Javier Asín Achá and Robert Nieuwenhuis. Curriculum-based course timetabling with SAT and MaxSAT. *Ann. Oper. Res.*, 218(1):71–91, 2014. doi:10.1007/S10479-012-1081-X.

5 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. MaxSAT Evaluation 2018: New Developments and Detailed Results. *J. Satisf. Boolean Model. Comput.*, 11(1):99–131, 2019. doi:10.3233/SAT190119.

6 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 929–991. IOS Press, 2021. doi:10.3233/FAIA201008.

7 Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified Core-Guided MaxSAT Solving. In Brigitte Pientka and Cesare Tinelli, editors, *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2023. doi:10.1007/978-3-031-38499-8_1.

8 Jeremias Berg, Antti Hyttinen, and Matti Järvisalo. Applications of MaxSAT in Data Analysis. In Daniel Le Berre and Matti Järvisalo, editors, *Proceedings of Pragmatics of SAT 2015, Austin, Texas, USA, September 23, 2015 / Pragmatics of SAT 2018, Oxford, UK, July 7, 2018*, volume 59 of *EPiC Series in Computing*, pages 50–64. EasyChair, 2018. doi:10.29007/3QKH.

9 A. Biere, M. Heule, and H. van Maaren. *Handbook of Satisfiability: Second Edition*. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021. URL: https://books.google.fr/books?id=dUAvEAAAQBAJ.

10 Miquel Bofill, Marc Garcia, Josep Suy, and Mateu Villaret. MaxSAT-Based Scheduling of B2B Meetings. In Laurent Michel, editor, *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*, volume 9075 of *Lecture Notes in Computer Science*, pages 65–73. Springer, 2015. doi:10.1007/978-3-319-18008-3_5.

11 Mohamed Sami Cherif, Djamal Habet, and André Abramé. Understanding the power of max-sat resolution through up-resilience. *Artif. Intell.*, 289:103397, 2020. doi:10.1016/J.ARTINT.2020.103397.

12 Sami Cherif, Heythem Sattoutah, Chu-Min Li, Corinne Lucet, and Laure Brisoux-Devendeville. ROADEF_SCHEDULING. Software, swhId: swh:1:dir:7083377094f69163d37d30b77d740c72c562139d (visited on 2024-08-16). URL: https://github.com/satoutahhaithem/ROADEF_SCHEDULING.

13 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.

14 Emir Demirovic and Nysret Musliu. MaxSAT-based large neighborhood search for high school timetabling. *Comput. Oper. Res.*, 78:172–180, 2017. doi:10.1016/J.COR.2016.08.004.

15 Emir Demirovic, Nysret Musliu, and Felix Winter. Modeling and solving staff scheduling with partial weighted maxSAT. *Ann. Oper. Res.*, 275(1):79–99, 2019. doi:10.1007/S10479-017-2693-Y.

16 Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. *J. Satisf. Boolean Model. Comput.*, 2(1-4):1–26, 2006. doi:10.3233/SAT190014.

17 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, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2018. doi:10.1007/978-3-319-94144-8_26.

18 Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an Efficient MaxSAT Solver. *J. Satisf. Boolean Model. Comput.*, 11(1):53–64, 2019. doi:10.3233/SAT190116.

19 Gerold Jäger, Sharlee Climer, and Weixiong Zhang. The complete parsimony haplotype inference problem and algorithms based on integer programming, branch-and-bound and Boolean satisfiability. *J. Discrete Algorithms*, 37:68–83, 2016. doi:10.1016/J.JDA.2016.06.001.

20 Alexandre Lemos, Filipe Gouveia, Pedro T. Monteiro, and Inês Lynce. Iterative Train Scheduling under Disruption with Maximum Satisfiability. *J. Artif. Intell. Res.*, 79:1047–1090, 2024. doi:10.1613/JAIR.1.14924.

21 Chu Min Li and Felip Manyà. MaxSAT, Hard and Soft Constraints. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 903–927. IOS Press, 2021. doi:10.3233/FAIA201007.

22 Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Combining clause learning and branch and bound for maxsat. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 38:1–38:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CP.2021.38.

23 Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Boosting branch-and-bound MaxSAT solvers with clause learning. *AI Commun.*, 35(2):131–151, 2022. doi:10.3233/AIC-210178.

24 Xiaojuan Liao, Hui Zhang, Miyuki Koshimura, Rong Huang, and Wenxin Yu. Maximum Satisfiability Formulation for Optimal Scheduling in Overloaded Real-Time Systems. In Abhaya C. Nayak and Alok Sharma, editors, *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*, volume 11670 of *Lecture Notes in Computer Science*, pages 618–631. Springer, 2019. doi:10.1007/978-3-030-29908-8_49.

25 Hratch Mangassarian, Andreas G. Veneris, and Farid N. Najm. Maximum circuit activity estimation using pseudo-boolean satisfiability. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 31(2):271–284, 2012. doi:10.1109/TCAD.2011.2169259.

26 Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-WBO: A Modular MaxSAT Solver. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, 2014. doi:10.1007/978-3-319-09284-3_33.

27 António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints An Int. J.*, 18(4):478–534, 2013. doi:10.1007/S10601-013-9146-2.

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

29 Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Proofs and Certificates for Max-SAT. *J. Artif. Intell. Res.*, 75:1373–1400, 2022. doi:10.1613/JAIR.1.13811.

30 Olivier Roussel and Vasco M. 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.

31 Ahmad Shabani and Bijan Alizadeh. PMTP: A MAX-SAT-Based Approach to Detect Hardware Trojan Using Propagation of Maximum Transition Probability. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(1):25–33, 2020. doi:10.1109/TCAD.2018.2889663.

On the Complexity of Integer Programming with Fixed-Coefficient Scaling

Jorke M. de Vlas
Linköping Universitet, Sweden

Abstract

We give a polynomial time algorithm that solves a CSP over $\mathbf{Z}$ with linear inequalities of the form $c^{a_1}x - c^{a_2}y \leq b$ where x and y are variables, a_1, a_2 and b are parameters, and c is a fixed constant. This is a step in classifying the complexity of CSP(Γ) for first-order reducts Γ from $(\mathbf{Z}, <, +, 1)$. The algorithm works by first reducing the infinite domain to a finite domain by inferring an upper bound on the size of the smallest solution, then repeatedly merging consecutive constraints into new constraints, and finally solving the problem using arc consistency.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Complexity theory and logic

Keywords and phrases constraint satisfaction problems, integer programming, CSP dichotomy

Digital Object Identifier 10.4230/LIPIcs.CP.2024.35

Category Short Paper

Funding This work is partially supported by the Swedish Research Council (VR) under grant VR-2021-04371.

Acknowledgements I wish to thank P. Jonsson and G. Osipov for discussions and feedback.

1 Introduction

Many computational problems in theoretical computer science can be formulated as a Constraint Satisfaction Problem or CSP. In these problems the goal is to find an assignment of values to a set of variables from a given domain that satisfies some given constraints that impose relations on subsets of variables. After fixing a domain and the allowed constraint types, we are left with a computational problem and want to determine its computational complexity. In general, we are interested in determining whether a given CSP is solvable in polynomial time or NP-hard.

For Boolean domains, Schaefer's dichotomy theorem gives a full classification [9]. This result has recently been extended to arbitrary finite domains [3, 12]. For infinite domains, there is no known dichotomy and this is an area of active research [1]. We focus on TVPI, the infinite-domain CSP which consists of linear inequalities with two variables per inequality. In general, this CSP is NP-complete [6, Theorem F], but it admits many subclasses whose tractability is unknown.

Inequalities in TVPI consist of the form $ax + by \leq d$ for variables x, y and integer coefficients a, b, d. Several tractable subclasses of TVPI focus on restricting the linear coefficients a and b. Restricting to $\{\pm 1\}$, we obtain the tractable UTVPI [5]. Restricting to $\{\pm 1, \pm 2\}$, we obtain the class BTVPI, which was recently shown to also be tractable [10].

An intractable subclass of TVPI is the CSP that concerns monotone inequalities, where the coefficients a and b have different signs [6]. This class admits a pseudo-polynomial time algorithm using a rounded version of a technique known as Fourier-Motzkin elimination, which iteratively combines pairs of inequalities into new inequalities [4]. Another technique used in constraint programming is arc consistency, which iteratively reduces variable domains until a fixed state is found. Similarly to FM-elimination, finding a fixed state is NP-hard in general [2] but admits a pseudo-polynomial time algorithm [11].

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 35; pp. 35:1–35:9
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Our contribution is an algorithm that combines these techniques into a polynomial-time algorithm for another subclass of TVPI. Our class concerns monotone inequalities whose coefficients are restricted to powers of c for some fixed constant $c > 0$. That is, inequalities of the form $c^{a_1}x - c^{a_2}y \leq b$. After dividing by $c^{\min(a_1,a_2)}$, we may assume that a_1 or a_2 is zero. Formally, for any $c \in \mathbf{N}$ we define R_c as the set of inequalities of the form $x \leq c^a y + b$ or $c^a x + b \leq y$ with $a \in \mathbf{N}$ and $b \in \mathbf{Z}$, and then consider $\mathrm{CSP}(\mathbf{Z}; R_c)$. In order to keep the size of the coefficients polynomial, we require that each a is given in unary. To simplify some arguments, we view the $\mathrm{CSP}(\mathbf{Z}, R_c)$ instance as a directed, weighted graph with colored arcs. We model a constraint of the form $x \leq c^a y + b$ as a *red* arc from x to y with weight $-a$, and a constraint of the other form as a *blue* arc from x to y with weight a.

The paper is structured as follows. In Section 2 we give a high-level outline of the algorithm and its main ideas. In Section 3 we give a detailed explanation of all the substeps. Finally, in Section 4 we combine all these steps to show that our algorithm is correct and satisfies the claimed runtime. To begin, we fix a value for c and an instance of $\mathrm{CSP}(\mathbf{Z}, R_c)$. For this instance, let n be the number of variables, m the number of constraints, and W the maximal value of any coefficient in any constraint. Since each a is given in unary, $\log(W)$ is polynomial in the input size.

2 Algorithm outline

The algorithm consists of four main ideas. The first one is boundedness: if the instance admits an integer solution, then the size of the "smallest" integer solution will not be too large. More precisely, we will prove that there exists an integer Ω, whose size is polynomial in the input, such that: if there exists an integer solution to the CSP, then there exists an integer solution where every variable is bounded in absolute value by 2^Ω. In particular, this implies that to solve the CSP we only need to look for solutions in a bounded subset of $\mathbf{Z}$. Let $I_\Omega := \{-2^\Omega, -2^\Omega + 1, \ldots, 2^\Omega\}$ be this subset.

The second idea is to use arc consistency. If we know a domain of possible values for some variable x, then any arc between x and some other variable y translates this into a domain of possible values for y. By starting with the initial domain $x \in I_\Omega$ for every variable x and repeatedly checking consistency along all arcs, we will eventually settle in a state where some domain is empty or where all arcs are consistent. Now, the instance has a solution if and only if all domains are nonempty: if a domain is empty the instance is obviously unsolvable, and if all domains are nonempty then we can set every variable to the smallest remaining value. This is a solution, since the smallest remaining value on the larger side of the inequality is (by arc consistency) larger than some remaining value on the smaller side of the inequality, and thus also larger than the smallest remaining value of that side. In more technical terms, min is a polymorphism.

Since the complete domains can be exponentially large, we only keep track of a description of them. It suffices to only consider integer lower bounds, so for every variable x we keep track of a lower bound $x \geq \ell, \ell \in \mathbf{Z}$ which increases during the arc consistency steps. A domain is now considered empty if we find a lower bound of at least 2^Ω. One arc consistency step now goes as follows: loop over every arc (x, y). The current lower bound at the starting vertex x implies a new lower bound on the end vertex y. If this lower bound is stricter then the bound we had for y, update y with the new bound.

This arc consistency procedure terminates in at most $n2^{\Omega+1}$ iterations: every iteration increases one of the n lower bounds by at least one, and each lower bound is contained in I_Ω. Unfortunately, this is still exponentially many. To reduce the required number of iterations, we introduce the third main idea: edge shortening.

If we have an arc from x to y and an arc from y to z, then any lower bound on x implies a lower bound on y which in turn gives a lower bound on z. In some cases (but not all!) we could have obtained this lower bound on z directly from the one on x by adding a new constraint between x and z which is functionally equivalent to the path from x to z via y. This means that if we would have added this new arc before applying the arc consistency procedure, we would have needed one iteration less to obtain the lower bound on z.

By this reasoning, we see that the arc consistency algorithm improves if we first preprocess the graph to introduce as many new arcs as possible. We do this by performing multiple edge shortening iterations. In one iteration we check every pair of consecutive arcs and, if possible, add a new arc. We will later show how to determine when this arc exists and what its parameters are. We will also prove that one edge shortening iteration reduces the required number of arc consistency steps by (roughly) one third and that after enough edge shortening iterations, the required number of arc consistency steps reduces to just two.

The problem with edge shortening is that we might introduce exponentially many new arcs and that the arcs might have exponentially large parameters. To solve this, we apply the fourth main idea: compression. We will show that any set of constraints S can be transformed into a new set S' whose total bitsize is polynomial in Ω such that any variable assignment $s \in (I_\Omega)^n$ satisfies S if and only if it satisfies S'. The intuition is that since we only concern solutions bounded by 2^Ω, then the set of variable assignments that satisfy an arc whose coefficients are much larger that 2^Ω can also be described using an arc with smaller coefficients. Furthermore, we only need at most one arc of each weight between every two variables: if there are multiple arcs of the same weight, we only need the one with the most restrictive constant term.

Overall, we obtain the following algorithm:

1. Perform the boundedness step: compute Ω.
2. Alternate between edge shortening iterations and compression steps polynomially many times (we later specify exactly how many).
3. Perform two arc consistency steps.
4. If there is a lower bound of at least 2^Ω, output NO SOLUTION. Otherwise, output the current values of the lower bounds.

In the next section, we will describe the four main steps in more detail and prove why they work.

3 Main steps

Boundedness

We will first show how to construct the global bound Ω using results from literature. Our goal is to apply the main theorem of [8], which states that if a CSP consisting of linear equations over the integers has n variables, m constraints, and coefficients bounded by a, and admits a solution consisting of positive integers, then it admits one where each integer in the solution is at most $n(ma)^{2m+1}$.

CSP(R_c) is different: our constraints are inequalities instead of equalities and the solution may contain negative integers. Fortunately, we can easily transform it into this standard form [7, Section 2.2] by using slack variables for the inequalities and expressing each unbounded variable as the difference between two positive variables. This modification adds $m + n$ additional variables and n additional constraints, so the resulting CSP has $2n + m$ variables, $m + n$ constraints, and coefficients bounded by W.

We can now apply the above theorem and find a new solution to the modified instance where each variable is positive and bounded by $\Omega' := (2n+m)((m+n)W)^{2(m+n)+1}$. This translates into a solution of the original instance where each variable may be negative again but still has an absolute value of at most Ω'. We conclude that $\Omega := \log(\Omega') = \mathcal{O}((m+n)(\log(W(m+n))$ suffices. Overall, we obtain the following proposition.

▶ **Proposition 1.** *There exists an integer $\Omega \in \mathbf{N}$, which only depends on and is polynomial in n, m and $\log(W)$, such that: if our $\mathrm{CSP}(\mathbf{Z}, R_c)$ instance has at least one integer solution, then there exists an integer solution where each variable has an absolute value of at most 2^{Ω}.*

Arc consistency

The second step is to apply arc consistency by propagating lower bounds along the constraint arcs. On a blue arc $c^a x + b \leq y$, a lower bound $x \geq \ell$ implies a lower bound $y \geq c^a \ell + b$. On a red arc $x \leq c^a y + b$ a lower bound $x \geq \ell$ implies a lower bound $y \geq \left\lceil \frac{\ell - b}{c^a} \right\rceil$. This chains across consecutive arcs: for any path p, a lower bound on the starting vertex results in a lower bound on the ending vertex. In general, we can identify any path with a valid lower bound by propagating the trivial lower bound $x \geq -2^{\Omega}$ on the starting vertex x along the path.

Conversely, every lower bound that can be obtained using arc consistency corresponds to a path; every arc consistency step effectively appends one arc to the path. Hence, we obtain an equivalence between paths and lower bounds. When viewing a path in the context of a lower bound chain, we will refer to it as a *propagation path.* We now give some properties of propagation paths.

▶ **Proposition 2.** *Propagation paths satisfy the following properties.*

(i) *Let $x \geq \ell$ be some lower bound reachable by propagation paths. The length of the shortest propagation path resulting in this lower bound (or a better one) is at most $n2^{\Omega+1}$.*

(ii) *Let p be a propagation path of length k ending in some lower bound $x \geq \ell$. Then, the arc consistency procedure will find this lower bound (or a better one) in at most k iterations.*

Proof. We prove the parts in order.

(i) If a propagation path of minimal length contains a vertex multiple times, then the corresponding lower bound on that vertex should increase otherwise the subcycle between these occurences is redundant and the path is not minimal. So, every vertex occurs at most $|I_\Omega| = 2^{\Omega+1}$ times, which bounds the length of the path to $n2^{\Omega+1}$.

(ii) We use induction on the length of p. If p has length zero, then the lower bound must be the initial lower bound $x \geq -2^{\Omega}$ which is found before any arc consistency iteration. Now suppose p has length $k > 0$. Let $x' \geq \ell'$ be the bound that corresponds to p with the final constraint removed. By the induction hypothesis, after $k-1$ steps of the arc consistency procedure we will have found a bound $x' \geq \ell'$ or better. One more arc consistency iteration will then propagate this lower bound into a lower bound of at least ℓ for x. ◀

A corollary of this proposition is termination of the arc consistency procedure: every lower bound that can be found by lower bound propagation will be found in at most $n2^{\Omega+1}$ iterations, so after at most this many iterations the procedure terminates.

Edge shortening

We will now describe in which cases we can concatenate consecutive arcs into a new arc.

▶ **Proposition 3.** *Let e_1 and e_2 be two consecutive arcs. Let x, y, z be their endpoints such that e_1 runs from x to y and e_2 runs from y to z. Suppose that e_1 and e_2 are colored in one of the following ways: (red, red), (blue, blue), or (blue, red). Then there exists an arc e_3 from x to z such that the set of integer values for x and z that satisfy the constraint from e_3 is the same as the set of integer values for x and z for which there exists an integer value for y such that both the constraints e_1 and e_2 are satisfied.*

Proof. We begin with the case where e_1 and e_2 are both red. Let $x \leq c^{a_1}y + b_1$ and $y \leq c^{a_2}z + b_2$ be the inequalities associated with e_1 and e_2, respectively. These inequalities combine into

$$x \leq c^{a_1}y + b_1 \leq c^{a_1}\left(c^{a_2}z + b_2\right) + b_1 = c^{a_1+a_3}z + c^{a_1}b_2 + b_1$$

We now define e_3 as this constraint: $a_3 := a_1 + a_2$ and $b_3 = c^{a_1}b_2 + b_1$. One direction is easy: if for some tuple of integer values for x and z there exists a y such that e_1 and e_2 are satisfied, then this tuple by construction satisfies e_3 as well. For the inverse direction, let (v_x, v_z) be a tuple of integer values for x and z that satisfy e_3. Then, we can set y to the value $v_y := c^{a_2}v_z + b_2$ which is integer. Now, the triplet (v_x, v_y, v_z) satisfies e_1 and e_2: e_1 rewrites to the inequality for e_3 and e_2 is satisfied by construction.

The other two cases are very similar. If e_1 and e_2 are both blue, we have the inequalities $c^{a_1}x + b_1 \leq y$ and $c^{a_2}y + b_2 \leq z$ which merge into $c^{a_1+a_2}x + c^{a_2}b_1 + b_2 \leq z$, so we set e_3 to this inequality. If (v_x, v_z) is a tuple of integer assignments for x and z that satisfies e_3, then we can set $v_y := c^{a_1}v_x + b_1$ to find a triplet of integer assignments for e_1 and e_2.

Finally, if e_1 is blue and e_2 is red, then we have the inequalities $c^{a_1}x + b_1 \leq y$ and $y \leq c^{a_2}z + b_2$. These merge into $c^{a_1}x + b_1 \leq c^{a_2}z + b_2$. This is not yet the proper form to become a valid constraint, but we can modify it a bit. We consider two cases: either $a_1 > a_2$ or $a_1 \leq a_2$. In the first case, we can rewrite the inequality to $c^{a_1-a_2}x + \frac{b_1-b_2}{c^{a_2}} \leq z$. Since z and $c^{a_1-a_2}x$ are always integer, this inequality is equivalent to $c^{a_1-a_2}x + \left\lceil\frac{b_1-b_2}{c^{a_2}}\right\rceil \leq z$. This inequality has the proper form for a blue edge: $a_3 := a_1 - a_2$ is a nonnegative integer and $b_3 := \left\lceil\frac{b_1-b_2}{c^{a_2}}\right\rceil$ is integer as well.

In the case where $a_1 \leq a_2$ we can do something analogous: we instead rewrite the equation to $x \leq c^{a_2-a_1}y + \left\lfloor\frac{b_2-b_1}{c^{a_1}}\right\rfloor$ to obtain an equation for a red edge. The correctness proof of the (blue, red) case is also analogous: if v_x and v_z are integer values for x and z that satisfy e_3, then we can set y to $c^{a_1}x + b_1$ or $c^{a_2}z + b_2$ (both work!) to obtain a triple that satisfies e_1 and e_2. ◀

▶ Remark 4. The case (red,blue) does not work since we cannot always set y to an integer value. For example, in the case $c = 2$ one might think that the arcs $x \leq 2y$ and $2y \leq z$ combine into the arc $x \leq z$, but this does not satisfy the requirements: setting x and z both to 1 satisfies the inequality $x \leq z$ but there is no integer value for y that satisfies $x \leq 2y \leq z$.

We will now consider the effect of edge shortening on the length of propagation paths.

▶ **Proposition 5.** *Let $x \geq \ell$ be a lower bound that is reachable by a propagation path of length k. Then, apply one iteration of the edge shortening procedure. Now, the bound $x \geq \ell$ will be reachable by a propagation path of length at most $\lceil 2k/3 \rceil$.*

Proof. The main idea is the following observation: for any three consecutive arcs, the edge shortening iteration will shorten at least one pair. Indeed, if the first pair is (red, blue), the only configuration in which we cannot apply edge shortening, then the second pair will be (blue, blue) or (blue, red) which are both shortable. Now let p be the path of length k used to reach the lower bound and partition it into triplets of consecutive arcs. Then, after applying the edge shortening operation, we can replace every triplet with just two arcs: the arc obtained from the shortening and the arc not used in the shortening. Overall, this will reduce p to a path of length $\lceil 2k/3 \rceil$, where the ceiling function is required to account for the (at most two) leftover arcs that do not fit in the triplets. ◀

▶ **Corollary 6.** *Let $x \geq \ell$ be a lower bound that is reachable by a propagation path. After polynomially many edge shortening iterations, this lower bound will be reachable by a propagation path of length at most 2.*

Proof. Let k be the length of the initial propagation path used to obtain the lower bound $x \geq \ell$. By Proposition 2(i), k is at most $n2^{\Omega+1}$. Every edge shortening iteration replaces k with $\lceil 2k/3 \rceil$. In particular, if $k \geq 3$ then $\lceil 2k/3 \rceil \leq 2k/3 + 2/3 \leq 2k/3 + 2k/9 = 8k/9$. So, after $\log_{9/8}(n2^{\Omega+1})$ iterations, k will be reduced to at most 3. One more iteration then reduces k to at most 2. This shows that after $\log_{9/8}(n2^{\Omega+1}) + 1$ iterations, which is polynomially many, the lower bound $x \geq \ell$ is reachable by a path of length 2. ◀

▶ Remark 7. We cannot apply more edge shortening iterations to reduce the length even further: if a propagation path of length 2 is colored in the colors (red, blue) then additional edge shortening iterations have no effect on this path.

Compression

We now show how to bound the number and size of parallel edges.

▶ **Proposition 8.** *Let S be a set of constraints. If we restrict each variable x to the domain $x \in I_\Omega$, then we can modify S into an equivalent set of constraints S' satisfying:*

(i) *The (absolute) weight of each edge is bounded by $\Omega + 1$.*

(ii) *There is at most one edge of each weight between any two variables.*

(iii) *The constant coefficient of each constraint is at most $2^{2\Omega+2}$.*

In particular, the total bitsize of S' is polynomially bounded in Ω.

Proof. We only consider red arcs; blue arcs are analogous but with the direction of the inequalities reversed. Let e be a red arc, say $x \leq c^a y + b$. We perform three simplification steps on e to satisfy the three required properties.

(i) Suppose e satisfies $a > \Omega + 1$. If $c = 1$, the value of a does not matter so we can set it to 0. If $c \geq 2$, then there is at most one value of y such that the right hand side $c^a y + b$ falls in the interval $|x| \leq 2^\Omega$. Let v_y be this value, and let $v_x := c^a v_y + b$ be the corresponding upper bound for x. The set of pairs (x, y) satisfying the constraint can now be described as: $y < v_y$ is impossible, $y = v_y$ implies $x \leq v_x$, and if $y > v_y$, there are no restrictions on x. This can also be achieved with the constraint $x \leq 2^{\Omega+1}(y - v_y) + v_x = 2^{\Omega+1}y - 2^{\Omega+1}v_y + v_x$ which does have weight $\Omega + 1$ so we simply replace e with this new constraint. This argument is visualized in Figure 1: it shows the shape of the feasible region for some constraints, and in particular it shows that the feasible region for a constraint with $a > \Omega + 1$ has the same shape as one with $a = \Omega + 1$.

(ii) Suppose there is another edge e' of the same weight, say $x \leq c^a y + b'$. Assume without loss of generality that $b \leq b'$. Then the inequality for e' follows directly from the one for e, so e' is redundant and can be removed from S.

(iii) Suppose that $|b| \geq 2^{2\Omega+2}$, and recall that $|x| \leq 2^{\Omega}$, $|y| \leq 2^{\Omega}$ and $0 \leq a \leq 2^{\Omega+1}$. We find that the left hand side of $x \leq c^a y + b$ is at most 2^{Ω} in absolute value while the right hand side is at least $2^{2\Omega+2} - 2^{\Omega+1} \cdot 2^{\Omega} = 2^{2\Omega+1}$ in absolute value. This shows that the constraint is either always satisfied or always unsatisfied (depending on the sign of b). So, we find that either the instance is infeasible or the constraint can be removed without affecting feasibility. ◀

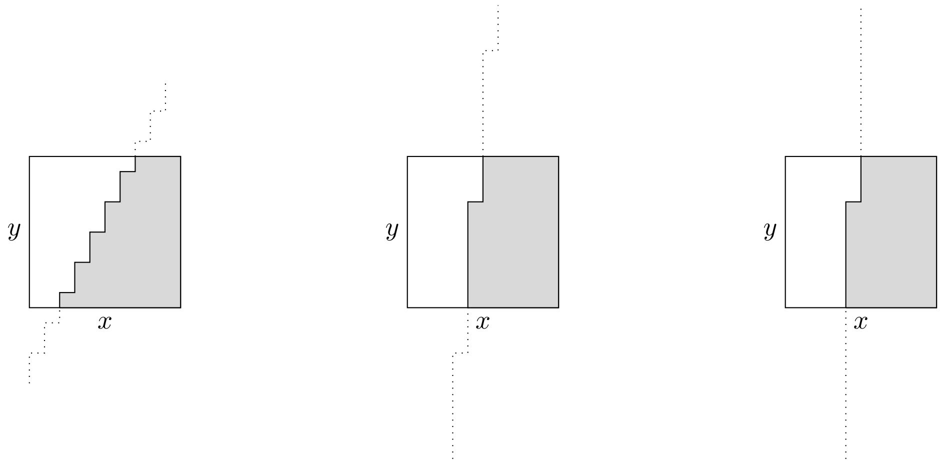


Figure 1 Some possible shapes for the feasible region of a single constraint. The square is the domain $(I_\Omega)^2$ and the wiggled line is the inequality. From left to right, the coefficients satisfy $1 < a < \Omega + 1$, $a = \Omega + 1$ and $a > \Omega + 1$.

4 Combining the parts

We will now give a formal proof of correctness for the algorithm described in Section 1 using the propositions from Section 3. This is split into two theorems, one for correctness and one for the runtime.

▶ **Theorem 9.** *The algorithm described in Section 1 returns an integer solution to the* CSP(R_c) *instance if one exists. Otherwise, it will output NO SOLUTION.*

Proof. We first show that if the algorithm returns some variable assignment, then this is indeed a solution. The final step asserts that all values of this assignment are at most 2^{Ω}, so by Proposition 8 the correctness is not affected by the compression steps. We now claim that the returned variable assignment satisfies all constraints of the modified instance obtained from the edge shortening iterations and compression steps: suppose to the contrary that there is an unsatisfied constraint. This means that one additional arc consistency step would have updated some lower bound to a new value, and most importantly that this new lower bound is a bound reachable by propagation paths. Propositions 3 and 8 show that this lower bound is also reachable in the original instance. Proposition 2(i) shows that the length of the shortest propagation path to reach this lower bound is at most $n2^{\Omega+1}$. Corollary 6 now shows that the edge shortening steps reduce it to one of length at most 2. Proposition 2(ii) then shows that this lower bound was already found in the two arc consistency steps. This is a contradiction: we conclude that the returned variable assignment satisfies all constraints of the modified instance. Since all original constraints are either still present in the modified instance or replaced by an equivalent or stricter constraint, the solution must also satisfy the original instance. This completes the first half of the proof.

We now show that if a solution exists, the algorithm will find it. By the corollary after Proposition 2, just using the arc consistency procedure would determine existence of a solution in $n2^{\Omega+1}$ steps. In particular, the assigned values in this solution are the result of a propagation path of length at most $n2^{\Omega+1}$. By Corollary 6, the edge shortening iterations reduce all these propagation paths to paths of length at most two, so by Proposition 2(ii), every assigned value will be found in the two arc consistency steps. Therefore, the solution will be found by the algorithm. This completes the second half of the proof, and consequently the full proof. ◀

▶ **Theorem 10.** *The algorithm described in Section 1 runs in polynomial time.*

Proof. We first note that Ω is polynomial in the size of the input. After each compression step, Proposition 8 shows that the current instance contains at most $2\Omega + 3$ constraints between any two variables, so at most $(2\Omega + 3)n^2$ in total, which is polynomially many. Denote this value by N. Furthermore, the coefficients of every constraint have polynomial bitsize of $\mathcal{O}(\Omega)$, so most arc operations takes $\mathcal{O}(\Omega)$ time.

Each edge shortening iteration adds at most one new arc for any two consecutive arcs, taking $\mathcal{O}(\Omega)$ time per arc pair and resulting in at most N^2 new arcs after this step. The compression step afterwards takes $\mathcal{O}(\Omega)$ time per arc to reduce the number of arcs back down to N again. Together, one edge shortening iteration and compression step take $\mathcal{O}(N^2\Omega)$ time. One exception is the first compression step: since there are initially m arcs instead of at most N, and coefficients are initially bounded by W instead of 2^Ω, the first step takes $\mathcal{O}(\log(W)m^2)$ time. Since these two steps are repeated $\log_{9/8}(n2^{\Omega+1}) + 1 = \mathcal{O}(\Omega \log(n))$ times, the overall runtime of these steps is $\mathcal{O}(\Omega^2 \log(n)N^2 + \log(W)m^2)$; still polynomial.

Finally, each arc consistency step checks every arc at most once, so this adds another $\mathcal{O}(N\Omega)$ time to the computation. Overall, we conclude that the algorithm runs in $\mathcal{O}(\Omega^2 \log(n)N^2 + \log(W)m^2)$ time. When transformed back to the original input parameters, this becomes $\mathcal{O}(\Omega^2 \log(n)(\Omega n^2)^2 + \log(W)m^2) = \mathcal{O}(\Omega^4 n^4 \log(n) + \log(W)m^2)$. Since $\Omega = \mathcal{O}((m + n)\log(W(m + n)))$ the $\log(W)m^2$ vanishes and the total runtime is $\mathcal{O}((m + n)^4 \log(W(m + n))^4 n^4 \log(n))$. ◀

5 Discussion

To conclude the paper, we have shown that the CSP with constraints of the form $c^a x + b \leq y$ and $x \leq c^a y + b$ is solvable in polynomial time. We stress that the given time complexity is far from optimal; a better runtime analysis and some optimizations will reduce the runtime significantly.

We end with some possible future research directions.

1. What are the implications on the infinite domain CSP-dichotomy over $\mathbf{Z}$? In particular, does our method generalize to CSPs whose parameters are restricted to a larger domain?
2. What is the complexity of the optimization variant of this CSP? The current algorithm outputs a solution where all variable assignments are minimal and can easily be modified into an algorithm where these are maximal. However, there is no simple reduction to find the optimal solution where we want some variables to be large and others to be small.
3. How far can we reduce the time complexity of this algorithm?

References

1 Manuel Bodirsky. *Complexity of Infinite-Domain Constraint Satisfaction*. Lecture Notes in Logic. Cambridge University Press, 2021.

2 Lucas Bordeaux, George Katsirelos, Nina Narodytska, and Moshe Vardi. The complexity of integer bound propagation. *J. Artif. Intell. Res. (JAIR)*, 40:657–676, October 2011. doi:10.1613/jair.3248.

3 Andrei A. Bulatov. A dichotomy theorem for nonuniform csps. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 319–330, 2017. doi:10.1109/FOCS.2017.37.

4 Dorit S. Hochbaum and Joseph (Seffi) Naor. Simple and fast algorithms for linear and integer programs with two variables per inequality. *SIAM Journal on Computing*, 23(6):1179–1192, 1994. doi:10.1137/S0097539793251876.

5 Joxan Jaffar, Michael J. Maher, Peter J. Stuckey, and Roland H. C. Yap. Beyond finite domains. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, pages 86–94, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

6 J. C. Lagarias. The computational complexity of simultaneous diophantine approximation problems. *SIAM J. Comput.*, 14(1):196–209, 1985. doi:10.1137/0214016.

7 K.G. Murty. *Linear Programming*. Wiley, 1983.

8 Christos H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, October 1981. doi:10.1145/322276.322287.

9 Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 216–226, New York, NY, USA, 1978. Association for Computing Machinery. doi:10.1145/800133.804350.

10 Piotr Wojciechowski and K. Subramani. A faster algorithm for determining the linear feasibility of systems of btvpi constraints. In Leszek Gąsieniec, editor, *SOFSEM 2023: Theory and Practice of Computer Science*, pages 313–327, Cham, 2023. Springer International Publishing.

11 Zhang Yuanlin and Roland H. C. Yap. Arc consistency on n-ary monotonic and linear constraints. In Rina Dechter, editor, *Principles and Practice of Constraint Programming – CP 2000*, pages 470–483, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

12 Dmitriy Zhuk. A proof of CSP dichotomy conjecture. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 331–342, October 2017. doi:10.1109/FOCS.2017.38.

Black-Box Value Heuristics for Solving Optimization Problems with Constraint Programming

Augustin Delecluse
TRAIL, ICTEAM, UCLouvain, Belgium

Pierre Schaus
ICTEAM, UCLouvain, Belgium

Abstract

Significant research efforts have focused on black-box variable selection, with less attention given to value heuristics. An ideal value heuristic enables depth-first-search to prioritize high-quality solutions first. The Bound-Impact Value Selection achieves this goal through a look-ahead strategy, trying every value of the selected variable and ranking them based on their impact on the objective. However, this method is generally too computationally intensive for the entire search tree. We introduce two simple yet powerful modifications to improve its scalability. First, a lighter fix point computation involving only the constraints on the shortest path in the constraint graph between the variable and the objective. Second, a reverse look-ahead strategy optimistically fixes the objective variable to its minimum in order to prioritize the remaining values. These two ideas have been empirically validated on a range of academic problems and in the XCSP^3 competition, demonstrating significant improvements in scalability.

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

Keywords and phrases Constraint Programming, Value Selection, Look-Ahead, Optimization

Digital Object Identifier 10.4230/LIPIcs.CP.2024.36

Category Short Paper

Supplementary Material *Software (Source Code)*: `https://github.com/augustindelecluse/choco-solver`, archived at `swh:1:dir:6ef58dcd7be2a19c98a7dbb0cdaafa41f134f2ab`

Funding *Augustin Delecluse*: This work was supported by Service Public de Wallonie Recherche under grant n°2010235 – ARIAC by DIGITALWALLONIA4.A.

1 Introduction

The Constraint Programming (CP) paradigm enables to solve combinatorial problems in a declarative way. The Holy Grail vision of CP is that the user simply has to state the problem [5]. However, for practical efficiency, this vision has been adjusted to the long-standing CP mantra: *CP = Model + Search*. The capability to program problem-specific searches remains crucial in CP to minimize the search tree size. Over time, the emphasis on search programming has diminished due to the development of effective black-box search strategies by the CP community [7, 4, 13, 20, 27]. A common search procedure involves a two-step decision-making process at each node: selecting an undecided variable and then choosing a value to assign from the domain on the left branch, which is removed upon backtracking. Despite extensive work on deriving efficient problem-independent variable selection heuristics based on the first-fail principle, few black-box strategies have been proposed for value selection. In the context of optimization problems, a simple yet effective generic method is the Bound-Impact Value Selector (BIVS) [12]. This look-ahead heuristic iterates over each value in the domain of a variable, successively assigning the variable to each of these values. After each assignment is made, the fixpoint filtering by the constraints is computed. The impact on the objective–specifically, the lower bound increase in the case of a minimization problem–is then measured. The value that yields the to the smallest increase for minimization problem is selected for the left assignment.[1]

Although BIVS finds solutions of better quality compared to other naive strategies, such as picking the minimum value in the domain of a variable, its computational cost does not always compensate its benefits in guiding the search toward promising directions. This is especially true for problems with large domain sizes or when a large number of constraints are involved in the fixpoint computation. To overcome this shortcoming, one tradeoff is to consider only the smallest and largest values from the domain, instead of every value, especially if the domain size is too large. This is the default value selection performed in Choco-solver [26].

Our contribution introduces two ideas to reduce the computational cost of BIVS. The first, called the *Restricted Fixpoint* is to compute the fixpoint on a restricted set of constraints. Specifically, we measure the impact only on a fixpoint that involves the constraints along the shortests paths from the selected variable to the objective variable in a constraint graph, where node constraints are connected to node variables in their scope.

The second improvement, called *Reverse Look-Ahead* starts from the objective assignment. Instead of sequentially fixing the selected variable to each value, we suggest optimistically setting the objective variable to its lower bound (assuming minimization) and then computing the fixpoint. The selected value is one of those that remain. If infeasibility is detected, we increase the objective bound and continue until no domain is empty.

The two proposed improvements are designed to be non-intrusive, meaning that they can be implemented without necessitating modifications to the solver's architecture.

The paper is organized as follows. Section 2 presents first the main concepts necessary to understand BIVS, before introducing the two ideas to lower the cost of BIVS, namely the Restricted Fixpoint and Reverse Look-Ahead. Their performances are assessed in section 3.

2 Reducing BIVS Cost

The search heuristic is generally decomposed in two stages for efficiency reason: the variable is selected, then the value in the domain of the variable. For the variable selection, many ideas inspired by the *first-fail* principle "to succeed, try first where you are most likely to fail" [16] have been proposed such as choosing the variable with the smallest ratio of size of domain versus number of failures when running the fixpoint on them (`dom/wdeg`) [7, 4], the variable involved in the most recent failures [13, 20, 21], the variable whose assignment has the largest estimated search space reduction [27], the variable with the most frequent domain changes [24, 4] etc.

Once the variable to branch on has been selected, a value must be assigned to it. In opposition to the *first-fail* principle, the *succeed-first* principle is used instead [14, 10]. The idea is to select the value being most likely to participate in a solution. Compared to variable selection, only a few generic selectors have been proposed for picking a value, which is why choosing the smallest value from the domain of a variable remains a popular default choice [19, 29]. Two exceptions are Belief Propagation (BP) [25], aimed at driving the search towards areas with a large number of solutions, or Bound-Impact Value Search (BIVS) [12], choosing the value with the best impact on the objective. Even though those two methods

[1] This idea is similar to *Strong Branching* in Mixed-Integer Linear Programming solvers [2].

do reduce a lot of the search space, their computational cost is too large to make them the default choice in most solver during the whole search procedure. Moreover, BP is better suited for constraint satisfaction problems rather than optimization problems. Research on machine learning for value selection heuristics [22, 9] is limited by extensive data requirements for lengthy training phases. While Objective Landscapes methods [18] collecting pre-search impacts work well in stable settings like scheduling, they falter in dynamic scenarios where impacts fluctuate with other assignments. An effective alternative is *phase-saving*, selecting values from previous solutions [11], though it relies on a default heuristic when no prior value is applicable.

Our work revisits the BIVS algorithm, detailed in Algorithm 1, to reduce its computational cost while maintaining its effectiveness in finding high-quality solutions. BIVS examines each value v in the domain of the selected variable $x \in \mathcal{X}$ (line 3), assigning v to x (line 5) and running the fixpoint algorithm (line 6). If the fixpoint is error-free and improves the objective's lower bound ($\lfloor \mathcal{D}(obj) \rfloor$), the value is considered for retention (line 7). The process is encapsulated within `saveState`$(\mathcal{X}, \mathcal{C})$ and `restoreState`$(\mathcal{X}, \mathcal{C})$ operations (lines 4 and 10), ensuring that the solver's state is reset before each new trial[2]. While this method is effective in steering the search towards high-quality solutions, the time complexity for selecting a value for variable x in the BIVS algorithm is $\Theta(\mathcal{F} \cdot |\mathcal{D}(x)|)$, with $\mathcal{F}$ as the fixpoint algorithm's complexity. As [4] notes, controlling this high computation cost is challenging. We propose two methods to reduce this complexity: one by lowering the cost of the fixpoint algorithm and the other by reducing the frequency of its calls. However, these changes do not guarantee identical outcomes as the original BIVS.

Algorithm 1 Bound-Impact Value Selector (assuming minimization), adapted from [12].

```
   Input  : $\mathcal{X}$: variables, $\mathcal{C}$: constraints, $x$: branching variable, $obj$: objective variable
   Ouput : bestV, the value to assign to the variable $x$.
1  bestV ← $\lfloor \mathcal{D}(x) \rfloor$
2  bestBound ← $\infty$
3  for $v \in \mathcal{D}(x)$ do
4      saveState($\mathcal{X}, \mathcal{C}$)
5      $\mathcal{D}(x) \leftarrow \{v\}$
6      success ← fixpoint($\mathcal{X}, \mathcal{C}$)
7      if success and $\lfloor \mathcal{D}(obj) \rfloor <$ bestBound then
8          bestBound ← $\lfloor \mathcal{D}(obj) \rfloor$
9          bestV ← $v$
10     restoreState($\mathcal{X}, \mathcal{C}$)
11 return bestV
```

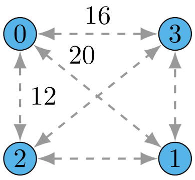
2.1 Restricted Fixpoint Computation

The Restricted Fixpoint (RF) approach assesses the impact on the objective by focusing solely on a limited set of constraints. Specifically, it considers only the constraints that are located on the shortest paths from the selected variable to the objective variable within a bipartite variables-constraints graph. In this graph, variables and constraints are nodes,

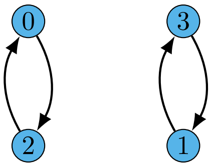
[2] See [23] for state saving and restoration with a trail.

connected by edges if the variable is within the scope of the constraint. For each variable, we precompute all constraints on these paths. This method reduces computation costs as it involves fewer constraints, but it may be less informative and risk missing potential failures.

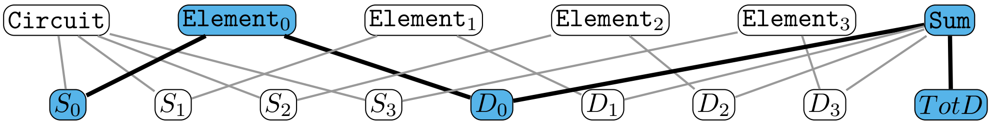
▶ **Example 1.** Consider the Traveling Salesman Problem (TSP), where a salesman needs to visit a set of cities, each city being visited exactly once, while minimizing the traveled distance. Given n cities and a distance matrix $d \in \mathbb{Z}^{n\times n}$, a commonly used CP model is minimize $TotD$ such that $\texttt{Circuit}(S)$, $D_i = \texttt{Element}(d_{i,*}, S_i)$ $\forall i \in \{0..n-1\}$, and $TotD = \texttt{Sum}(D)$. It introduces two variables per city: S_i is the visit occurring after the city i in the tour, and D_i the distance between city i and its successor S_i. The circuit constraint enforces every city to be visited within a single tour. The distance D_i between a city i and its successor S_i corresponds to the S_i-th entry within the line i in the distance matrix d, enforced with an element constraint. The sum of traveled distance $TotD$ is the objective to minimize. A TSP with 4 cities is shown in Figure 1 and the graph in Figure 3. When selecting values for S_0 using RF with BIVS, only 2 constraints are considered per iteration, compared to 6.



■ **Figure 1** A TSP instance with 4 cities to visit. The distances are shown on the edges.



■ **Figure 2** When fixing $\mathcal{D}(TotD)$ to $\{48\}$, all successors points towards their nearest city. This violates the $\texttt{Circuit}$ constraint.



■ **Figure 3** Constraints and variables of the TSP instance from Figure 1. Variables and constraints in blue are located on the shortest path to the objective $TotD$ when considering variable S_0.

2.2 Reverse Look-Ahead

The Reverse Look-Ahead (RLA) strategy reduces the number of calls to the fixpoint computation by restricting optimistically the domain of the objective and observing the effects on the variable to $\mathcal{D}(x)$, rather than fixing x directly. It is similar to the Destructive Lower Bound used in scheduling [17] and can also tighten bounds on the objective.

Algorithm 2 outlines RLA. A value δ controls the domain size of the objective variable during the fixpoint computation, starting with a value of 1 to fix the objective to its minimum value. Several iterations may be performed, each increasing δ until the fixpoint computation succeeds. At this point, the minimum value from the domain of x is returned at line 11. The value of δ doubles at each iteration, resulting in an exponential evolution. Note that when the fixpoint computation fails, the lower bound of the domain of the objective variable can be safely increased (line 14).

Algorithm 2 Reverse Look-Ahead (assuming minimization).

```
   Input  : $\mathcal{X}$: variables, $\mathcal{C}$: constraints, $x$: branching variable, $obj$: objective variable
   Ouput : $success$: boolean indicating node expandability, $v$: assigned value for $x$
1  $\delta \leftarrow 1$
2  success $\leftarrow$ true
3  while success do
4      saveState($\mathcal{X}$, $\mathcal{C}$)
5      $\lceil\mathcal{D}(obj)\rceil \leftarrow \min(\lceil\mathcal{D}(obj)\rceil, \lfloor\mathcal{D}(obj)\rfloor - 1 + \delta)$
6      success $\leftarrow |\mathcal{D}(obj)| > 0$
7      if success then
8          if fixpoint($\mathcal{X}$, $\mathcal{C}$) then
9              $v \leftarrow \lfloor\mathcal{D}(x)\rfloor$
10             restoreState($\mathcal{X}$, $\mathcal{C}$)
11             return (true, $v$)
12         else
13             restoreState($\mathcal{X}$, $\mathcal{C}$)
14             $\lfloor\mathcal{D}(obj)\rfloor \leftarrow \lfloor\mathcal{D}(obj)\rfloor + \delta$
15             success $\leftarrow |\mathcal{D}(obj)| > 0$
16             $\delta \leftarrow \delta * 2$
17     else
18         restoreState($\mathcal{X}$, $\mathcal{C}$)
19 return (false, 0)
```

▶ **Example 2.** Consider the same situation as Example 1, shown in Figure 1. The initial fixpoint yields $\mathcal{D}(D_i) = \{12, 16, 20\}\ \forall i \in \{0..n-1\}$ and $\mathcal{D}(TotD) = \{48, \ldots, 60\}$. When using RLA to choose a value for S_0, the following iterations occur:

1. The fixpoint is triggered with $\mathcal{D}(TotD) = \{48\}$. This means that the successor of every city must be the closest city, violating the `Circuit` constraint (cf Figure 2). The iteration fails and the lower bound of the objective is now set to 49 for the sub-tree to consider.
2. $\delta = 2$ and $\mathcal{D}(TotD) = \{49, 50\}$. Similarly, this fails and sets the lower bound to 51.
3. $\delta = 4$ and $\mathcal{D}(TotD) = \{51, \ldots, 54\}$. The fixpoint proceeds without failure, resulting in $\mathcal{D}(S_0) = \{2, 3\}$. Value 2, the nearest neighbor, is picked and the state is restored while keeping the lower bound of the objective to 51. Finally, 2 is returned (lines 9 to 11).

The time complexity of RLA is $\Omega(\mathcal{F})$ in the best case, if only one iteration needs to be performed, and $\mathcal{O}(\mathcal{F} \cdot \log_2 |\mathcal{D}(obj)|)$ in the worst case. Moreover, RF can also be used with RLA, meaning that the complexity of the fixpoint can be lowered.

▶ **Example 3.** We reuse the model from Example 2. Initially, RLA restricts $\mathcal{D}(TotD) = \{48\}$ and runs the RF. This scenario suggests all successors should be nearest neighbors, which is infeasible given the `Circuit` constraint (see Figure 2). However, when considering only the shortest path constraints (the `Sum` constraint and one `Element` constraint, cf Figure 3), the failure is missed. Consequently, the domain of S_0 becomes $\{2\}$ (its closest neighbor), and 2 is returned. Compared to the scenario in Example 2, only one iteration has been performed (being less costly) but the heuristic itself could not tighten the bounds of the objective.

3 Experiments

To assess the performances of our methods, we consider two main settings. The first one analyzes three classical discrete problems easily modeled with CP. The second one reports the performances on various optimization problems, using the XCSP^3 2023 competition [8, 3].

All experiments were conducted using two Intel(R) Xeon(R) CPU E5-2687W. The implementation was done in Java in the Choco-Solver (version 4.10.5), a state-of-the-art general purpose constraint programming solver [26]. In all settings, instances needing more than 32GB were discarded. The variable selection used is DomWDeg [7] combined with last conflict [20], a popular default selection. The timeout was set to 30 minutes.

3.1 Fundamental Problems

Three fundamental problems are studied. (i) The TSP and the instances from TSPLib [28]. (ii) The JobShop and the instances from [30]. (iii) The Quadratic Assignment Problem and the instances from [1]. For each model, the standard models are used. The JobShop model branches on precedences like in [15]. For each model a custom white-box value heuristic is used called Greedy in the results. In total, the value selectors analyzed are:

Min choosing the smallest value in the domain of the variable.

BIVS the original algorithm as proposed in [12], using the author's implementation in Choco-solver. In this implementation, when the domain size of a variable is larger than 100, only the minimum and maximum values of the domain are considered by BIVS.

BIVS+RF indicates BIVS but with the restricted fixpoint, presented in section 2.1.

RLA depicts the Reverse Look-Ahead.

RLA+RF depicts the Reverse Look-Ahead using the restricted fixpoint.

One criterion used to compare the value selection heuristics is the primal gap introduced in [6]. It gives a value between 0 and 1 measuring the gap between the value of a solution found *obj* and the best found solution obj_{opt}. A value close to 0 means that the solution found is the best one found, while a value of 1 indicates that no solution was found.

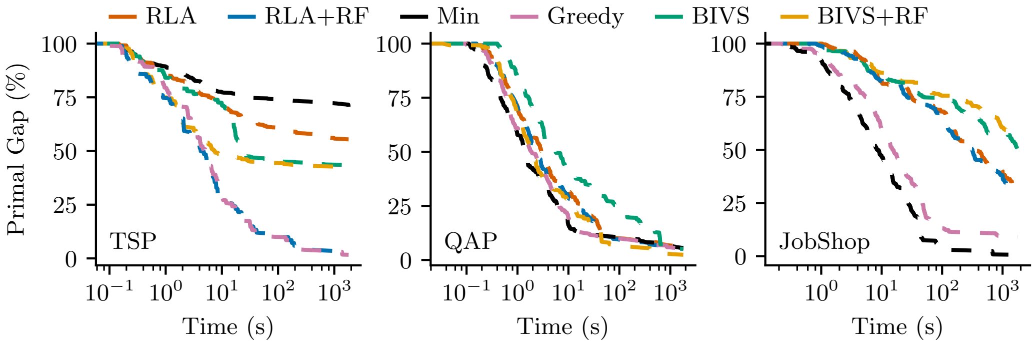


Figure 4 Primal gap in percentage over time.

Heuristic performance varies by problem (Figure 4). For TSP, RLA+RF matches the greedy heuristic, while BIVS+RF lags slightly due to considering bounds for domain sizes over 100; without this restriction, BIVS+RF performs comparably to RLA+RF. In QAP, BIVS and its RF variant outperform others, with RF significantly speeding up BIVS. For JobShop, RLA surpasses BIVS but is less effective than MIN due to the cost of fixpoint calls.

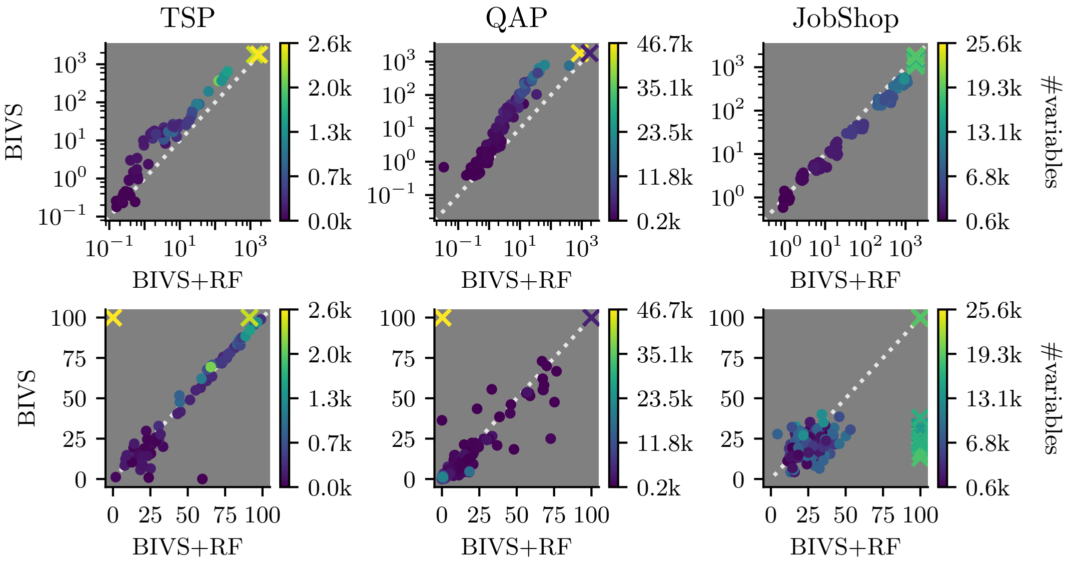


Figure 5 Comparison of BIVS and BIVS+RF regarding the time to find the first feasible solution (top, in seconds) and its corresponding gap (bottom, in percentage). Each dot represents an instance across problems: TSP (left), QAP (center), and JobShop (right). Dots on the diagonal indicate equal performance between methods. Dots above the diagonal show that BIVS was slower or found poorer solutions. Crosses denote timeouts by at least one method, resulting in a 100% gap.

The addition of RF to BIVS results in speedups for TSP and QAP, maintaining average solution quality across instance sizes, as shown in Figure 5 and supported by Figure 4 where BIVS falls behind BIVS+RF. Conversely, RF slows performance on JobShop. On this problem, the cost of scanning all constraints and potentially deactivating some, to save time on the two iterations performed by BIVS (as the precedence variables have a domain of size 2) does not offer benefits compared to considering all constraints.

3.2 XCSP^3

We consider instances from the XCSP^3 COP 2023 competition [8, 3]. The instances requiring more than 32GB were discarded, leaving 18 problems and 232 instances.

Table 1 displays gap and number of solutions found per problem, with the average gap over time shown in Figure 6. Performance varies greatly across problems, with no single heuristic outperforming others universally. However, adding RF to BIVS reduces the average gap and aids in finding solutions missed otherwise. Similar benefits, though smaller, are observed with RLA. On average, RLA outperforms BIVS, with RF enhancing both methods. Min excels in finding feasible solutions, mostly attributed to its constant time complexity, allowing for better learning from variable selection.

Moreover, Figure 6 presents two Virtual Best Solver entries: “VBS (old)” computed from previous selectors available in Choco (BIVS, Min, middle, Max and random domain selection) and “VBS (new)” adding RLA and RF-based methods. The 3.87% increase in the final gap demonstrates that RLA and RF explore the search space differently than traditional heuristics, enhancing portfolio efficiency. The figure also compares BIVS(+RF) with BIVS*(+RF), where the latter considers all domain values – not just the bounds when the domain size exceeds 100 – showing improved performance with RF, suggesting the removal of the domain size consideration. Both BIVS+RF and BIVS*+RF outperform their non-RF versions, indicating their superior efficiency. Notably, the average gap by BIVS+RF at 100.0s is matched by BIVS only at 938.32s, demonstrating its significant speed advantage.

Table 1 Performances between the methods for each problem. Each column shows the average primal gap over all instances, in percentage, and the instances where at least one feasible solution was found, in parentheses. Best results are highlighted in bold if at least one heuristic was outperformed.

Problem (#instances)	Min	BIVS	BIVS+RF	RLA	RLA+RF
AircraftAssemblyLine (20)	95.00 (1)	**90.00 (2)**	100.00 (0)	95.00 (1)	95.00 (1)
CarpetCutting (20)	61.29 (9)	57.17 (9)	65.48 (8)	62.67 (8)	**54.48 (11)**
GBACP (20)	78.86 **(11)**	**61.43** (8)	79.69 (10)	69.54 (9)	85.51 (9)
GeneralizedMKP (15)	49.74 **(15)**	**6.78** (14)	6.81 (14)	26.57 (12)	20.26 (13)
HCPizza (10)	**33.56** (10)	34.91 (10)	34.43 (10)	34.67 (10)	34.62 (10)
Hsp (18)	**0.00 (18)**	5.56 (17)	5.56 (17)	5.56 (17)	**0.00 (18)**
KMedian (15)	50.84 (8)	57.32 (7)	**43.96 (9)**	56.01 (7)	50.84 (8)
KidneyExchange (14)	44.63 **(14)**	85.71 (3)	51.11 (10)	**43.68** (13)	52.79 (10)
LargeScaleScheduling (9)	66.76 **(6)**	66.83 **(6)**	66.83 **(6)**	45.56 (5)	**34.44 (6)**
PSP1 (8)	100.00 (0)	100.00 (0)	**87.50 (1)**	100.00 (0)	100.00 (0)
PSP2 (8)	**87.50** (1)	**87.50** (1)	87.62 (1)	**87.50** (1)	87.66 (1)
ProgressiveParty (7)	57.14 (3)	57.14 (3)	57.14 (3)	57.14 (3)	57.14 (3)
RIP (12)	5.06 (12)	6.31 (12)	4.59 (12)	**3.24** (12)	4.78 (12)
Rulemining (9)	100.00 (0)	100.00 (0)	100.00 (0)	100.00 (0)	100.00 (0)
SREFLP (15)	7.27 (15)	**3.08** (15)	7.44 (15)	8.78 (15)	7.72 (15)
Sonet (16)	**1.40** (16)	2.28 (16)	3.42 (16)	2.42 (16)	3.15 (16)
TSPTW1 (8)	87.81 **(1)**	100.00 (0)	87.84 **(1)**	87.81 **(1)**	**87.50 (1)**
TSPTW2 (8)	**75.19 (2)**	87.50 (1)	**75.19 (2)**	75.42 **(2)**	75.24 **(2)**
All (232)	52.02 **(142)**	51.04 (124)	50.23 (135)	49.85 (132)	**49.52** (136)

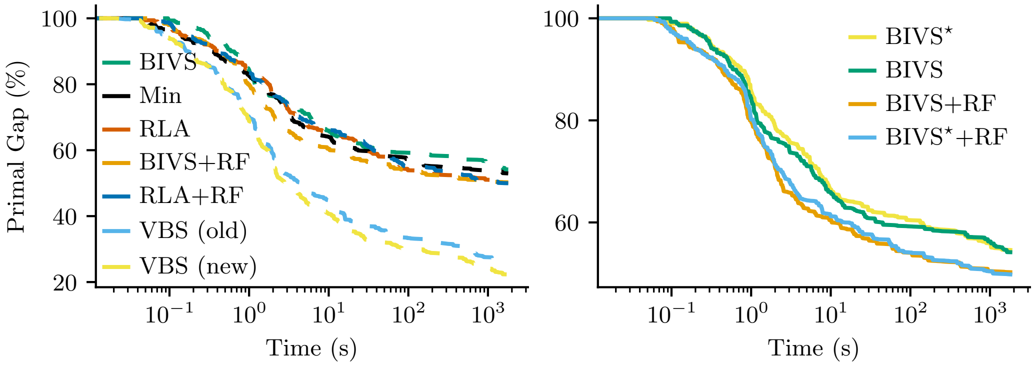


Figure 6 Average primal gap over time on the XCSP^3 instances, in percentage. The right part shows only 4 selectors compared to the left part, and their y-scale differs.

4 Conclusion

Deriving effective, generic value heuristics that balance speed and informativeness remains challenging. Bound-Impact Value Search (BIVS) stands out among these approaches, yet its cost limited its applicability in certain scenarios. By incorporating a restricted fixpoint in the look-ahead process and employing a reverse look-ahead strategy, we significantly reduced costs, making previous restrictions on BIVS usage less relevant. The methods we proposed do not require any training, are well suited for black-box settings, and substantially improve performance. When utilized alone or within a portfolio approach, these strategies continue to enhance the efficiency of solving constrained optimization problems.

References

1 QAPLIB-Problem Instances and Solutions – COR@L, April 2024. [Online; accessed 5. Apr. 2024]. URL: `https://coral.ise.lehigh.edu/data-sets/qaplib/qaplib-problem-instances-and-solutions`.

2 David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Finding cuts in the tsp (a preliminary report), 1995.

3 Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2023 xcsp3 competition. *arXiv preprint arXiv:2312.05877*, 2023.

4 Gilles Audemard, Christophe Lecoutre, and Charles Prud'Homme. Guiding backtrack search by tracking variables during constraint propagation. In *International Conference on Principles and Practice of Constraint Programming*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

5 Roman Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of the Week of Doctoral Students (WDS99)*, volume 4, pages 555–564. MatFyzPress Prague, 1999.

6 Timo Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6):611–614, 2013.

7 Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.

8 Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3: an integrated format for benchmarking combinatorial constrained problems. *arXiv preprint arXiv:1611.03398*, 2016.

9 Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre A Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3677–3687, 2021.

10 Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial intelligence*, 34(1):1–38, 1987.

11 Emir Demirović, 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 *Principles and Practice of Constraint Programming: 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings 24*, pages 99–108. Springer, 2018.

12 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. IEEE, 2017.

13 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.

14 P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of ECAI'92*, pages 31–35, 1992.

15 Diarmuid Grimes, Emmanuel Hebrard, and Arnaud Malapert. Closing the open shop: Contradicting conventional wisdom. In *International conference on principles and practice of constraint programming*, pages 400–408. Springer, 2009.

16 Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

17 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, 1999.

18 Philippe Laborie. Objective landscapes for constraint programming. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26–29, 2018, Proceedings 15*, pages 387–402. Springer, 2018.

19 Christophe Lecoutre. Ace, a generic constraint solver. *arXiv preprint arXiv:2302.05405*, 2023.

20 Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict (s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.

21 Hongbo Li, Minghao Yin, and Zhanshan Li. Failure based variable ordering heuristics for solving csps (short paper). In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2021.

22 Tom Marty, Tristan François, Pierre Tessier, Louis Gautier, Louis-Martin Rousseau, and Quentin Cappart. Learning a Generic Value-Selection Heuristic Inside a Constraint Programming Solver. 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 25:1–25:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.CP.2023.25`.

23 Laurent Michel, Pierre Schaus, and Pascal Van Hentenryck. Minicp: a lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1):133–184, 2021.

24 Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28–June1, 2012. Proceedings 9*, pages 228–243. Springer, 2012.

25 Gilles Pesant. From support propagation to belief propagation in constraint programming. *Journal of Artificial Intelligence Research*, 66:123–150, 2019.

26 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`.

27 Philippe Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming–CP 2004: 10th International Conference, CP 2004, Toronto, Canada, September 27-October 1, 2004. Proceedings 10*, pages 557–571. Springer, 2004.

28 Gerhard Reinelt. TSPLIB–a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

29 Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. Modeling and programming with gecode. *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael*, 1, 2010.

30 Eva Vallada, Rubén Ruiz, and Jose M Framinan. New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research*, 240(3):666–677, 2015.

A Additional overview of the XCSP^3 results

Figure 7 shows the cumulated number of instances solved to optimality over time. Table 2 shows the number of optimality proven and best bound founds, for each problem. We can also see that the best heuristic changes depending on the problems, highly impacting the overall readings (for instance BIVS being the best methods regarding the number of best bounds is mostly due to its good performances on the SREFLP problem).

For ranking the solvers in the XCSP^3 competition, one criterion used is a score, telling the number of times a method gave the best known results, compared to its competitors. For each instance and each heuristic:

- 1 point is awarded for proving the unsatisfiability of an instance;
- 0 point is won for proving a solution with a worse bound than its competitors
- 1 point for proving the optimality of a solution
- 1 point is won for providing (a solution with) the best bound on an instance without proving optimality. In case where another solver proved the optimality of this bound, only 0.5 points are granted.

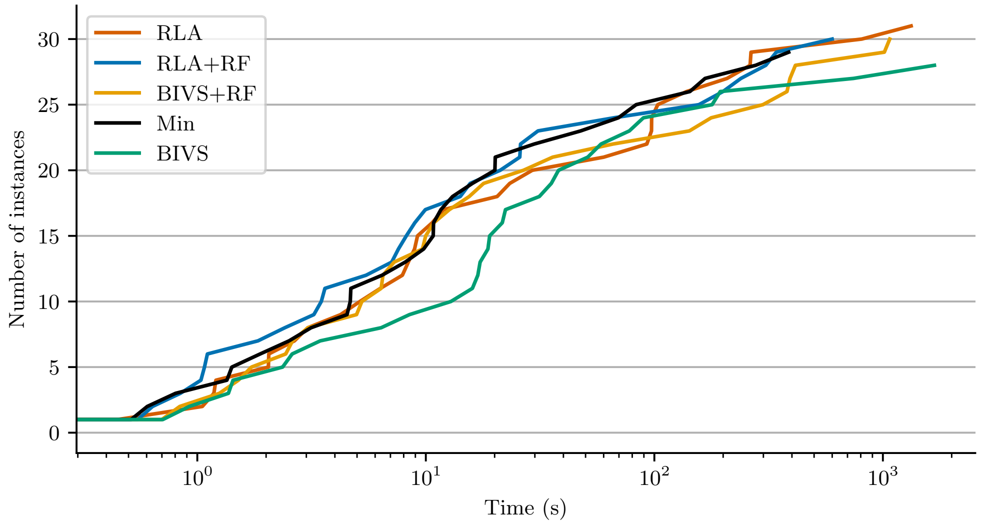


Figure 7 Cactus plot showing the cumulated number of XCSP^3 instances solved to optimality for each method.

Table 2 Performances between the methods for each problem. Each column shows first the number of instances where optimality was proven, and then number of best bounds found, in parentheses.Best results are highlighted in bold if at least one heuristic was outperformed.

Problem (#instances)	Min	BIVS	BIVS+RF	RLA	RLA+RF
AircraftAssemblyLine (20)	0 (1)	0 **(2)**	0 (0)	0 (1)	0 (1)
CarpetCutting (20)	2 (4)	**4 (9)**	2 (4)	3 (6)	2 (4)
GBACP (20)	0 (1)	0 **(7)**	0 (1)	0 (2)	0 (0)
GeneralizedMKP (15)	0 (1)	0 (10)	0 **(12)**	0 (0)	0 (1)
HCPizza (10)	0 **(9)**	0 (3)	0 (4)	0 (2)	0 (3)
Hsp (18)	**17 (18)**	16 (17)	**17** (17)	**17** (17)	**17 (18)**
KMedian (15)	0 (2)	0 (2)	0 **(5)**	0 (1)	0 (3)
KidneyExchange (14)	**2** (3)	1 (2)	**2 (8)**	**2** (4)	**2** (4)
LargeScaleScheduling (9)	0 (0)	0 (0)	0 (0)	0 **(3)**	0 **(3)**
PSP1 (8)	0 (0)	0 (0)	0 **(1)**	0 (0)	0 (0)
PSP2 (8)	0 **(1)**	0 **(1)**	0 (0)	0 **(1)**	0 (0)
ProgressiveParty (7)	3 (3)	3 (3)	3 (3)	3 (3)	3 (3)
RIP (12)	2 (4)	2 (3)	**3 (6)**	**3 (6)**	**3** (5)
Rulemining (9)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
SREFLP (15)	1 (3)	1 **(12)**	1 (1)	1 (2)	1 (1)
Sonet (16)	1 **(13)**	1 (11)	1 (8)	1 (11)	1 (8)
TSPTW1 (8)	0 (0)	0 (0)	0 (0)	0 (0)	0 **(1)**
TSPTW2 (8)	**1** (1)	0 (1)	**1** (1)	**1** (1)	**1** (1)
All (232)	29 (64)	28 **(83)**	30 (71)	**31** (60)	30 (56)

The evolution of the score over time is shown in Figure 8. The score of an heuristic at time t may decrease, for instance if the method provided the best bound to an instance before time t, but a better solution was provided at t by another selector.

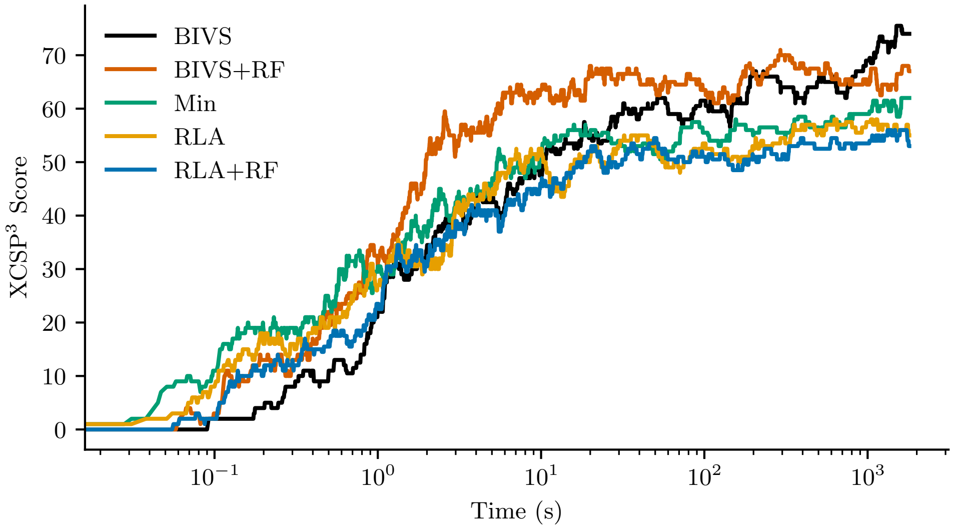


Figure 8 XCSP^3 score over time on the XCSP^3 instances, for each method.

B TSP, QAP and JobShop description

This section describes the models used for the TSP, the QAP and the Jobshop, as well as the decision variables considered and the greedy heuristics that were implemented.

TSP: the model is the same as in Example 1

JobShop: a scheduling problem where jobs must be planned, each job being decomposed into tasks having precedence between them, and each tasks being assigned to a given machine. No task can overlap on a machine and the goal consists in minimizing the completion time of the latest task.

QAP: an assignment problem where facilities must be opened at given locations, in order to minimize the sum of distances multiplied by the flow between facilities.

Note that, for all those problems, a first solution satisfying the constraints can always be easily derived. In the case of the TSP, any permutation of nodes corresponds to a valid tour for the salesman. The same reasoning holds for any permutation of facility location for the QAP. Regarding the JobShop, it suffices to provide a timing horizon long enough so that a schedule can always be constructed. Those first solutions being easy to construct, we have implemented a custom greedy value selection heuristic for those problems in order to compare white-box approaches to our own black-box approaches. More precisely, the variables considered for selection and the greedy value selection are as follows:

TSP: the decision variables are the n successor variables in a TSP tour. The greedy value selection consists in visiting the closest city.

JobShop: the decision variables to consider are the precedences between two tasks executed on the same machine. Each variable is a boolean telling, for a given machine, whether a tasks comes before of after another one. Once all precedences between tasks are fixed, the makespan can be assigned to its lower bound in order to produce a valid solution. Given a precedence variable, the greedy heuristic consists in choosing the value producing the most slack, the slack describing how much time is still available between the two tasks.

QAP: with n facilities, the n decision variables are the locations where the facilities will be opened. Given a facility not assigned to a location, the greedy heuristic choose its location as the one minimizing the weighted flow with the other facilities already placed.

Computing Small Rainbow Cycle Numbers with SAT Modulo Symmetries

Markus Kirchweger
Algorithms and Complexity Group, TU Wien, Austria

Stefan Szeider
Algorithms and Complexity Group, TU Wien, Austria

Abstract

Envy-freeness up to any good (EFX) is a key concept in Computational Social Choice for the fair division of indivisible goods, where no agent envies another's allocation after removing any single item. A deeper understanding of EFX allocations is facilitated by exploring the rainbow cycle number ($R_f(d)$), the largest number of independent sets in a certain class of directed graphs. Upper bounds on $R_f(d)$ provide guarantees to the feasibility of EFX allocations (Chaudhury et al., EC 2021).

In this work, we precisely compute the numbers $R_f(d)$ for small values of d, employing the SAT modulo Symmetries framework (Kirchweger and Szeider, CP 2021). SAT modulo Symmetries is tailored specifically for the constraint-based isomorph-free generation of combinatorial structures. We provide an efficient encoding for the rainbow cycle number, comparing eager and lazy approaches. To cope with the huge search space, we extend the encoding with invariant pruning, a new method that significantly speeds up computation.

2012 ACM Subject Classification Mathematics of computing → Extremal graph theory; Software and its engineering → Constraint and logic languages; Hardware → Theorem proving and SAT solving; Mathematics of computing → Graph enumeration

Keywords and phrases EFX, rainbow cycle number, SAT modulo Symmetries, combinatorial search

Digital Object Identifier 10.4230/LIPIcs.CP.2024.37

Category Short Paper

Supplementary Material
Software (Source Code): `https://github.com/markirch/sat-modulo-symmetries/`
Software (Documentation): `https://sat-modulo-symmetries.readthedocs.io/`

Funding The authors acknowledge the support from the Austrian Science Fund (FWF), project 10.55776/P36688, and the Vienna Science and Technology Fund (WWTF), project ICT19-065.

1 Introduction

The quest for a fair division of indivisible goods, a fundamental challenge in economics, computer science, and social choice theory, hinges on developing allocation protocols that balance fairness and efficiency. Envy-freeness is among the most celebrated concepts in this domain, a criterion that ensures no agent prefers another's allocation over their own. However, the indivisibility of goods often precludes the possibility of genuinely envy-free allocations, prompting researchers to explore various relaxations of this ideal. Envy-freeness up to any good (EFX) is a particularly prominent and well-studied relaxation, which requires that no agent should envy another's allocation after the hypothetical removal of any single good from the latter's allocation [2, 4, 5, 6, 7, 8].

Despite its conceptual appeal, the existence of EFX allocations remains unresolved in general settings, marking it as one of the most intriguing open problems in discrete fair division. This has led to a vibrant area of study that seeks to establish the existence of

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 37; pp. 37:1–37:11
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

such allocations and explores the combinatorial structures underlying these problems. A key development in this research trajectory is the exploration of the *rainbow cycle number* $R_f(d)$. This combinatorial constant, which will be defined more formally in Section 2, is the largest integer k such that a directed k-partite graph exists in which every vertex of each block has an incoming edge from any other block, each block has size at most d and the whole graph contains no *rainbow cycle*, i.e., a cycle that runs through every block at most once. Bounds on $R_f(d)$ directly influence the feasibility and construction of approximate EFX allocations with minimal discarded goods. Variants $R_i(d)$ and $R_p(d)$ of the rainbow cycle number (see Section 2 for definitions) have also been studied [3].

Determining $R_f(d)$ and its variants is a natural question of extremal graph theory, in particular to zero-sum combinatorics, independently of its application to EFX-allocations [1, 7]. The first upper bound $R_f(d) \in O(d^4)$ was shown by Chaudhury et al. [7]. This bound was later improved to $d^{2+o(1)}$ [3], and $O(d \cdot \log(d))$ [1, 17].

It is known that for all d, $R_i(d) \leq R_p(d) \leq R_f(d)$, $R_i(d) \geq d$, and it has been conjectured that $R_f(d) = d$; however, this conjecture has only been established so far for $d \leq 3$ in the general setting, and for $d \leq 6$ for $R_i(d)$ [23]. Computing $R_f(d)$ for $d > 3$ is challenging since the number of combinatorial objects to consider explodes.

In this paper, we develop a constraint-based approach within the *SAT modulo Symmetries (SMS)* framework to obtain the upper bounds of $R_f(d)$, which allows us to show $R_f(4) = 4$ and $R_p(5) = 5$. SMS is a recently proposed framework for graph generation modulo isomorphisms under constraints [21]. SMS combines a conflict-driven (CDCL) SAT solver with a custom propagator (since recently via the IPASIR-UP interface [12]) to determine whether a partial assignment encoding a graph can be extended to a canonical, fully defined graph. By employing dynamic symmetry breaking, SMS efficiently searches the vast combinatorial space, avoiding the exploration of isomorphic copies.

We provide an efficient propositional encoding for the rainbow cycle numbers, comparing eager and lazy approaches to ensure the absence of rainbow cycles. The eager approach adds constraints to the encoding at the beginning, while the lazy approach uses a custom propagator to check for the presence of rainbow cycles during the search.

To cope with the huge search space, we introduce the novel technique of *invariant pruning*, which significantly speeds up the computation by strengthening the encoding with graph invariants not known a priori. Invariant pruning exploits the fact that one can observe a significant speedup if a graph invariant is explicitly stated in the encoding and proceeds along a complete decision tree on values for the invariant under consideration (in our concrete setting, it is the maximum degree).

We also employ further symmetry-breaking techniques within the SMS framework to avoid exploring isomorphic copies of graphs in the search space. The applicable permutations for symmetry breaking are described, and the set of all such permutations is represented using ordered partitions. When invariant pruning is applied, the permutations used for symmetry breaking are further restricted to maintain the invariant properties.

Experimental results demonstrate the effectiveness of our approach, with invariant pruning providing a speedup of almost two orders of magnitude in some cases. The time spent in the propagator to ensure the absence of rainbow cycles is only a small fraction of the total computation time. The results also indicate that restricting the functions to permutations significantly decreases the search space compared to the general case.

2 Preliminaries

For a positive integer n, we write $[n] = \{1, 2, \dots, n\}$, $\mathcal{F}_n$ for the set of all functions from $[n]$ to $[n]$, and $\mathcal{S}_n$ for the set containing all permutations over $[n]$. We assume familiarity with fundamental notions of propositional logic [22].

Graphs. We review basic notions from graph theory that are relevant to our discussion. All considered graphs are *directed* and simple (i.e., without parallel edges or self-loops, but pairs of edges in the opposite direction are allowed). A *graph* G consists of a set $V(G)$ of vertices and a set $E(G)$ of edges; we denote the edge from $u \in V(G)$ to $v \in V(G)$ by (u, v). The *order* of a graph G is the number of its vertices, $|V(G)|$. We write $\mathcal{G}_n$ to denote the class of all graphs with $V(G) = [n]$. The *adjacency matrix* of a graph $G \in \mathcal{G}_n$, denoted by A_G, is the $n \times n$ matrix where the element $A_G(v, u)$ at row v and column u is 1 if $(v, u) \in E(G)$ and 0 otherwise. We write $N_G^-(v)$ for the *in-neighborhood* of a vertex v in G, i.e., $N_G^-(v) = \{\, u \in V(G) \mid (u, v) \in E(G) \,\}$. The indegree is given by $d_G^-(v) = |N_G^-(v)|$.

Fixed point cycles and rainbow cycles. A function $f \in \mathcal{F}_d$ has a *fixed point* if $f(x) = x$ for some $x \in [d]$. A *d-labeled graph* is a pair (G, f) such that $G \in \mathcal{G}_n$ and $f_e \in \mathcal{F}_d$ for each $e \in E(G)$; f is an *edge labeling* of G. A *fixed point cycle* of a d-labeled graph (G, f) is a simple cycle $(v_1, v_2, \dots, v_k)$, i.e., $(v_1, v_2), (v_2, v_3), \dots, (v_k, v_1) \in E(G)$ and $v_i \neq v_j$ for $i \neq j$, such that $f_{(v_k, v_1)} \circ \dots \circ f_{(v_2, v_3)} \circ f_{(v_1, v_2)}$ has a fixed point.

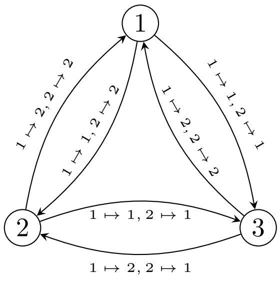
A graph $G \in \mathcal{G}_n$ is *ℓ-partite* if there is a partition $\mathcal{B} = \{B_1, \dots, B_\ell\}$ of $V(G)$ such that no edge of G has both its ends in the same $B \in \mathcal{B}$. We call the sets in $\mathcal{B}$ *blocks.* Let $G \in \mathcal{G}_n$ be an ℓ-partite graph and $\mathcal{B}$ the corresponding partition of $V(G)$. A *rainbow cycle* is a directed cycle that contains at most one vertex of each block in $\mathcal{B}$.

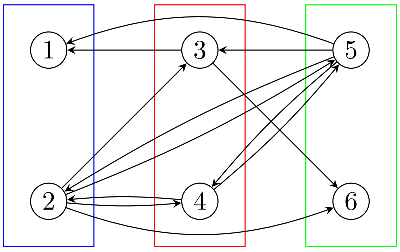
Next, we describe the connection between d-labeled graphs with fixed point cycles and ℓ-partite graphs with rainbow cycles allowing us to focus on the latter. Each d-labeled graph (G, f) with $G \in \mathcal{G}_\ell$ can be transformed into an ℓ-partite graph $H \in \mathcal{G}_{\ell \cdot d}$ such that (G, f) has a fixed point cycle if and only if H has a rainbow cycle. The idea is to introduce for each vertex in $V(G)$ a block with d vertices representing the set $[d]$. The edges in $E(H)$ indicate the mapping, i.e., if $f_{(i,j)}(x) = y$ then there is an edge from the vertex associated with y in the j-th block to the vertex associated with x in the i-th block. More formally

$$E(H) = \{\, ((j-1) \cdot d + f_{(i,j)}(x) \ , \ (i-1) \cdot d + x) \mid (i, j) \in E(G), x \in [d] \,\}$$

with the partition $\mathcal{B}_{d,\ell} := \{\, \{d \cdot (i-1) + 1, \dots, d \cdot i\} \mid i \in [\ell] \,\}$.

▶ **Example 1.** We give an example for $d = 2$ and $\ell = 3$. There, we have $\mathcal{B}_{2,3} = \{\{1, 2\}, \{3, 4\}, \{5, 6\}\}$.


1
$1 \mapsto 2, 2 \mapsto 2$
$1 \mapsto 1, 2 \mapsto 2$
$1 \mapsto 2, 2 \mapsto 2$
$1 \mapsto 1, 2 \mapsto 1$
2
$1 \mapsto 1, 2 \mapsto 1$
3
$1 \mapsto 2, 2 \mapsto 1$



On the left side, we see a d-labeled graph. The graph itself is the complete directed graph on 3 vertices. On the right side, we see the corresponding 3-partite graph. The rectangles indicate the blocks. The cycle $(2, 5)$ forms a rainbow cycle because at most one vertex of each block is in the cycle. This also indicates that $f_{(3,1)} \circ f_{(1,3)}$ has 2 as a fixed point.

The number $R_f(d)$ is the largest integer, such that the edges of the complete directed graph of order $R_f(d)$ can be labeled with $f_e \in \mathcal{F}_d$ without a fixed point cycle. Equivalently, there exists an $R_f(d)$-partite graph such that each block has size d, each vertex has exactly one incoming edge from each other block, and there is no rainbow cycle [3]. $R_f(d)$ is also known as a *rainbow cycle number.* $R_p(d)$ and $R_i(d)$ are defined similarly, except for $R_p(d)$, the edge labelings are restricted to permutations and for $R_i(d)$, the edge labelings are restricted to functions of the form $x \mapsto x + k \mod d$ for $k \in \mathbb{N}$. These numbers are related as follows: $R_i(d) \leq R_p(d) \leq R_f(d)$ [3]. Further, it is known that $R_i(d) \geq d$ [7], given by the edge labeling f:

$$f_{(i,j)}(x) = \begin{cases} x & \text{if } i < j, \\ x + 1 \mod d & \textit{otherwise}. \end{cases}$$

EFX. Let $[n]$ be the set of agents, M the set of indivisible goods, and $v_i : 2^M \to \mathbb{R}_{\geq 0}$ an evaluation function for each $i \in [n]$. An *allocation* $A = (A_1, \ldots, A_n)$ assigns the goods to the agents, i.e., $A_i \subseteq 2^M$ for $i \in [n]$ such that $A_i \cap A_j = \emptyset$ for $i = j$ and $\bigcup_{i \in [n]} A_i = M$. An allocation A is envy-free if $v_i(A_i) \geq v_i(A_j)$ for all $i, j \in [n]$, in other words, no agent prefers the goods of another agent. An *EFX* (envy-freeness up to any good) allocation is an allocation such that $v_i(A_i) \geq v_i(A_j \setminus \{g\})$ for all $i, j \in [n], g \in A_j$. An α-EFX allocation for some $\alpha \in (0, 1]$ is an allocation A such that $v_i(A_i) \geq \alpha \cdot v_i(A_j \setminus \{g\})$ for all $i, j \in [n], g \in A_j$. In a partial-EFX allocation, not all goods need to be assigned, i.e., $\bigcup_{i \in [n]} A_i \subseteq M$.

The relation between the rainbow cycle number and partial α-EFX allocations is given by the following theorem:

▶ **Theorem 2** ([7]). *Let $\varepsilon \in (0, 1/2]$ and let $g(y)$ be the smallest integer d such that $d \cdot R_f(d) \geq y$. Then, there is always a partial $(1-\varepsilon)$-EFX allocation with at most $4n/(\varepsilon \cdot g(2n\varepsilon))$ many unallocated items.*

SAT modulo Symmetries. SAT Modulo Symmetries (SMS) [21] is a recently proposed framework for graph generation modulo isomorphisms under constraints. The original work is restricted to undirected graphs, but later work extends it to directed graphs [20]. SMS combines a conflict-driven (CDCL) SAT solver with a custom propagator to determine whether a partial assignment (which encodes a graph) can be extended to a *canonical* fully defined graph; if not, the solver immediately backtracks. A canonical graph is a distinguished member of its isomorphism class – in SMS, this is typically the graph with the lexicographically smallest adjacency matrix[1]. SMS thus employs what is known as dynamic symmetry breaking [11, 24]. For undirected graphs, prior work has addressed static symmetry breaking [9, 10, 15, 16].

An important concept in the context of SMS are partially defined graphs. A *partially defined graph* is a graph G where $E(G)$ is split into two disjoint sets $D(G)$ and $U(G)$. $D(G)$ contains the *defined* edges, $U(G)$ contains the *undefined* edges. A (fully defined) graph is

[1] We refer to the original work [20] for the precise canonical form.

a partially defined graph G with $U(G) = \emptyset$. Similarly to $\mathcal{G}_n$, let $\mathcal{P}_n$ denote the class of all partially defined graphs G with $V(G) = [n]$. During solving, the presence or absence of some edges is not known, hence a partially defined graph is a suitable way to represent the current solver state. Based on that, the SMS framework additionally enables adding custom propagators to refute partially defined graphs during search.

3 Encoding

In this section, we describe propositional encodings $F_{d,\ell}$ which are satisfiable if and only if $R_f(d) \geq \ell$ for $d, \ell \in \mathbb{N}$. In the satisfiable case, we can extract a directed graph from a model of the formula. Using these formulas, we compute the exact number $R_f(d)$ using several SAT calls. In this work, we are only interested in validating the conjecture $R_f(d) = d$ for small d. For that, showing unsatisfiability of $F_{d,d+1}$ is sufficient.

Given $d, \ell \in \mathbb{N}$, the number of vertices in the searched-for directed graph is $n_{d,\ell} := d \cdot \ell$. The variables $e_{i,j}$ for $i, j \in [n_{d,\ell}], i \neq j$, denote whether the edge (i, j) is present.
First, we ensure that the resulting graph is ℓ-partite with respect to the partition $\mathcal{B}_{d,\ell}$:

$$\mathit{partition}_{d,\ell} = \bigwedge_{B \in \mathcal{B}_{d,\ell}} \bigwedge_{\substack{i,j \in B,\\ i<j}} \neg e_{i,j}.$$

Each vertex in each block has exactly one incoming edge from each other block. In our encoding, the direction of edges is reversed, as it is faster in our experiments in combination with SMS. We encode that each vertex in each block has exactly one outgoing edge to each other block as follows:

$$\mathit{function}_{d,\ell} := \bigwedge_{B_1 \in \mathcal{B}_{d,\ell}} \bigwedge_{i \in B_1} \bigwedge_{\substack{B_2 \in \mathcal{B}_{d,\ell},\\ B_1 \neq B_2}} (\bigvee_{j \in B_2} e_{i,j} \wedge \bigwedge_{\substack{j_1, j_2 \in B_2,\\ j_1 < j_2}} (\neg e_{i,j_1} \wedge \neg e_{i,j_2})).$$

To ensure that no rainbow cycle is present, we distinguish between an eager and a lazy approach. In the first case, we add an encoding to the formula $F_{d,\ell}$ at the beginning. We also refer to this as the static approach. The lazy approach uses a propagator to ensure the absence of a rainbow cycle during search.

3.1 Using a propagator to ensure acyclicity

Given a partially defined graph $G \in \mathcal{P}_{d \cdot \ell}$ and the partition $\mathcal{B}_{d,\ell}$, we want to decide whether a rainbow cycle is present. We use a second SAT solver to check the presence of a rainbow cycle. In the satisfiable case, we use the model to compute a rainbow cycle, which is then excluded by a single clause, ensuring that at least one of the directed edges in the cycle is not present. We design the encoding dependent on the edge variables and use assumptions to fix the graph we want to test. We consider all undefined edges to be absent, because this doesn't introduce additional cycles.

We use variables s_i for $i \in [n_{d,\ell}]$ to indicate whether vertex i is part of the selected cycle. Only one vertex of each block can be selected:

$$\bigwedge_{B \in \mathcal{B}_{d,\ell}} \bigwedge_{i,j \in B, i<j} (\neg s_i \vee \neg s_j).$$

Additionally, we ensure that each selected vertex has exactly one incoming and one outgoing edge to another selected vertex. This is done by using additional variables $se_{i,j}$, which indicate whether the directed edge (i, j) is present and both vertices are selected:

$$se_{i,j} \leftrightarrow (e_{i,j} \wedge s_i \wedge s_j).$$

With these additional variables, we ensure that the indegree and outdegree of the induced subgraph given by the selected vertices is exactly 1. Selecting a minimal rainbow cycle results in a model of the formula. Note that the encoding allows multiple disjoint cycles to be selected, but in this case, we only select one cycle from the model.

We use the IPASIR-UP [12] interface to integrate this propagator.

3.2 Static encoding

We present an encoding exponential in size to ensure that no rainbow cycle is present. One possibility is enumerating all potential cycles and, for each, add a clause to ensure that at least one edge is not present. We use a more compact version using additional variables $p_{u,v,\mathcal{B}'}$ for $B_u, B_v \in \mathcal{B}_{d,\ell}, u \in B_u, v \in B_v, B_u \neq B_v$, and $\mathcal{B}' \subseteq \mathcal{B}_{d,\ell} \setminus \{B_u, B_v\}$. The variables indicate whether there is a colorful path from u to v, i.e., a path only using at most one vertex from each block, only containing vertices in $B_u \cup B_v \cup \bigcup_{B \in \mathcal{B}'} B$.

We have the following constraints using B_v as shorthand for indicating the block containing vertex v:

- $p_{u,v,\emptyset} \leftrightarrow e_{u,v}$: this is the base case.
- $(e_{u,v} \wedge p_{v,w,\mathcal{B}'}) \rightarrow p_{u,w,\mathcal{B}' \cup \{B_v\}}$ if $B_u \notin \mathcal{B}'$: this ensures that if there is a directed edge from u to v and a colorful path from v to w not using the block containing u, then there is a colorful path from u to w additionally using block B_v.
- $p_{u,v,\mathcal{B}'} \rightarrow p_{u,v,\mathcal{B}' \cup \{B\}}$ for $B \in \mathcal{B}' \setminus \{B_u, B_v\}$: adding additional blocks preserves rainbow paths.
- $\neg e_{u,v} \vee \neg p_{v,u,\mathcal{B}' \setminus \{B_u, B_v\}}$: this guaranties that no rainbow cycle is present.

For very small values of d and ℓ, the static approach is still valid, although the number of variables and clauses is exponential.

3.3 Invariant pruning

We introduce the new technique of *invariant pruning* in the context of SMS. A graph *invariant* is a property of a graph invariant under graph isomorphisms. Examples of invariants are the maximum indegree and the maximum outdegree. The goal is to strengthen the formula by computing invariants that are unknown beforehand.

We explain invariant pruning for the rainbow cycle problem regarding the invariant maximum indegree. Let Δ be an upper bound on the maximum indegree, for example, the number of vertices. Instead of solving the formula directly, we fix the first vertex, i.e., the vertex with label 1, to have exactly indegree Δ and all other vertices indegree $\leq \Delta$ in addition to the standard formula $F_{d,\ell}$. If this formula is unsatisfiable, then the maximum indegree is at most $\Delta - 1$, since for any vertex v, there is permutation mapping the vertex v to the first vertex and preserving the partition.

Now, either one solves $F_{d,\ell}$ with the restriction that the maximum indegree is at most $\Delta - 1$ or again tries to refute the case of a vertex having indegree exactly $\Delta - 1$ by fixing the first to have indegree $\Delta - 1$.

In our experiments, we continue pruning the invariant until we reach a certain value at which solving the formula becomes fast. As we see in the experiments in Section 4, this gives a tremendous speedup.

It is intriguing that explicitly demonstrating the impossibility of certain invariants using a SAT solver and then solving the formula proves faster than solely running the solver on the original formula. This phenomenon might stem from the solver finding it easier to identify a "bad" property when it is already associated with a vertex and does not appear arbitrarily.

Table 1 Results for computing $R_f(d)$ and $R_p(d)$ in seconds. For "prop", we additionally provide the fraction of time spent in the cycle-propagator.

	$R_f(d)$		$R_p(d)$	
d	static	prop	static	prop
3	0.03	0.07 (23%)	0.01	0.02 (13%)
4	1603.10	1374.23 (22%)	0.23	1.30 (18%)
5	t.o.	t.o.	1160.23	1236.24 (14%)

3.4 Symmetry breaking

An important part of solving graph search problems with constraint-based methods is symmetry breaking to avoid isomorphic copies in the search space. The vertices can be permuted arbitrarily except the partition $\mathcal{B}_{d,\ell}$ must be preserved. The applicable permutations can be described as follows:

$$\{\, \pi \in \mathcal{S}_{n_{d,\ell}} \mid \forall B \in \mathcal{B}_{d,\ell}\ \exists B' \in \mathcal{B}_{d,\ell} : \pi(B) = B' \,\}.$$

In other words, the image of each block maps to another block.

We break all these symmetries using SMS. Let $\mathcal{O} = [B_1, \dots, B_m]$ be an ordered partition, $l_i = \sum_{j<i} B_i$ and $u_i = \sum_{j\le i} B_i$. We associate with an ordered partition $\mathcal{O}$ the set of permutations

$$\mathrm{Perm}(\mathcal{O}) = \{\, \pi \in \mathcal{S}_{n_{d,\ell}} \mid l_i \le \pi(i) \le u_i \text{ for all } i \in [n_{d,\ell}] \,\}.$$

An ordered partition describes a range to which each vertex can be mapped. For example $\mathrm{Perm}([\{2,3\},\{1\}]) = \{\{2 \mapsto 1, 3 \mapsto 2, 1 \mapsto 3\}, \{2 \mapsto 2, 3 \mapsto 1, 1 \mapsto 3\}\}$.

SMS allows breaking symmetries given by a set of ordered partitions. The set of all applicable permutations for $d, \ell \in \mathbb{N}$ can be represented as follows:

$$\{\, [\{(\pi(1)-1)\cdot d+1, \dots, \pi(1)\cdot d\}, \dots, \{(\pi(\ell)-1)\cdot d+1, \dots, \pi(\ell)\cdot d\}] \mid \pi \in \mathcal{S}_\ell \,\}.$$

The permutation $\pi \in \mathcal{S}_\ell$ describes how the blocks are swapped; the vertices within the block can be permuted arbitrarily.

Note that the presented symmetry breaking is not necessarily compatible with invariant pruning and needs to be adapted slightly. For the example of fixing the indegree of the first vertex, we must further restrict the permutations applicable for symmetry breaking. The first vertex is restricted to map to itself, which also fixes the first block.

4 Experimental Results

In this section, we present the results of our computations. We perform several experiments on a cluster of machines with Intel Xeon E5-2640 v4 processors at 2.40GHz, running Ubuntu 18.04 on Linux 4.15. The source code is available on GitHub[2] and the documentation on Read the Docs[3]. All experiments are executed with a single thread.

[2] https://github.com/markirch/sat-modulo-symmetries

[3] https://sat-modulo-symmetries.readthedocs.io/applications/#computing-small-rainbow-cycle-numbers

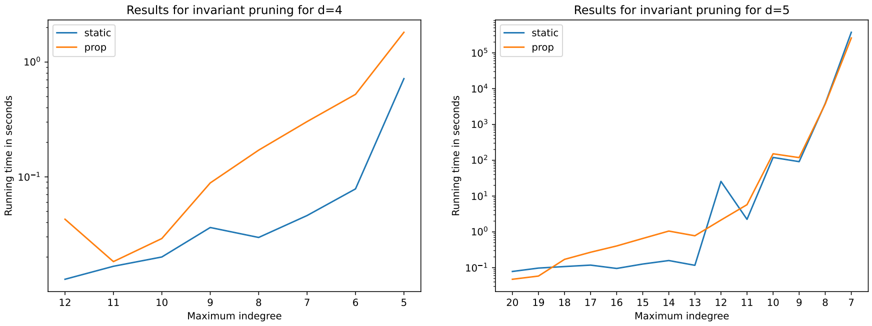


Figure 1 Results for computing $R_f(d)$ using invariant pruning.

Table 1 summarizes the duration in seconds for computing $R_f(d)$ and $R_p(d)$, i.e., showing unsatisfiability for $F_{d,d+1}$. "prop" refers to the version using an external propagator for ensuring the absence of a rainbow cycle (Section 3.1) and "static" to the static encoding (Section 3.2). The results $R_f(4) = R_p(4) = 4$, and $R_p(5) = 5$ were not known before.

As a sanity check, we also enumerate all graphs up to isomorphism encoded by $F_{3,3}$. It results in 64 graphs, two of which encode permutations. For $F_{4,4}$ we stop after enumerating more than 3 million graphs. This also means that there are many extremal graphs. To highlight the importance of symmetry breaking, we run the SAT encoding for $R_p(4)$ without SMS, which doesn't terminate within a day, whilst with SMS, the computation terminates in less than 3 seconds.

We see that the time spent in the propagator is not a bottleneck of the computation. For both $R_f(d)$ and $R_p(d)$, the static and propagator version perform relatively similar for the hardest solved case. The result also shows that restricting the functions only to permutations decreases the search drastically.

Next, we provide results computing $R_f(d)$ using invariant pruning. As invariant, we use the maximum indegree. Note that the outdegree is already known to be $\ell - 1$, i.e., exactly d for the case $\ell = d + 1$. The results are summarized in Figure 1. The figure gives for given fixed maximum indegree the time in seconds to refute this case. We use a logarithmic y-axis. We see that the cases with high indegree are refuted relatively quickly.

For $d = 4$, we can determine within 0.95 seconds that the maximum indegree is strictly smaller than 5 using the static approach and 2.99 seconds using the propagator. The case where the maximum indegree is ≤ 4 is solved in 36.17 and 31.25 seconds, respectively. This means we have a speedup of almost two orders of magnitude using invariant pruning. For $d = 5$, we can restrict the indegree to ≤ 6 within 5 days, but refuting the remaining cases is not feasible yet.

We test a second invariant with invariant pruning, namely the maximum indegree of a vertex with respect to only one block, i.e.,

$$\max_{v \in V(G)} \max_{B \in \mathcal{B}_{d,\ell} \setminus \{B_v\}} |\{\, u \mid (u, v) \in E(G) \,\}|.$$

We call this the *block degree*. If the block degree is determined to be 1, then this is equivalent to the edge labelings being permutations. For $d = 5$, using the previous results that the maximum indegree is ≤ 6, we compute that the block degree is ≤ 2 within 5 days.

Following previous work [18, 19], we verify the reasoning of the solver using DRAT proofs [14] for all our experiments and check the correctness of the symmetry-breaking clauses with a separate script. This is done by first letting the solver run, including propagators, and storing all additionally produced clauses. Next, we feed the original formula, enhanced with the additional clauses, to a SAT solver to produce a DRAT proof, which can then be checked by a DRAT proof checker [25]. Note that it is also possible to use a recently introduced incremental proof format [13] to produce proofs including clauses from propagators without the necessity of running a SAT solver twice.

5 Conclusion

We have investigated the computation of exact rainbow numbers using a constraint-based approach with SAT modulo Symmetries, focusing on the efficacy of invariant pruning to expedite the search process. While our study has primarily examined two invariants, further exploration could uncover additional invariants that might prove beneficial in efficiently handling the case $d = 5$.

A point of improvement for invariant pruning is reusing learned clauses from different invariants by using assumptions for asserting a certain invariant and using one solver incrementally.

An interesting avenue for future research is to evaluate the impact of invariant pruning on various graph generation and enumeration problems. Additionally, we see a potential for the applicability of invariant pruning techniques to solve other highly symmetric formulas not directly tied to graphs.

References

1 Hannaneh Akrami, Bhaskar Ray Chaudhury, Jugal Garg, Kurt Mehlhorn, and Ruta Mehta. EFX allocations: Simplifications and improvements. *CoRR*, abs/2205.07638, 2022. doi:10.48550/arXiv.2205.07638.

2 Georgios Amanatidis, Evangelos Markakis, and Apostolos Ntokos. Multiple birds with one stone: Beating 1/2 for EFX and GMMS via envy cycle elimination. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1790–1797. AAAI Press, 2020. doi:10.1609/AAAI.V34I02.5545.

3 Benjamin Aram Berendsohn, Simona Boyadzhiyska, and László Kozma. Fixed-point cycles and approximate EFX allocations. In *47th International Symposium on Mathematical Foundations of Computer Science, MFCS 2022, August 22-26, 2022, Vienna, Austria*, volume 241 of *LIPIcs*, pages 17:1–17:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.MFCS.2022.17.

4 Ben Berger, Avi Cohen, Michal Feldman, and Amos Fiat. Almost full EFX exists for four agents. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pages 4826–4833. AAAI Press, 2022. doi:10.1609/AAAI.V36I5.20410.

5 Ioannis Caragiannis, David Kurokawa, Hervé Moulin, Ariel D. Procaccia, Nisarg Shah, and Junxing Wang. The unreasonable fairness of maximum Nash welfare. *ACM Trans. Economics and Comput.*, 7(3):12:1–12:32, 2019. doi:10.1145/3355902.

6 Bhaskar Ray Chaudhury, Jugal Garg, and Kurt Mehlhorn. EFX exists for three agents. *J. ACM*, 71(1):4:1–4:27, 2024. doi:10.1145/3616009.

7 Bhaskar Ray Chaudhury, Jugal Garg, Kurt Mehlhorn, Ruta Mehta, and Pranabendu Misra. Improving EFX guarantees through rainbow cycle number. In *EC '21: The 22nd ACM Conference on Economics and Computation, Budapest, Hungary, July 18-23, 2021*, pages 310–311. ACM, 2021. doi:10.1145/3465456.3467605.

8 Bhaskar Ray Chaudhury, Telikepalli Kavitha, Kurt Mehlhorn, and Alkmini Sgouritsa. A little charity guarantees almost envy-freeness. *SIAM J. Comput.*, 50(4):1336–1358, 2021. doi:10.1137/20M1359134.

9 Michael Codish, Graeme Gange, Avraham Itzhakov, and Peter J. Stuckey. Breaking symmetries in graphs: The nauty way. In *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 157–172. Springer Verlag, 2016. doi:10.1007/978-3-319-44953-1_11.

10 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.

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

12 Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. IPASIR-UP: user propagators for CDCL. 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 8:1–8:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.SAT.2023.8.

13 Katalin Fazekas, Florian Pollitt, Mathias Fleury, and Armin Biere. Certifying incremental SAT solving. In *LPAR 2024: Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning, Port Louis, Mauritius, May 26-31, 2024*, volume 100 of *EPiC Series in Computing*, pages 321–340. EasyChair, 2024. doi:10.29007/PDCC.

14 Marijn J. H. Heule. The DRAT format and DRAT-trim checker. *CoRR*, abs/1610.06229, 2016. URL: http://arxiv.org/abs/1610.06229.

15 Marijn J. H. Heule. Optimal symmetry breaking for graph problems. *Math. Comput. Sci.*, 13(4):533–548, 2019. doi:10.1007/S11786-019-00397-5.

16 Avraham Itzhakov and Michael Codish. Breaking symmetries with high dimensional graph invariants and their combination. In *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.

17 Shayan Chashm Jahan, Masoud Seddighin, Seyed Mohammad Seyed Javadi, and Mohammad Sharifi. Rainbow cycle number and EFX allocations: (almost) closing the gap. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*, pages 2572–2580. ijcai.org, 2023. doi:10.24963/IJCAI.2023/286.

18 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. doi:10.24963/IJCAI.2023/216.

19 Markus Kirchweger, Manfred Scheucher, and Stefan Szeider. A SAT attack on Rota's Basis Conjecture. In *Theory and Applications of Satisfiability Testing - SAT 2022 - 25th International Conference, Haifa, Israel, August 2-5, 2022, Proceedings*, 2022. doi:10.4230/LIPIcs.SAT.2022.4.

20 Markus Kirchweger, Manfred Scheucher, and Stefan Szeider. SAT-based generation of planar graphs. In Meena Mahajan and Friedrich Slivovsky, editors, *The 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), July 04-08, 2023, Alghero, Italy*, LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.SAT.2023.14.

21 Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation and enumeration. *ACM Trans. Comput. Log.*, 25(3), 2024. doi:10.1145/3670405.

22 Hans Kleine Büning and Theodor Lettman. *Propositional logic: deduction and algorithms*. Cambridge University Press, Cambridge, 1999.

23 Tamás Mészáros and Raphael Steiner. Zero sum cycles in complete digraphs. *Eur. J. Comb.*, 98:103399, 2021. doi:10.1016/J.EJC.2021.103399.

24 Hakan Metin, Souheib Baarir, Maximilien Colange, and Fabrice Kordon. CDCLSym: introducing effective symmetry breaking in SAT solving. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018*, volume 10805 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2018. doi:10.1007/978-3-319-89960-2_6.

25 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.

Frugal Algorithm Selection

Erdem Kuş
School of Computer Science, University of St Andrews, UK

Özgür Akgün
School of Computer Science, University of St Andrews, UK

Nguyen Dang
School of Computer Science, University of St Andrews, UK

Ian Miguel
School of Computer Science, University of St Andrews, UK

Abstract

When solving decision and optimisation problems, many competing algorithms (model and solver choices) have complementary strengths. Typically, there is no single algorithm that works well for all instances of a problem. Automated algorithm selection has been shown to work very well for choosing a suitable algorithm for a given instance. However, the cost of training can be prohibitively large due to running candidate algorithms on a representative set of training instances. In this work, we explore reducing this cost by choosing a subset of the training instances on which to train. We approach this problem in three ways: using active learning to decide based on prediction uncertainty, augmenting the algorithm predictors with a timeout predictor, and collecting training data using a progressively increasing timeout. We evaluate combinations of these approaches on six datasets from ASLib and present the reduction in labelling cost achieved by each option.

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

Keywords and phrases Algorithm Selection, Active Learning

Digital Object Identifier 10.4230/LIPIcs.CP.2024.38

Category Short Paper

Supplementary Material *Other (Source code, data, experimental results and appendix)*: `https://doi.org/10.5281/zenodo.13294528` [24]

Funding The experiments made use of Cirrus, 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).
Ian Miguel: Ian Miguel is funded by EPSRC grant EP/V027182/1.

1 Introduction

Solving combinatorial optimisation problems is a challenging task, where often multiple approaches compete to offer the most effective solution. In many cases, these problems require large amounts of computational resource. Typically, different algorithms (model and solver choices) are better suited for different problem instances and new potential approaches are continuously developed. Identifying the most suitable algorithm for a particular problem instance has the potential to provide significant efficiency gains.

Machine learning (ML) has emerged as a powerful tool for automating algorithm selection, effectively learning to predict which algorithm is most likely to perform well on a given problem instance. By analysing the features of various problem instances and the corresponding performance of different algorithms, ML models can be trained to recommend the most suitable algorithm for a specific problem [23, 21]. This approach has proven highly effective,

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 38; pp. 38:1–38:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

often leading to significant improvements in computational efficiency and overall problem-solving capability (e.g., [45, 46, 5]). Despite the success of ML-based algorithm selection techniques, there is a notable drawback: the high computational cost associated with running candidate algorithms on a large and representative set of training instances. In order to generate a robust model, it is necessary to evaluate the performance of each algorithm on numerous problem instances. However, this process can be time-consuming and resource-intensive, limiting the scalability and practicality of current algorithm selection methods.

Herein we propose to select instances for training AS models iteratively. This setting is called Active Learning (AL) [8], and is popular for cost-effective training in ML. We focus on AS scenarios that aim at optimising the runtime of the predicted algorithm on a specific instance, a common setting in CP and SAT domains. In these scenarios, each run of an algorithm on a problem instance is limited by a typically large cutoff time. Full information about performance data of timeout cases is expensive to collect but not necessarily useful.

In a typical AL scenario, the *labelling cost* (in our context the cost of running an algorithm to solve a selected instance) is assumed to be uniform. Our situation is further complicated by the fact that, within the cutoff, there is typically significant variance in the cost of running an algorithm. Therefore, we propose two strategies within the active learning setting to reduce labelling cost, thus improving learning efficiency. We evaluate the proposed strategies in combination with two methods for selecting new instances in the active learning setting: an uncertainty-based and a naive random selection method (see Section 3.1 for details). Experimental results on six scenarios from ASLib [6], the standard benchmarking library for algorithm selection, are presented, showing the effectiveness of applying active learning setting to the AS context: in most cases, we can achieve 100% of the predictive power of an AS method that uses all of the training data, while requiring less than 60% of total labelling cost. This is further reduced to as low as 10% of the labelling cost in some scenarios, thanks to a combination of using a timeout predictor and dynamic timeouts.

The primary contributions of our work are:

1. Novel use of timeout predictors and dynamic timeouts for cost-effective algorithm selection.
2. A thorough empirical evaluation across six benchmarks from ASLib, validating the effectiveness of the proposed timeout aware active learning approach.

2 Background

This section provides background for our work, covering three key areas: Automated Algorithm Selection, where the primary challenge is to optimally select an algorithm from a portfolio based on specific instance characteristics and desired performance outcomes; ASLib, a pivotal resource for benchmarking algorithm selection; and Active Learning, a method for organising the training process by strategically choosing informative data points to label.

2.1 Automated Algorithm Selection

The automated algorithm selection (AS) problem can be defined as follows. Given a portfolio of n algorithms $\mathcal{A} = \{a_1, a_2, ..., a_n\}$ with *complementary* strengths, an instance distribution $\mathcal{D}_\mathcal{I}$ where each instance $i \sim \mathcal{D}_\mathcal{I}$ is described as a feature vector $v(i)$, a metric $c(i, a_k)$ that measures the performance of algorithm $a_k \in \mathcal{A}$ on instance i, the AS problem involves building an automated selector $f(v(i))$ to select the best algorithm $a \in \mathcal{A}$ for an instance i. More formally, we want to find f such that $E_{i \sim \mathcal{D}_\mathcal{I}}[c(i, f(v(i))]$ is optimised. There is sometimes an extra cost associated with the extraction of instance features, which must be taken into account in our optimisation objective. In practice, we are often given in advance a set of training instances, their feature vectors, the cost of extracting such features, and the performance of each algorithm in the portfolio on each given instances. This data is used for training a machine learning model to predict the best algorithm for an unseen instance.

The first AS problem was described in 1976 by Rice [35]. Over the last two decades, several AS approaches have been proposed and the applications of such techniques have provided a significant boost in state-of-the-art performance across a wide range of computational areas, such as Constraint Programming [32, 39], propositional satisfiability (SAT) solving [45, 46], AI planning [42, 36, 37], and combinatorial optimisation [22]. A wide range of machine learning-based techniques were adopted in those AS approaches. Some examples include empirical performance models [45], k-nearest neighbours [9], clustering-based methods [20, 5], and cost-sensitive pairwise classification approaches [46]. In addition to building a machine learning model to predict the best performing algorithm, modern AS systems often adopt an extra component called a *pre-solving schedule* [46, 17, 25, 15], a static schedule of algorithms run for a small amount of time before (expensive) feature extraction and algorithm selection are conducted. Another related approach is algorithm scheduling, where instead of selecting a single algorithm for given instance, we build a *schedule* of algorithms [3, 4, 26]. For detailed overviews of AS approaches and their applications, we refer to [21, 23].

In constraint programming, alternative algorithms typically arise from different models for the same problem or from the use of various solvers. Alternative models can be created manually or generated using automated modelling tools such as Conjure [2]. Tools like Savile Row [31, 13] and Minizinc [30] can automatically target multiple alternative solvers.

2.2 The Algorithm Selection Library (ASLib)

The Algorithm Selection Library [6] (ASLib) is a widely-used benchmarking library for automated algorithm selection. It provides a standard format for representing algorithm selection scenarios across a wide range of algorithms and problems, currently consisting of 44 datasets from multiple application domains, including SAT, Quantified Boolean Formula (QBF), Maximum Satisfiability (MAX-SAT), Constraint Satisfaction Problems, Answer Set Programming (ASP), and combinatorial optimisation problems. In these scenarios, the performance metric is either the algorithm running time (for solving a given instance) or the solution quality obtained within a time limit.

The process of data labelling, i.e. obtaining the target output labels for all training data points, is a critical task in machine learning as it is an essential component for building effective models. However, this process is not free of cost, and it can be a time-intensive endeavour, often requiring more resources than training the machine learning model itself. In automated algorithm selection, the cost of collecting performance data can be substantial. As an example, several scenarios in ASLib require more than 100 CPU days to collect the full performance data on the given instance sets (some may require up to 3 CPU years). Our hypothesis is that we can significantly reduce this cost by only collecting partial information about the performance data without sacrificing algorithm selection quality.

2.3 Active Learning for querying informative instances

Active learning is a methodology that maintains machine learning accuracy with fewer labelled data points by allowing incremental labelling of training data [38, 19]. This iterative process comprises training a model on a small set of labelled data, using the model to identify and query the most uncertain unlabelled data points, obtaining labels for these queried points, and then retraining the model with the newly labelled data.

Several methods exist for querying in active learning [47, 44, 10, 33]. Herein, we use pool-based sampling (we maintain a list of candidate training data points that can be queried) and two methods for querying from this pool: uncertainty-based and random instance selection. These are widely regarded as effective methods for active learning, particularly suitable for our context. See Appendix B for more about the behaviour of alternative query methods.

3 Frugal Algorithm Selection

In this section, we first explain the underlying AS model. We then describe how to use active learning to select a subset of instances for labelling during the training (Section 3.1) and the two additional strategies we propose, namely *timeout predictors* (Section 3.2) and *dynamic timeout* (Section 3.3), to reduce labelling cost during the instance selection process.

The AS model used in our experiment follows a pairwise classification approach, as it has been shown to be effective for several AS scenarios (e.g. [46]). The approach is a collection of binary classifiers, each designed to compare a pair of algorithms to determine which one is faster for a given instance. The algorithm with the highest number of votes across all classifiers is chosen as the best option. Following previous work [46], we use a random forest for each classifier. This setup is also advantageous for our goal of minimising resource usage, as it allows selective training of classifiers on specific subsets of instances. The sequence diagram overview of our approach can be found in Appendix A.

Passive learning. an AS model trained using the entire training set. This is the baseline for investigating the effectiveness of our frugal AS methods: we want the frugal AS to achieve the same performance as this passive learning model, while using a significantly less amount of training data.

Frugal methods. We explore three configuration options, each offering two alternatives, to create a range of strategies for frugal algorithm selection. The first configuration option is instance selection, which involves comparing uncertainty-based selection (focusing on potentially informative instances) with random selection. The second configuration option is whether we use timeout predictors and the third configuration option is whether we use dynamic timeouts. In the rest of this section we explain these three configuration options. In Section 4 we empirically evaluate all 8 combinations of these configuration settings.

3.1 Selecting Instances: Uncertainty-based vs random

In our frugal algorithm selection methods, we begin by training all machine learning classifiers on a small number of randomly selected instances. The remaining instances in the training set are kept in a pool of candidates. For each classifier we maintain a separate pool of candidates: this allows us to run an instance on a subset of the algorithms instead of necessarily running it on all algorithm options. This flexibility can be particularly useful when some algorithms tend to timeout very often and hence take up a lot of resources unnecessarily.

At each step of our algorithm, we select N samples from the available unlabelled set of instances. One option for the selection process is employing an uncertainty-based query strategy. Based on an "informativeness" measure, this strategy aims to prioritise the instances that are most likely to yield valuable insights when annotated. We perform uncertainty-based querying by using the predictive model we have partially trained so far. We predict a class for each candidate instance in the pool. The machine learning predictor returns a confidence level in addition to a class prediction. We then query the instances where the predictor is least confident. Uncertainty-based querying is based on the premise that, by focusing on the data points where the model's predictions are least confident, the model is expected to learn from the most uncertain data points. We also allow different machine learning models to make different numbers of queries, based on the confidence levels. For each predictive model we create a table of requested data points and the associated confidence level. We then combine these tables into a single table, sort the table by confidence level and select top N requests. This approach enables predictive models with a high level of uncertainty to query more instances in comparison to those with very low levels of uncertainty.

Uncertainty-based querying can be expensive because it requires feature extraction for all instances in the candidate pool at the start of the procedure. It also only considers informativeness without taking labelling cost into account. To evaluate whether uncertainty-based querying is effective in our work, we compare it with a random query order.

3.2 Timeout Predictor

Instances that time out with a given algorithm are particularly costly in automated algorithm selection. This is partly because if an instance cannot be solved by two algorithms within the timeout, we spend considerable time running these algorithms but gain no new information. Furthermore, when one algorithm solves an instance quickly and another times out, we gain no additional information by allowing the slower algorithm to run to completion. The binary classifiers are provided information only about which algorithm is faster.

We enhance our base machine learning architecture of binary classifiers for all algorithm pairs with dedicated timeout predictors: additional random forest classifiers, one per algorithm, whose task is to predict whether an unseen instance will time out for a specific algorithm. The hypothesis is that training a timeout predictor is a simpler learning task, and this classifier can be trained without requiring additional data. We adjust our voting mechanism to take the timeout predictors into account. If an algorithm is predicted to time out, we exclude it from the options before calculating the votes using the pairwise predictors. An exception is in instances where all algorithms are predicted to time out; in such cases, we do not eliminate any of the options and continue to use pairwise predictors for the entire set. The timeout predictors are also used during the instance selection process: any pair of algorithms and instances predicted to be timeout will be pushed to the end of selection queue, ensuring that we focus on instances that are more likely to provide useful information.

3.3 Dynamic Timeout

The dynamic timeout strategy begins with an initially defined timeout period and incrementally increases it up to a maximum of one hour. After the initial training phase in active learning, the algorithms selected for querying are executed on the selected data within the current time limit. An algorithm that fails to solve the example within this specified time is classified as a timeout for the active time limit. This approach is intended to minimise labelling costs by initially running instances with a short time limit. Hence, resources are not wasted on instances that both algorithms are likely to fail to solve in the early stages. In single timeout cases, where one of the two algorithms can solve the instance, samples can be labelled at a lower cost. The condition for increasing the timeout is determined based on the performance of the model predictions on the validation set. If the model predictions reach a plateau on the validation set, we increase the timeout at the specified rate.

When timeout predictors and dynamic timeouts are used in combination we train the timeout predictors with respect to the particular timeout value at a given moment.

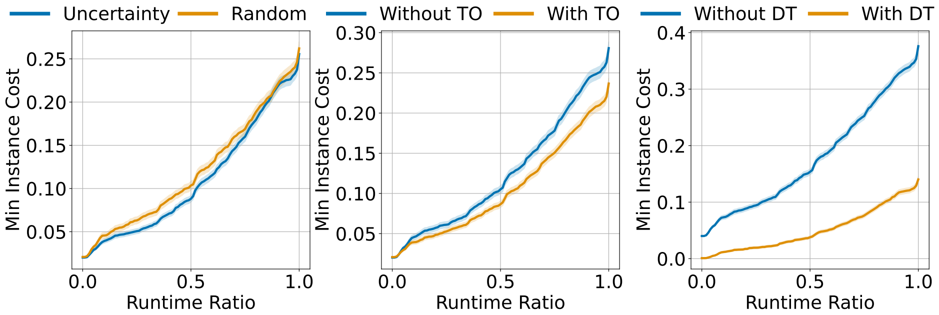


Figure 1 Results aggregated by configuration option. Instance selection (random vs uncertainty-based) does not make a big difference, timeout predictor (TO) improves runtime ratio slightly, dynamic timeout (DT) improves runtime ratio significantly.

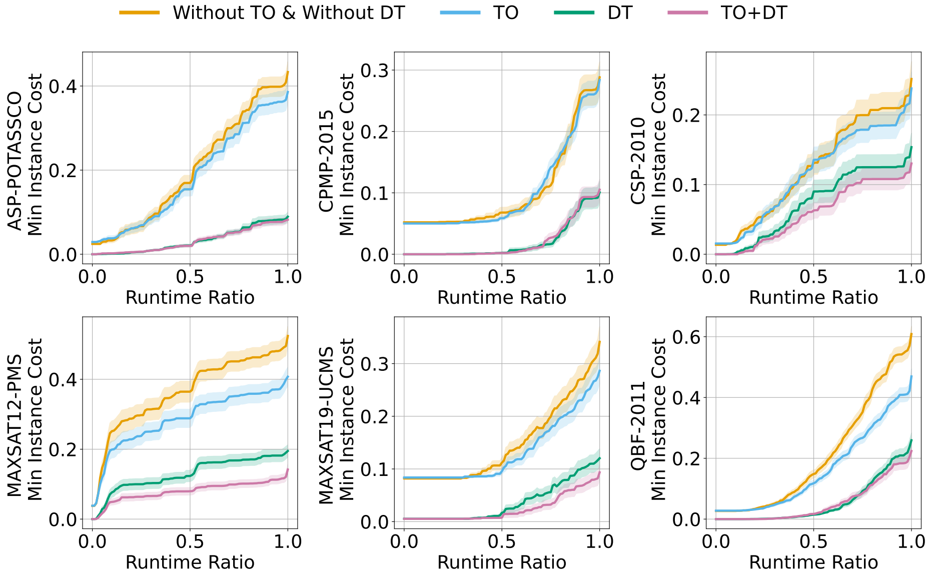


Figure 2 Performance of different timeout configurations (aggregated by instance selection). Notably, the combination of timeout predictor and dynamic timeouts (TO+DT) and the standalone dynamic timeout option exhibit significantly better performance.

4 Experimental Results

We evaluate the performance of all eight configurations of the frugal algorithm selection methods using six datasets from ASLib, chosen for their diverse characteristics. These datasets include various problem-solving domains: one from Answer Set Programming (ASP), two from Constraint Programming (CP), two from propositional satisfiability (SAT), and one from Quantified Boolean Formula (QBF) solving. The datasets vary significantly in complexity and size, with between 2 to 11 algorithm options, 22 to 138 features, and 527 to 2024 instances. Appendix E presents detailed descriptive statistics of the selected datasets.

Experimental setup. To evaluate our methodology we split each dataset into three subsets: training, validation, and test. We allocate 10% of the instances to the test set and perform 10-fold cross-validation to ensure thorough evaluation, running each fold five times with

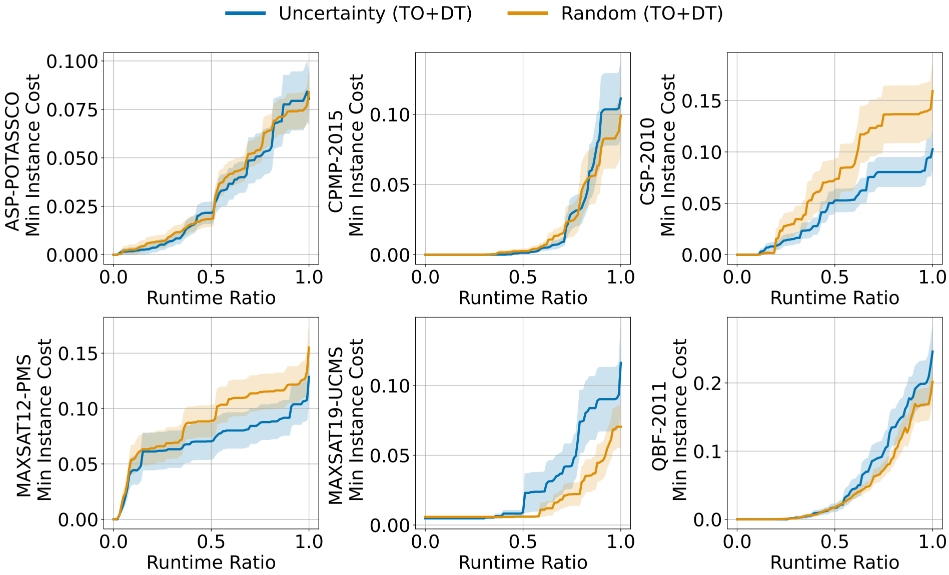


Figure 3 TO+DT options (best in Figure 2) and disaggregating by approach. No clear winner.

different random seeds. An additional 10% of the training set is used as the validation set, used to decide when to increase the timeout (discussed in Section 3.3). At each step, we label 1% of the training data. Appendix D provides full details of the experimental setup.

We present our results in a series of plots designed to compare the configurations at different aggregation levels. All have the same structure. The horizontal axis is the ratio of the performance of passive learning to that of the frugal method, where performance is measured as the total runtime of the predicted algorithms on all test instances. This metric serves as a proxy for predictive performance, indicating how closely the frugal method approaches the benchmark established by passive learning. The vertical axis is the minimum amount of training data (as a ratio of the entire training set) required to achieve the performance indicated on the horizontal axis. The representation highlights the efficiency of the frugal method in terms of data utilisation. We plot the mean and a ribbon showing standard error.

All source code, data, experimental results and the appendix are available at `https://github.com/stacs-cp/CP2024-Frugal`

Key findings. Even the worst configuration of frugal algorithm selection is able to reduce the labelling cost without sacrificing predictive performance relative to passive learning (Figure 2). In several cases training effort is reduced to 10% of the labelling cost of passive learning. It is clear that our frugal approaches are able to reduce the training cost independent of configuration.

Figure 1 presents an overview of the entire set of experiments. The results are aggregated one configuration option at a time, combining results of all configurations that share, for example, Uncertainty as the instance selection method in the first plot. Overall, instance selection strategy does not make a big difference, using timeout predictors (TO) improves performance slightly, and using dynamic timeouts (DT) improves performance significantly.

Since instance selection strategy does not make a big difference and we observe an interesting interaction between TO and DT, we aggregate over instance selection strategy and plot 4 options in Figure 2. Using TO slightly improves performance, while DT improves performance significantly. Moreover the best configurations use TO and DT together.

In Figure 3 we focus on the configurations that include TO+DT and compare the effect of instance selection of the best configuration setting found in Figure 2. We further validate that there is not a clear winner among uncertainty-based instance selection and random instance selection. This finding is consistent across all combinations of strategies, as detailed in Appendix C, which includes raw data plots for all eight configurations.

The observation that uncertainty-based instance selection is not better than random may be unexpected, but selecting instances purely based on informativeness (via prediction uncertainty) does not take the cost of running an instance on a particular algorithm into account. Where there is a uniform cost across all candidates, uncertainty-based selection may perform better than random. In our setting, however, sample cost varies. Hence, we wish to maximise how much information is gained per time spent. Therefore having explicit timeout predictors and a dynamic timeout strategy makes a more significant contribution.

5 Related work

Selecting a representative subset of benchmark instances from a large pool for a reliable and cost-effective comparison of algorithms has been investigated across different domains, including SAT [18, 27, 14], CP [28], combinatorial optimisation [29], evolutionary computation [7], and machine learning [34]. While a majority focuses on selecting a subset of instances in a static setting (i.e., all instances are chosen at once), some recent work has proposed selecting instances iteratively. Matricon et al. [28] present a statistical-based method to incrementally select instances for comparing two solvers. Fuchs et al. [14] propose an active learning-based approach for cost-effective benchmark instance selection. The key difference between the approaches above and our work is that they focus on identifying the algorithm with the best *overall performance* across a given problem instance distribution, while our work focuses on algorithm selection, where the aim is to predict the best *instance-specific* algorithm.

The closest work to ours is Volpato and Song [43], where three commonly-used active learning techniques were evaluated in an AS scenario for SAT[1]. However, they did not consider the significantly varying labelling costs among algorithms and instances, a common characteristic of SAT scenarios. Consequently, the effectiveness of active learning for instance selection was reported based on the percentage of labelled data being saved only.

Although the majority of active learning techniques assume uniform labelling cost, there is work on non-uniform labelling cost settings (e.g., [38, 40, 41]). A common approach is to predict the labelling cost and to strive for a balance between informativeness and the predicted cost of a new data point. We adopt a similar technique in our work, where timeout predictors are used for identifying costly (unlabelled) data points. It can be considered a “softened” version of the cost estimation approach, as predicting the precise runtime of algorithms in the domain of combinatorial optimisation is often difficult: most AS techniques focus on learning the ranking among algorithms instead of trying to predict runtime directly [16].

Effective algorithm selection relies on the availability of a representative set of instances for a problem. When such instances are not available, automated methods can generate discriminating instances [1, 11]. These methods aim to find instances that one algorithm can solve quickly while another algorithm struggles, and vice versa.

[1] These three methods result in the same set of selected instances when the prediction model is binary classification as in our work – see Appendix B.

6 Conclusion

We have proposed and evaluated several approaches to *frugal* algorithm selection, an active learning method that reduces labelling cost by using only a subset of the training data, together with a dynamic timeout strategy that uses incomplete information about the performance of algorithms in the portfolio. Our results confirm the utility of the proposed approach and our analysis offers insights into the contribution of each of its components. Interestingly, the standard active learning data selection technique contributes little to performance, while our proposed dynamic timeout mechanism results in significant savings.

In future, we plan to incorporate enhancement techniques in AS into the proposed active learning framework, including the use of a pre-solving schedule [45] and cost-sensitive pairwise classification AS models [46, 25]. Other important avenues include investigating the impact of hyper-parameter tuning in the active learning setting, and developing an early-stopping mechanism to terminate the learning process once diminishing returns are observed.

References

1 Özgür Akgün, Nguyen Dang, Ian Miguel, András Z Salamon, Patrick Spracklen, and Christopher Stone. Discriminating instance generation from abstract specifications: A case study with cp and mip. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21–24, 2020, Proceedings 17*, pages 41–51. Springer, 2020.

2 Özgür Akgün, Alan M Frisch, Ian P Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artificial Intelligence*, 310:103751, 2022.

3 Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Sunny: a lazy portfolio approach for constraint solving. *Theory and Practice of Logic Programming*, 14(4-5):509–524, 2014.

4 Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Sunny-cp: a sequential cp portfolio solver. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1861–1867, 2015.

5 Carlos Ansótegui, Joel Gabas, Yuri Malitsky, and Meinolf Sellmann. Maxsat by improved instance-specific algorithm configuration. *Artificial Intelligence*, 235:26–39, 2016.

6 Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. Aslib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, 2016. doi:10.1016/j.artint.2016.04.003.

7 Gjorgjina Cenikj, Ryan Dieter Lang, Andries Petrus Engelbrecht, Carola Doerr, Peter Korošec, and Tome Eftimov. Selector: selecting a representative benchmark suite for reproducible statistical comparison. In *Proceedings of The Genetic and Evolutionary Computation Conference*, pages 620–629, 2022.

8 David Cohn. *Active Learning*, pages 10–14. Springer US, Boston, MA, 2010. doi:10.1007/978-0-387-30164-8_6.

9 Marco Collautti, Yuri Malitsky, Deepak Mehta, and Barry O'Sullivan. Snnap: Solver-based nearest neighbor for algorithm portfolios. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part III 13*, pages 435–450. Springer, 2013.

10 Nguyen Viet Cuong, Wee Sun Lee, Nan Ye, Kian Ming A. Chai, and Hai Leong Chieu. Active learning for probabilistic hypotheses using the maximum gibbs error criterion. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, pages 1457–1465, Red Hook, NY, USA, 2013. Curran Associates Inc.

11 Nguyen Dang, Özgür Akgün, Joan Espasa, Ian Miguel, and Peter Nightingale. A framework for generating informative benchmark instances. *arXiv preprint arXiv:2205.14753*, 2022.

12 Tivadar Danka and Péter Horváth. modal: A modular active learning framework for python. *CoRR*, abs/1805.00979, 2018. arXiv:1805.00979.

13 Ewan Davidson, Özgür Akgün, Joan Espasa, and Peter Nightingale. Effective encodings of constraint programming models to smt. In *Principles and Practice of Constraint Programming: 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7–11, 2020, Proceedings 26*, pages 143–159. Springer, 2020.

14 Tobias Fuchs, Jakob Bach, and Markus Iser. Active learning for sat solver benchmarking. In *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 I*, pages 407–425. Springer, 2023.

15 François Gonard, Marc Schoenauer, and Michèle Sebag. Algorithm selector and prescheduler in the icon challenge. *Bioinspired heuristics for optimization*, pages 203–219, 2019.

16 Jonas Hanselle, Alexander Tornede, Marcel Wever, and Eyke Hüllermeier. Hybrid ranking and regression for algorithm selection. In *German Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 59–72. Springer, 2020.

17 Holger Hoos, Marius Lindauer, and Torsten Schaub. claspfolio 2: Advances in algorithm selection for answer set programming. *Theory and Practice of Logic Programming*, 14(4-5):569–585, 2014.

18 Holger H Hoos, Benjamin Kaufmann, Torsten Schaub, and Marius Schneider. Robust benchmark set selection for boolean constraint solvers. In *Learning and Intelligent Optimization: 7th International Conference, LION 7, Catania, Italy, January 7-11, 2013, Revised Selected Papers 7*, pages 138–152. Springer, 2013.

19 Sheng-Jun Huang, Rong Jin, and Zhi-Hua Zhou. Active learning by querying informative and representative examples. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(10):1936–1949, 2014. doi:10.1109/TPAMI.2014.2307881.

20 Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac–instance-specific algorithm configuration. In *ECAI 2010*, pages 751–756. IOS Press, 2010.

21 Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary computation*, 27(1):3–45, 2019.

22 Pascal Kerschke, Lars Kotthoff, Jakob Bossek, Holger H Hoos, and Heike Trautmann. Leveraging tsp solver complementarity through machine learning. *Evolutionary computation*, 26(4):597–620, 2018.

23 Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. *Data mining and constraint programming: Foundations of a cross-disciplinary approach*, pages 149–190, 2016.

24 Erdem Kuş. stacs-cp/CP2024-Frugal. Other, version 1.1. (visited on 2024-08-20). URL: https://doi.org/10.5281/zenodo.13294528.

25 Marius Lindauer, Holger H Hoos, Frank Hutter, and Torsten Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015.

26 Tong Liu, Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. sunny-as2: Enhancing sunny for algorithm selection. *Journal of Artificial Intelligence Research*, 72:329–376, 2021.

27 Norbert Manthey and Sibylle Möhle. Better evaluations by analyzing benchmark structure. *Proc. PoS*, 2016.

28 Théo Matricon, Marie Anastacio, Nathanaël Fijalkow, Laurent Simon, and Holger H Hoos. Statistical comparison of algorithm performance through instance selection. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

29 Mustafa Mısır. Benchmark set reduction for cheap empirical algorithmic studies. In *2021 IEEE Congress on Evolutionary Computation (CEC)*, pages 871–877. IEEE, 2021.

30 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.

31 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.

32 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.

33 Mijung Park and Jonathan W. Pillow. Bayesian active learning with localized priors for fast receptive field characterization. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pages 2348–2356, Red Hook, NY, USA, 2012. Curran Associates Inc.

34 João Luiz Junho Pereira, Kate Smith-Miles, Mario Andrés Muñoz, and Ana Carolina Lorena. Optimal selection of benchmarking datasets for unbiased machine learning algorithm evaluation. *Data Mining and Knowledge Discovery*, 38(2):461–500, 2024.

35 John R Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976.

36 Mattia Rizzini, Chris Fawcett, Mauro Vallati, Alfonso E Gerevini, and Holger H Hoos. Portfolio methods for optimal planning: an empirical analysis. In *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 494–501. IEEE, 2015.

37 Mattia Rizzini, Chris Fawcett, Mauro Vallati, Alfonso E Gerevini, and Holger H Hoos. Static and dynamic portfolio methods for optimal planning: An empirical analysis. *International Journal on Artificial Intelligence Tools*, 26(01):1760006, 2017.

38 Burr Settles. Active learning literature survey, 2009. URL: `https://api.semanticscholar.org/CorpusID:324600`.

39 Patrick Spracklen, Nguyen Dang, Özgür Akgün, and Ian Miguel. Automated streamliner portfolios for constraint satisfaction problems. *Artificial Intelligence*, 319:103915, 2023.

40 Katrin Tomanek and Udo Hahn. A comparison of models for cost-sensitive active learning. In *Coling 2010: Posters*, pages 1247–1255, 2010.

41 Yu-Lin Tsou and Hsuan-Tien Lin. Annotation cost-sensitive active learning by tree sampling. *Machine Learning*, 108(5):785–807, 2019.

42 Mauro Vallati, Lukáš Chrpa, and Diane Kitchin. Asap: an automatic algorithm selection approach for planning. *International Journal on Artificial Intelligence Tools*, 23(06):1460032, 2014.

43 Riccardo Volpato and Guangyan Song. Active learning to optimise time-expensive algorithm selection. *arXiv preprint arXiv:1909.03261*, 2019.

44 Liantao Wang, Xuelei Hu, Bo Yuan, and Jianfeng Lu. Active learning via query synthesis and nearest neighbour search. *Neurocomputing*, 147:426–434, 2015. Advances in Self-Organizing Maps Subtitle of the special issue: Selected Papers from the Workshop on Self-Organizing Maps 2012 (WSOM 2012). `doi:10.1016/j.neucom.2014.06.042`.

45 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.

46 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.

47 Jingbo Zhu, Huizhen Wang, Benjamin K. Tsou, and Matthew Ma. Active learning with sampling by uncertainty and density for data annotations. *IEEE Transactions on Audio, Speech, and Language Processing*, 18(6):1323–1331, 2010. `doi:10.1109/TASL.2009.2033421`.

A Appendix A: Sequence Diagram Overview

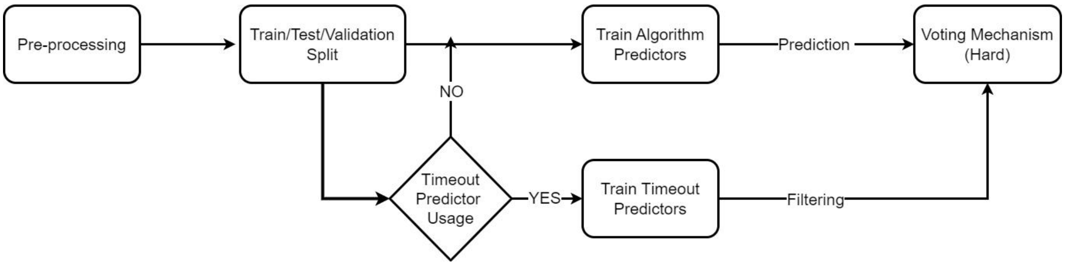


Figure 4 The diagram depicts the process of passive learning, which includes preprocessing, splitting the dataset, and training multiple binary Random Forest (RF) algorithm classifiers and timeout classifiers, followed by a hard voting mechanism to finalize predictions. Timeout predictors are used for filtering algorithm predictors in the voting mechanism. All models are included in the voting mechanism where the timeout predictor configuration is not applied.

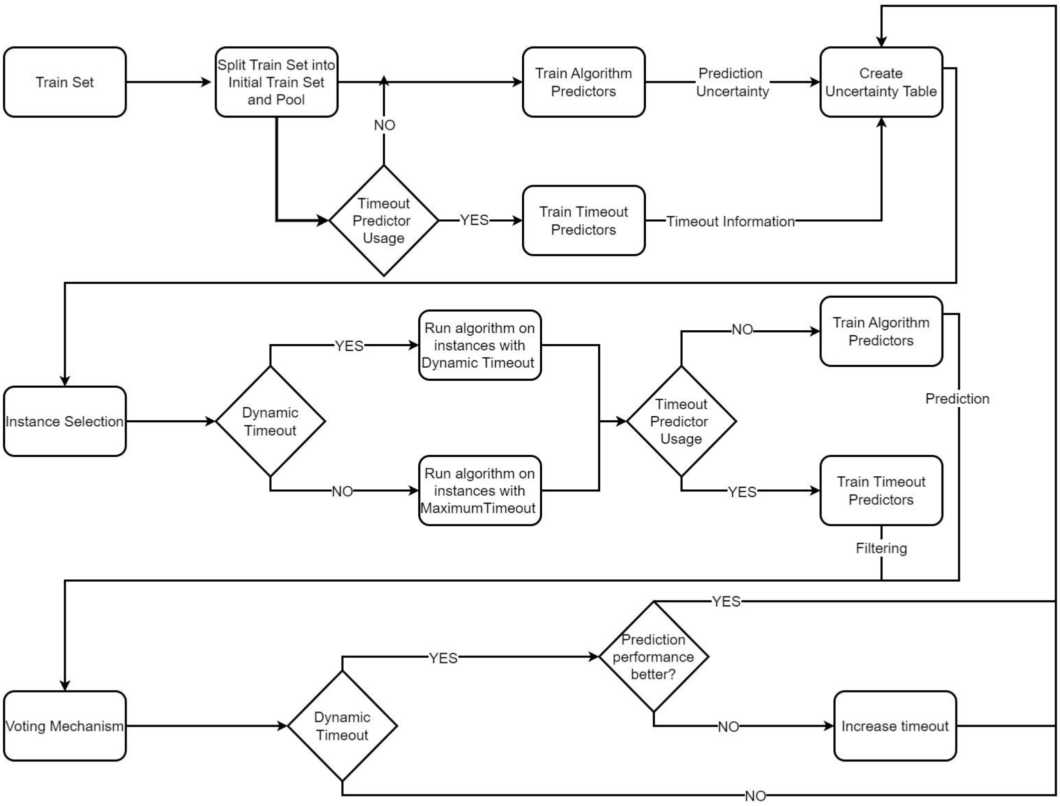


Figure 5 The diagram illustrates the steps in the proposed approach. It starts with splitting the training set into an initial training set and a pool for each model. Multiple binary Random Forest (RF) algorithm and timeout classifiers are then trained. The instance selection process involves identifying the most uncertain data points across all models and excluding instances predicted to time out by the timeout predictors. A dynamic timeout is applied during the labeling process, which is increased when there is no performance enhancement on the validation set. After labeling, the iterative process begins again, continuously refining the models.

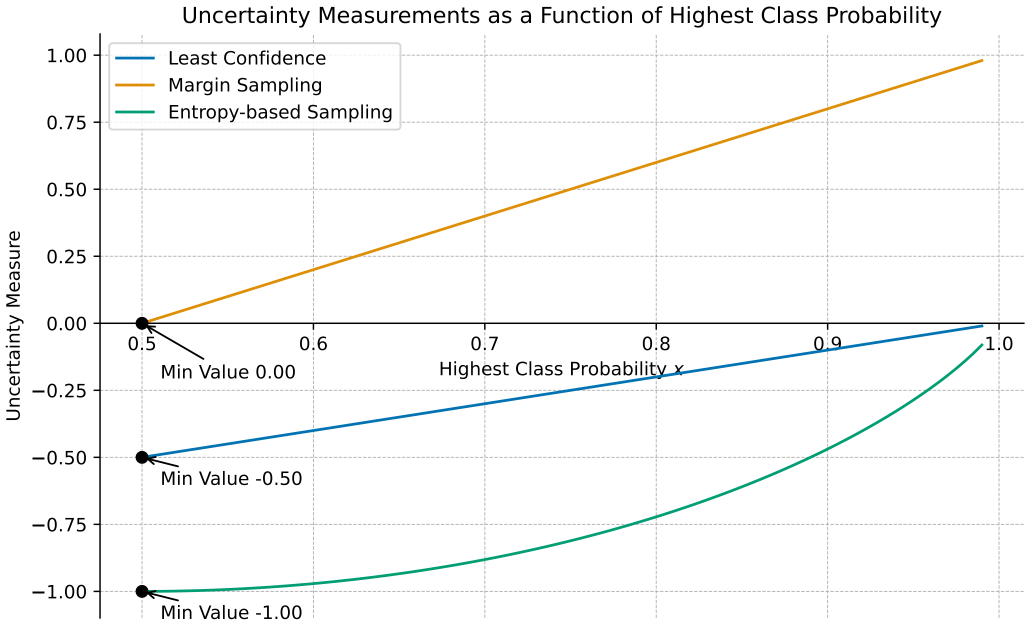


Figure 6 Uncertainty measurements as a function of the highest class probability. The red curve represents the Least Confidence uncertainty (LC) calculated as $LC = x - 1$, the green curve denotes Margin Sampling (MS) using the formula $MS = x - (1 - x)$, and the blue curve illustrates the Entropy-based method ($H(x) = -[x \log_2(x) + (1-x)\log_2(1-x)]$). Critical minimum values for each method are marked with black circles and annotated to emphasise the points where the uncertainty function is minimised.

B Appendix B: Analysis of Uncertainty Measurement Behaviours in Active Learning for Binary Classification

There are three main approaches for uncertainty sampling in active learning. However, in a binary classification setting (which is what we use) these approaches perform identically to each other. We explain the different approaches here. Figure 6 shows the behaviour of these uncertainty sampling methods graphically.

We implement "Least Confidence" in our approach.

- *Least Confidence*: for a given input x and an output label $\hat{y}$, we can measure the posterior probability $P(\hat{y}|x;\theta)$ of observing $\hat{y}$ given x via the current model (parameterised by θ). The Least Confidence method selects data points x^* with the smallest maximum posterior probability across all labels:

$$x^* = \operatorname*{argmin}_x \max_{\hat{y}} P(\hat{y}|x;\theta) \tag{1}$$

- *Margin-based*: this approach takes the two highest posterior probability values for each input data point x and calculates their difference. The smaller the difference, the less certain the model is about its prediction and vice versa. More formally, let $\hat{y_1}$ and $\hat{y_2}$ the output labels with the highest and second-highest posterior probabilities for a given input x, respectively, the queried points x^* are chosen as:

$$x^* = \operatorname*{argmin}_x P(\hat{y_1}|x;\theta) - P(\hat{y_2}|x;\theta) \tag{2}$$

- *Entropy-based*: this approach takes into account the posterior probability values across *all* output classes. The idea is to select the data points x^* where there is a high entropy among the predicted output labels:

$$x^* = \underset{x}{\operatorname{argmax}} - \sum_i P(\hat{y}|x;\theta) \log P(\hat{y}|x;\theta) \tag{3}$$

C Appendix C: Performance of 8 individual configurations

Figure 7 illustrates a side-by-side comparison of the following eight active learning strategies in binary classification without aggregation across configurations:

- Uncertainty Sampling without Timeout Predictor & without Dynamic Timeout (NO TO & NO DT)
- Uncertainty Sampling with Timeout Predictor (TO)
- Uncertainty Sampling with Dynamic Timeout (DT)
- Uncertainty Sampling with Timeout Predictor and Dynamic Timeout (TO+DT)
- Random Sampling without Timeout Predictor & without Dynamic Timeout (NO TO & NO DT)
- Random Sampling with Timeout Predictor (TO)
- Random Sampling with Dynamic Timeout (DT)
- Random Sampling with Timeout Predictor and Dynamic Timeout (TO+DT)

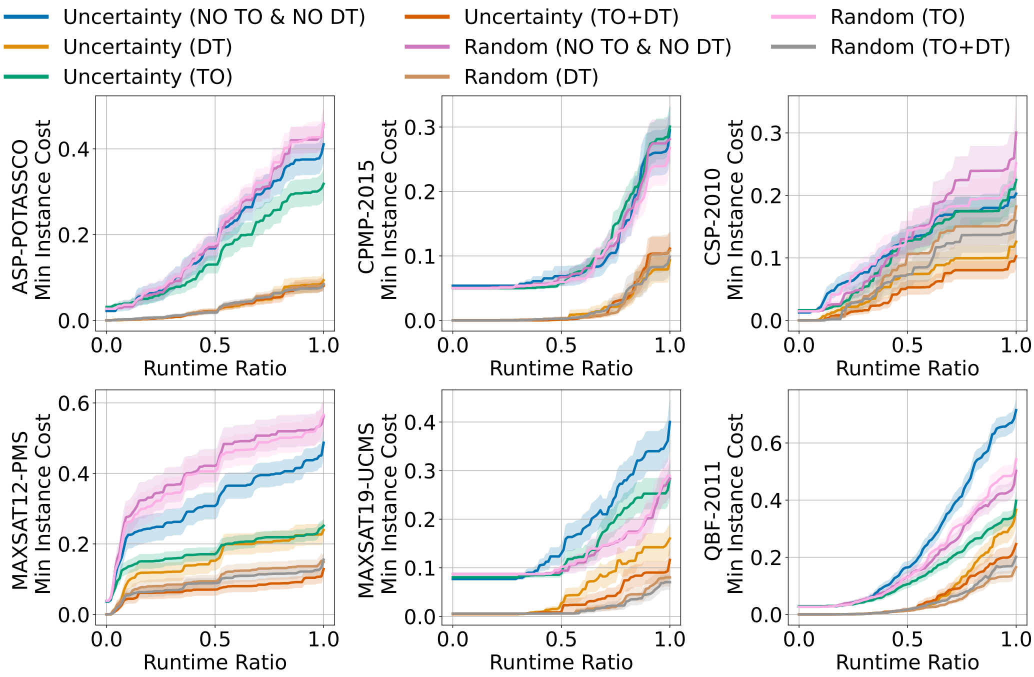


Figure 7 Comparison of performance across eight configurations as described in the paper. Each configuration was normalised according to the passive learning prediction performance ratio.

D Appendix D: Experimental Setup

This study used a Random Forest classifier configured with 100 estimators and the Gini impurity measure to determine the best splits. Each tree is limited to using up to the square root of the number of features, and the depth of the decision trees is practically unlimited (with a maximum depth set to 2^{31}). Nodes require at least two samples before splitting, and bootstrapping is enabled for sampling data when building each decision tree. These settings were determined through experimentation in the passive learning setup and were consistently used throughout the study.

We also addressed missing data by removing features where more than 20% of the instances had missing values and applied a median imputer to fill the remaining gaps.

We employed a cross-validation approach with 10 splits to validate the robustness of our study. To ensure reproducibility, we used 5 distinct seeds (7, 42, 99, 123, 12345) across our experiments, ensuring consistent generalization across multiple runs.

To determine when to increase the timeout in configurations where dynamic timeout is used, 10% of the training set was allocated as the validation set. Throughout the experiments, timeout values were scaled by a factor of 10, following the PAR10 measure.

Additional Parameters and Configurations.

Timeout Predictor Usage: This parameter determines whether the timeout predictor is used on the system.

Timeout Limit: Sets the initial time for the dynamic timeout. We used an initial timeout of 100 seconds when employing dynamic timeout, and a fixed timeout of 3600 seconds when not using dynamic timeout.

Timeout Increase Rate: Adjusts the dynamic timeout when there is no improvement in prediction performance on the validation set. We set this rate to increase by 100 seconds when no improvement was observed.

Initial Train Size: Determines the size of the initial training set for uncertainty selection. The initial training set was created by randomly selecting 20 data points from the overall training set.

Query Size: Refers to the percentage of the dataset queried in each iteration. We set this to 1%, meaning 1% of the total pool of candidates was queried in each iteration of our experiments.

For active learning, we utilized the modAL framework [12], which facilitated the implementation of uncertainty sampling and other active learning strategies in our experiments.

E Appendix E: Description Table of Selected Datasets

Table 1 shows key information about the datasets used in this study. It includes the time it took for the algorithms to run, the Virtual Best Solver(VBS) representing the best algorithm for each problem, and the Single Best Solver (SBS) as the best overall algorithm. While VBS is the hypothetical best, SBS serves as a benchmark for comparison against other algorithms.

Table 1 Descriptive statistics of selected datasets. Times rounded to the nearest whole number.

Dataset	Instances	Algorithms	Features	Total Time	VBS	SBS
ASP-POTASSCO	1294	11	138	2,085h	8h	112h
CPMP-2015	527	4	22	682h	33h	134h
CSP-2010	2024	2	86	435h	49h	82h
MAXSAT12-PMS	876	6	37	1,472h	8h	85h
MAXSAT19-UCMS	572	7	54	545h	20h	52h
QBF-2011	1368	5	46	352h	28h	300h

F Appendix F: Timeout (TO) Configuration Impact on Passive Learning

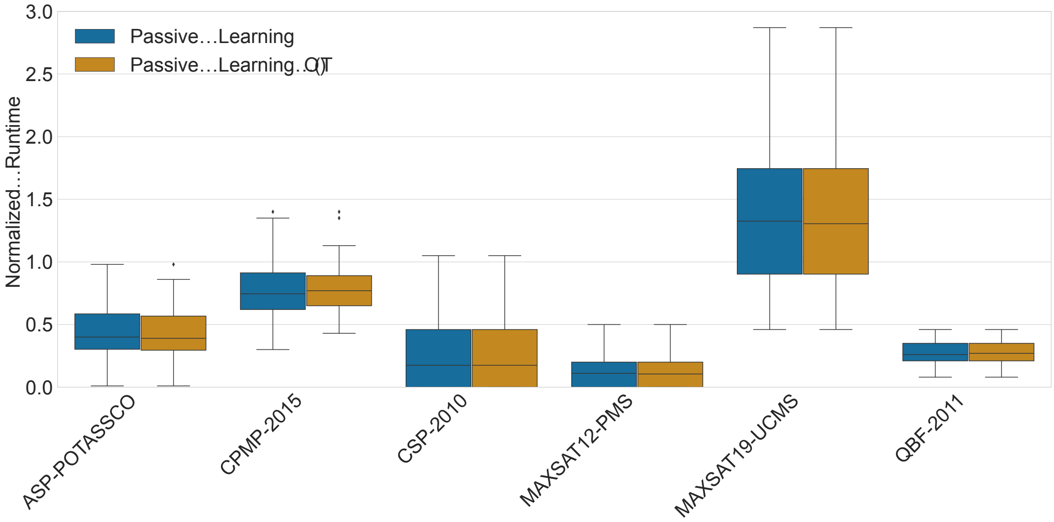


Figure 8 Comparison of Timeout (TO) Configuration Impact on Passive Learning: The graph illustrates that implementing the TO configuration in passive learning on the test set does not significantly enhance performance, yet importantly, it does not compromise prediction accuracy either.

An Investigation of Generic Approaches to Large Neighbourhood Search

Filipe Souza
Insight SFI Research Centre for Data Analytics, National University of Ireland Galway, Ireland
SFI Centre for Research Training in Artificial Intelligence, Cork, Ireland
School of Computer Science & IT, University College Cork, Ireland

Diarmuid Grimes
Munster Technological University, Cork, Ireland
SFI Centre for Research Training in Artificial Intelligence, Cork, Ireland

Barry O'Sullivan
Insight SFI Research Centre for Data Analytics, National University of Ireland Galway, Ireland
SFI Centre for Research Training in Artificial Intelligence, Cork, Ireland
School of Computer Science & IT, University College Cork, Ireland

Abstract

A bottleneck in the more wide-spread use of approaches such as Large Neighborhood Search is the need for domain-specific knowledge. To this end, a number of generic LNS methods have previously been proposed that automate the selection of variables in the neighborhood with the aim of reducing the expertise requirement. Recently a new generic approach, Improved Variable-Relationship Guided LNS (iVRG), was proposed that showed promising initial results. This method combines static information regarding problem structure and dynamic information from search performance in its neighborhood selection.

In this work, we first show the generalisability of the approach by comparing it on two widely studied problems, car sequencing and steel mill slab, where it outperformed existing generic approaches. We then provide a detailed examination of iVRG, investigating its key components (static/dynamic information, the use of a Tournament Selection operator) to assess their individual impact and provide insight into iVRGs overall behavior.

2012 ACM Subject Classification Computing methodologies → Heuristic function construction

Keywords and phrases Combinatorial Optimization, Metaheuristics, Large Neighborhood Search (LNS), Machine Reassignment Problem, Car Sequencing Problem, Steel Mill Slab Problem

Digital Object Identifier 10.4230/LIPIcs.CP.2024.39

Category Short Paper

Supplementary Material *Software (Source Code)*: `https://github.com/filipesouzait/iVRG-LNS`

Funding Supported by SFI Centre for Research Training in Artificial Intelligence under Grant No. 18/CRT/6223 and SFI under Grant No. 12/RC/2289-P2, co-funded under the European Regional Development Fund.

1 Introduction

Large neighborhood search [15] is a metaheuristic approach that works by iteratively improving an initial solution through optimising subsets of variables. Each iteration involves the selection and relaxation (unassignment) of a neighborhood of variables (*destroy* phase). The problem is then optimised (*repair* phase) with the neighborhood restricted to only those variables that can be searched over, all other variables are fixed to their values in the current solution. This has led to significant advances in terms of problem size that can be handled by such optimisation algorithms, enabling them to explore vast solution spaces efficiently.

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 39; pp. 39:1–39:10
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

There are a number of components that must be chosen for a given implementation of the basic LNS framework: the method to select the variables in a given neighborhood, the size of the neighborhood, and the solution approach (typically an exact method such as CP/MIP). However, the more general applicability of LNS to a wide range of problem domains remains a challenge as it often requires domain-specific insights to effectively design the neighborhood selection mechanism. To address this limitation a number of domain-independent neighbourhood selection approaches have been proposed down through the years [12, 7, 6, 10]. These approaches involve strategies that can adapt to various problem domains without relying on domain-specific knowledge.

Despite the importance of domain-independent neighbourhood selection approaches recent research in the area of generic neighborhood selection operators has been relatively scarce with much of the focus on *adaptive LNS* approaches [13, 5, 20]. These approaches use a portfolio of heuristics with the system adapting weights based on search performance to decide on the probability of selecting a given heuristic in the next iteration. The quality of these approaches is sensitive to the diversification of heuristics included in the portfolio. More recently there has also been a focus on machine learning approaches for learning neighborhood selection, albeit primarily restricted to mixed integer programming (MIP) [10, 22].

In this paper, we present an analysis of a recently proposed domain-independent neighbourhood selection approach, Improved Variable-Relationship Guided LNS (iVRG) [19]. This approach showed promising results, albeit only evaluated on Google's Machine Reassignment Problem (MRP). We delve into its key components, including the use of structural relationships, search state information, and a tournament selection mechanism.

To provide a robust comparative analysis, we empirically compare iVRG with two state-of-the-art generic neighbourhood selection approaches: Propagation Guided LNS (PG-LNS) [12] and Cost Impact Guided LNS [6]. The former was originally evaluated using the Car Sequencing Problem (CSP), while the latter was tested on the Steel Mill Slab Problem (SMSP). We extend our previous analysis [19] to these two problem domains to assess the generalisability of iVRG. We then investigate the impact of the different components of the iVRG approach, the use of structural information, the use of search state information, and finally the use of tournament selection within the heuristic.

2 Related Work

2.1 Propagation Guided Large Neighbourhood Search

Perron et al. [12] proposed using propagation information to identify strongly connected neighbourhoods. To generate this information the basic PG-LNS approach starts from all variables unassigned and initially chooses a variable at random. It then repeatedly chooses a variable to assign until a predefined neighborhood size is reached. The variable to assign is chosen randomly from the top ten variables ranked according to their domain reduction after assigning the previously selected variable. If this list is empty, selection reverts to a random choice from the remaining relaxed variables.

A complementary approach, Reverse PG-LNS, was also proposed. This starts from all variables being assigned and repeatedly unassigns a variable until the desired neighborhood size is achieved. The variable to unassign is chosen randomly from a list of the ten variables with the highest *closeness* score accumulated from the previous selected variables. This score is based on the impacts computed in PG-LNS.

2.2 Cost-Impact Guided LNS

Cost-Impact Guided Large Neighbourhood Search (CIG-LNS) [6] selects variables for relaxation based on their impact on the cost (objective function). This cost impact is determined by observing the variations in the lower bound that occur when each variable is assigned a value. These variations are captured through *dives*, where a dive is the re-application of the current solution in a rearranged order. An additional parameter α is incorporated into each variable's score to control the level of diversification. The parameter value is in the range $[0, 1]$, with 0 equating to pure random selection and 1 being that the cost alone is used.

3 Improved Variable-Relationship Guided LNS

Improved Variable-Relationship Guided LNS [19] is a neighborhood selection operator that uses the structural relationships between variables to guide search towards *connected* neighborhoods. It combines this static information with dynamic information from search to prioritize variables with a higher likelihood of enhancing the solution, and uses tournament selection to only consider a subset of variables for selecting the next variable for the neighborhood. The latter serves to both increase diversification, and to reduce computational effort (since heuristic information is only computed for the subset of variables).

3.1 Structural Relationship

The structural relationship is incorporated in neighborhood selection by considering only variables linked to the most recently relaxed variable. Specifically, the probability of selecting a variable j subsequent to the relaxation of variable i is determined by the following formula:

$$\frac{1}{|C_i|} * \sum_{c \in C_{i,j}} \frac{1}{|V_c|}$$

where $C_{i,j}$ denotes the set of constraints involving both variables j and i, while $|V_c|$ is the arity of constraint c, and $|C_i|$ is the number of constraints involving variable i. The logic of using constraint arity is that the relevance of the relationship between any two variables due to a constraint diminishes as the number of variables sharing that constraint increases. At its most extreme, when a constraint includes all variables, it fails to offer meaningful insight into the strengths of the relationships between those variables. Note since these are static values, this need only be computed once at the start of an LNS run on an instance.

However, there are also problems where variables without such a direct relationship can still contribute greatly to solution improvement when selected together (e.g. swapping their bins in a bin packing problem). For these problems, forcing all neighborhood variables to be connected may impede performance. Therefore, iVRG uses the combination of structural relationship and search state information (SSI) for half of the neighborhood variables, and otherwise only uses SSI.

3.2 Search State Information

There are a number of different forms of search state information (SSI) that could be used, e.g. variables that are in conflict during search, variables whose selection resulted in large improvements during search, etc. In previous work on iVRG [19], two aspects of SSI were combined. The first is the *Variable Cost*, that centered on the principle that the most impactful neighborhood selections involve high-cost variables. This heuristic measures a variable's contribution to the overall cost by computing the impact on the objective function of its removal from the current solution. The second SSI component is focused on diversification, maintaining a count of the number of iterations the variable was relaxed in. The heuristic considers the *Variable Cost* divided by the frequency of selection across previous iterations.

3.3 Tournament Selection

Tournament selection within the iVRG framework selects the variable with best SSI value from a subset of variables. This subset can be either selected based on their structural relationship to the previously relaxed variable (using the formula defined in Section 3.1), or randomly selected. As previously mentioned, variables that don't share a constraint can also contribute to solution improvement when selected together. Therefore, half the tournaments have the subset of variables selected randomly for the tournament, and the other half use the relationship to the previously relaxed variable.

4 Problem Description

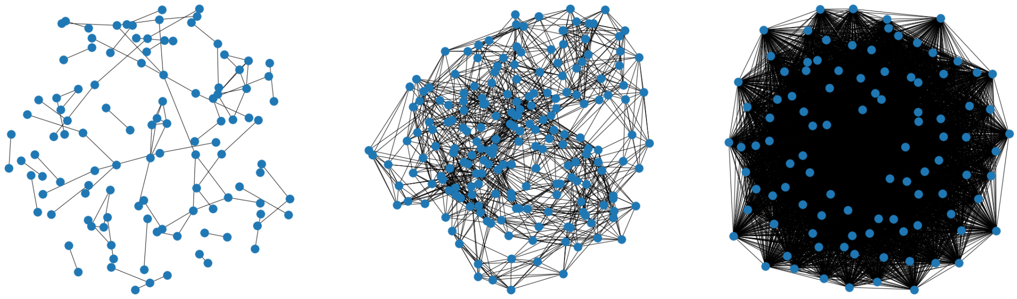
The Steel Mill Slab Problem (SMSP) [8, 14] involves assigning steel orders to slabs while minimising slab wastage. Each slab has a maximum weight capacity, and orders have specific weight and colour. The challenge is to efficiently group orders onto slabs in a way that we have a maximum of two colours per slab while minimising the total waste.

The Car Sequencing Problem (CSP) was initially formulated as a satisfaction problem [1, 16], involving the assignment of a sequence of cars to a limited number of slots in a production line where each bay installs specific options and has a limited capacity. Many optimisation variants with different objective functions have since been proposed (e.g. [11, 3]). In this work we consider the variant proposed by Souza *et al.* [18] that defines the objective function in terms of minimising the total number of options in the cars not placed in the production line.

The Machine Reassignment Problem (MRP) considered here was proposed by Google for the 2012 Roadef Challenge.[1] The problem requires optimising the reallocation of a set of processes to a set of machines with the goal of minimising a multi-objective function: weighted sum of objective function components. The problem further involves a number of constraints related to capacity, conflicting subsets of processes, as well as spread and dependency amongst groups of processes. Due to its complexity and specificity, the MRP has been the focus of many works [2].

The problem type instances were chosen based on what each approach (PG-LNS, CiG-LNS and iVRG) had used for their evaluation in their respective publications. They also represent a diverse set of combinatorial optimisation challenges with varying constraints and complexities (the low propagation of the SMSP, the higher constrained solutions of the CSP, and the density and size of the MRP instances). This variety ensures a comprehensive evaluation of each heuristic's adaptability and scalability. Figure 1 shows the structure of a sample instance for each of the problem types. We note that the SMSP is very sparse, with disconnected components. The MRP on the other hand is extremely dense, while the CSP is somewhat in the middle of the other two problem types in terms of its density.

[1] https://www.roadef.org/challenge/2012/files/problem_definition_v1.pdf



(a) SMSP. **(b)** CSP. **(c)** MRP.

Figure 1 Problem structure (variable relationship) of sample instances: (a) Steel Mill Slab Problem, (b) Car Sequencing Problem, and the (c) Machine Reassignment Problem.

5 Experimental Setup

The experiments were run on a machine running Ubuntu 18.04.3 LTS (GNU/Linux 4.15.0-70-generic) with 16 cores and 32Gb of RAM. All runs had a runtime cutoff of 2 minutes per instance for the 2 smaller problem types: the Car Sequencing Problem and the Steel Mill Slab Problem. While for the more complex Machine Reassignment Problem, the cutoff was 5 minutes per instance. Those runtimes were chosen to be consistent with the values used in previous literature. The MRP was introduced in the ROADEF 2012 with a runtime cutoff of 5 minutes and most of the papers that addressed this problem focused on this setting. The runtime used in the original PG-LNS paper [12] for their CSP experiments was 2 minutes. In the paper that introduced CIG-LNS [6] the authors used an iteration-limit of 1000 for their SMSP experiments rather than a time-limit, therefore we used 2 minutes for the SMSP in these experiments, which generated more than 2000 iterations on average for each approach.

Gecode[2] was the CP solver used for subproblem optimisation in the two smallest instance types. However as it could not handle the large MRP, a dedicated solver was implemented for this problem type. Furthermore, as the approaches have stochastic components, the presented results are the average of 10 runs with different seeds. Table 1 presents the parameter configurations that were used to run the experiments. The failure threshold / first solution improvement were the stopping conditions used by the CP solver per iteration.

Table 1 Configurations parameters for the benchmark experiments.

Parameter	Value
Runtime	120 seconds (SMSP, CSP); 300 seconds (MRP)
Neighbourhood Size	10 variables
Tournament Size	10 variables
Failure Threshold	200

It should be noted that for PG-LNS we implemented the best configuration presented in [12], that iterates through the following three neighborhood operators: Propagation Guided; Reverse Propagation Guided; and purely random selection. However, we defined the

[2] https://www.gecode.org

neighbourhood size based on the number of relaxed variables instead of the search space size, in order to compare all approaches on the same neighbourhood size. Similarly for CIG-LNS, we implemented the optimal configuration identified in [6]. In particular the value of α was set to 0.5, and a dive was performed after every 10 unsuccessful LNS iterations and subsequent to each improve solution.

5.1 Benchmark Instances

For the MRP, the three sets of instances from the 2012 Roadef challenge [9] were used (A,B and X), where each set has 10 instances. The A instance set is composed of smaller instances with a maximum of 1k variables and domains of size 100. The other two sets of instances are more complex and larger, with up to 50k variables and domains of up to 5k. For the CSP, three sets of hard instances proposed by Caroline Gagne and available on the CSPLib [16] were used. There are 10 instances in each set and the total number of cars per instance (equal to the number of variables, and to domain size -1) is 200, 300 and 400, respectively, for the three sets. Finally, for the Steel Mill Slab Problem, the same set of instance used in [6] was used. This set involves 80 instances, each with 111 variables and domain size 111, divided into four groups based on the number of slab capacities available (2,3,4 and 5).

5.2 Evaluation Metrics

The metrics used for our evaluation on the three different problem types are a normalized scoring metric to assess the quality of the solution, and a similarity metric to assess the diversity of the neighborhoods. Note all three problem types are minimisation.

The score metric is that used in ROADEF'12 [9]. The metric measures the distance the solution found is from the best known solution, but also considers how much improvement was made from the initial solution. It is calculated $\boldsymbol{((Cost - BK)/initialCost) * 100}$, where $Cost$ denotes the cost of the solution found, BK is the best known cost for the problem instance [21, 2, 17, 4], and $initialCost$ is the cost of the initial solution. For fair comparison, note that the same initial solution was used for all algorithms for a given run on a given instance. This score was chosen because the solution costs have a huge difference in the scale, e.g. the best known solution for MRP instance A2_1 is 151 and for A2_2 is 720671511.

The similarity metric represents the average percentage of intersection (common variables in neighborhoods) observed across the first 1,000 iterations of the LNS, as illustrated in the equation below. In the case where an approach did not manage to perform 1,000 iterations in the defined runtime (which occurred only for PG-LNS on the largest MRP problem sets), the metric represents the average percentage of intersection in all iterations.

$$Similarity = \frac{1}{\binom{1000}{2}} \sum_{i=0}^{999} \sum_{j=i+1}^{999} \frac{|N[i] \cap N[j]|}{|N[i]|}.$$

6 Results

6.1 Comparison of different domain-independent neighborhood operators for LNS

We first investigated if the performance previously shown on MRP [19] would hold on the SMSP and the CSP. We compared the three generic neighborhood selection heuristics and a pure random approach (*Rand*), in terms of average score, similarity and number of iterations across problem sets. The results are given in Table 2, showing that iVRG consistently outperformed the other approaches on all problem sets. CIG-LNS and PG-LNS had varying performance across the problem types, with the latter outperforming the former on the CSP but the opposite the case for the other two problem types. Indeed in our experiments the random approach outperformed both on the SMSP, albeit by a small amount.

Somewhat surprisingly we find that iVRG has the highest similarity in the two smallest problem types. However, we note that it was not significantly higher than the other approaches. Indeed all approaches contain a strong random component, but a very low similarity is not necessarily a good characteristic, for example to try to maintain a balance between diversification and intensification of search.

An analysis of the number of iterations performed by the different approaches also provides insights. Firstly, PG-LNS and CIG-LNS suffered from scalability. In particular for the MRP, the number of iterations performed on the instance sets B and X, which had instances with up to 50k variables and domains of maximum size 5k, were more than 80% less than on the A set. In comparison, Rand and iVRG had less than 50% drop in iterations. The cost of computing the cost-impact / propagation-impact was prohibitive for these large instances.

The iterations on the other two problem sets demonstrate the two extreme cases in neighborhood selection that we wish to avoid, both of which are heavily influenced by a lack of relationship amongst variables in the selected neighborhood. In one case there is no search space to search, as the only consistent values the relaxed variables can take are the values they take in the current solution. Therefore, there are many iterations performed, with few nodes explored. This can be seen for the CSP, with Rand and CIG-LNS performing nearly an order of magnitude more iterations than PG-LNS and iVRG. Indeed analysis of the average nodes explored per iteration shows that Rand and CIG-LNS explored around 10 nodes per iteration on average, compared to 200-300 for PG-LNS and iVRG. We also note that both the latter two approaches were significantly better in terms of score on this problem type compared to the former two.

The opposite case is where there is too large a search space, due to lack of propagation when values assigned. This is the case for the SMSP where the variables are not strongly connected, as shown in Figure 1a. Here, iVRG performed many more iterations than the others, with average nodes per iteration of 100 for iVRG compared to approximately three times as many by the other approaches. Given the failure threshold of 200, this shows that the solver was able to improve most neighborhoods chosen by iVRG, but rarely was able to improve any of the neighborhoods chosen by the other methods for the SMSP.

Table 2 Comparison of iVRG, PG-LNS, CIG-LNS, and Random Selection on the three problem types: Steel Mill Slab (SMSP), Car Sequencing (CSP), and Machine Reassignment (MRP).

Problem	**Group**	**Score**				**Similarity**				**#Iterations** (x1000)			
		Rand	PG	CIG	iVRG	Rand	PG	CIG	iVRG	Rand	PG	CIG	iVRG
SMSP	2	10.23%	10.24%	10.79%	**5.51%**	**9.01%**	10.10%	9.62%	10.24%	1.7	1.7	1.6	**4.4**
	3	10.80%	11.81%	11.59%	**5.17%**	**9.01%**	10.06%	9.76%	10.26%	2.0	1.9	1.9	**4.3**
	4	5.51%	5.97%	5.68%	**2.81%**	**9.01%**	10.11%	9.85%	10.12%	2.3	2.3	2.3	**7.6**
	5	4.78%	5.57%	4.58%	**2.13%**	**9.01%**	10.17%	10.05%	10.08%	2.7	2.7	2.6	**7.6**
	Overall	7.83%	8.40%	8.16%	**3.91%**	**9.01%**	10.11%	9.82%	10.17%	2.2	2.1	2.1	**6.0**
CSP	200	9.71%	5.36%	8.97%	**4.43%**	**5.00%**	5.01%	5.26%	5.51%	78.2	12.7	**131.7**	18.4
	300	10.36%	5.46%	9.57%	**3.83%**	**3.33%**	3.34%	3.44%	3.64%	52.4	9.1	**87.3**	12.8
	400	11.58%	5.67%	10.11%	**3.86%**	2.50%	**2.50%**	2.55%	2.72%	32.9	6.3	**55.5**	9.1
	Overall	10.55%	5.50%	9.55%	**4.04%**	**3.61%**	3.61%	3.75%	3.95%	54.5	9.3	**91.5**	13.4
MRP	A	3.69%	5.25%	3.17%	**2.33%**	**4.56%**	5.11%	8.80%	5.06%	87.3	7.6	**98.7**	70.1
	B	0.31%	0.94%	0.36%	**0.26%**	**0.26%**	0.26%	3.14%	0.35%	**52.2**	0.8	13.6	44.0
	X	0.46%	0.62%	0.41%	**0.34%**	0.29%	**0.25%**	3.69%	0.38%	**53.9**	0.8	15.7	34.9
	Overall	1.49%	2.27%	1.31%	**0.98%**	**1.70%**	1.87%	5.21%	1.93%	**64.5**	3.0	42.7	49.7

6.2 Analysis of iVRG Components

Given the performance of iVRG in the previous section, we next investigated the contribution of each of its three main components to this performance. In particular we compared iVRG against the following iVRG versions: without tournament selection (*NonT*); without using search information (*NonS*); and finally without using the structural relationship (*NonR*). For NonS, variables were chosen randomly, albeit maintaining the structural relationship with previously selected variables in the neighborhood.

The results are presented in Table 3, and show that overall the use of structural relationship had the biggest impact. *NonR* consistently had the biggest drop in performance compared to iVRG. Tournament selection was the next most important, with significant drops in performance on the CSP and the MRP. Somewhat surprisingly, the results of *NonS* demonstrate that the search state information has only a relatively small impact on iVRG performance across all three problem sets, compared to the other two components.

Interestingly, we find a stronger correlation between similarity and performance here than in the previous table. NonR has consistently higher similarity than iVRG. This indicates a more diverse neighborhood selection in iVRG due to the effect of variable relationships, which restrict the variables available to relax to a different group (based on relationships of the first chosen variable) every iteration.

NonT also had higher similarity for all except the sparse SMSP problems, with these problems being the only ones where NonT had comparable performance with iVRG. Of course, tournament selection has less impact on smaller instances. Note that under the current settings of 10 for both neighborhood size and tournament size, 100 variables are considered in each LNS iteration when choosing the neighborhood, and SMSP instances had only 111 variables.

The average iterations metric reveals the importance of the relationship information in neighborhood selection in iVRG. Without this, iVRG suffers from the same pitfalls as discussed in Table 2. The sparseness of the SMSP results in searching a large search space, even with just 10 variables in the neighborhood, due to the lack of propagation. On the other hand, it performs a much greater number of iterations on the two other problem sets, but many of these required little or no search as again disconnected neighborhood variables in much more constrained instances resulting in most variables being assigned without search.

Table 3 iVRG compared to iVRG without: Tournament Selection (NonT), search state information (NonS), and variable-relationship (NonR).

Problem	Group	Score				Similarity				#Iterations (x1000)			
		iVRG	NonT	NonS	NonR	iVRG	NonT	NonS	NonR	iVRG	NonT	NonS	NonR
SMSP	2	**5.51%**	6.23%	6.24%	10.38%	10.24%	9.92%	**9.82%**	22.42%	**4.4**	4.3	4.1	1.8
	3	5.17%	**4.93%**	5.34%	14.41%	10.26%	10.03%	**9.96%**	21.57%	4.3	**4.5**	4.4	2.1
	4	2.81%	**2.79%**	2.82%	13.68%	10.12%	9.88%	**9.85%**	21.27%	7.6	7.9	**8.0**	2.1
	5	2.13%	**2.06%**	2.28%	17.38%	10.08%	9.90%	**9.84%**	20.97%	7.6	**8.0**	7.4	1.9
	Overall	**3.91%**	4.00%	4.17%	13.96%	10.17%	9.93%	**9.87%**	21.56%	6.0	**6.2**	6.0	2.0
CSP	200	**4.43%**	10.47%	4.50%	9.59%	5.51%	12.57%	**5.02%**	6.03%	18.4	14.1	14.2	**86.8**
	300	**3.83%**	11.11%	4.28%	10.71%	3.64%	10.92%	**3.35%**	4.02%	12.8	9.7	9.9	**56.1**
	400	**3.86%**	9.57%	3.87%	11.32%	2.72%	7.21%	**2.50%**	3.06%	9.1	6.1	7.4	**34.9**
	Overall	**4.04%**	10.38%	4.22%	10.54%	3.95%	10.23%	**3.62%**	4.37%	13.4	10.0	10.5	**59.3**
MRP	A	**2.33%**	6.04%	2.66%	5.85%	5.06%	23.83%	**4.75%**	10.90%	70.1	65.4	63.1	**91.2**
	B	**0.26%**	0.74%	0.29%	0.39%	0.35%	20.10%	**0.28%**	0.61%	44.0	8.5	44.9	**57.0**
	X	**0.34%**	0.90%	0.37%	0.46%	0.38%	20.83%	**0.31%**	0.67%	34.9	7.2	40.8	**89.2**
	Overall	**0.98%**	2.56%	1.11%	2.23%	1.93%	21.59%	**1.78%**	4.06%	49.7	27.0	49.6	**79.1**

The advantages of tournament selection is particularly prominent on the large MRP B and X instances, where the cost of computing heuristic values and sorting across 1000s of variables is prohibitive. Here, in the worst case, the SSI must be calculated for 50,000 variables, compared to 100 variables with tournament selection. This demonstrates that the benefit of tournament selection is not only in terms of diversification, as shown by NonT similarity scores, but also in terms of scalability.

7 Conclusion

We have demonstrated the generalisability of the recently proposed iVRG. It was shown to significantly outperform similar approaches (PG-LNS and CIG-LNS) on three challenging problem types of different characteristics. Additional analysis revealed that it is able to avoid two of the main pitfalls of neighborhood selection, unlike the comparison approaches. The main iVRG component from this respect was the use of static information regarding the relationship of variables, in order to select neighborhoods with a high degree of connectivity.

An ablation study of the main components was then performed, considering the use by iVRG of: static structural information; dynamic search state information; and finally the use of tournament selection within the neighborhood operator. The results revealed that the problem structure was the most important aspect, followed closely by tournament selection. The former contributes to the algorithm's ability to adapt for distinct problem characteristics, effective even for problem such as Steel Mill Slab with a low level of relationship between decision variables. Tournament selection was shown to be effective in increasing diversification, and scalability. While, the search state information had relatively low impact in iVRG compared to the other two components, it still yielded consistent improvements over random selection. Indeed these two components are generic and could possibly result in improved performance if plugged into other LNS approaches.

References

1 Christian Artigues, Emmanuel Hebrard, Valentin Mayer-Eichberger, Mohamed Siala, and Toby Walsh. Sat and hybrid models of the car sequencing problem. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 268–283. Springer, 2014.

2 Dario Canales, Nicolas Rojas-Morales, and Maria-Cristina Riff. A survey and a classification of recent approaches to solve the google machine reassignment problem. *IEEE Access*, 8:88815–88829, 2020.

3 Jens Gottlieb, Markus Puchta, and Christine Solnon. A study of greedy, local search, and ant colony optimization approaches for car sequencing problems. In *Applications of Evolutionary Computing: EvoWorkshops 2003*, pages 246–257. Springer, 2003.

4 Stefan Heinz, Thomas Schlechte, Rüdiger Stephan, and Michael Winkler. Solving steel mill slab design problems. *Constraints*, 17(1):39–50, 2012.

5 Philippe Laborie and Daniel Godard. Self-adapting large neighborhood search: Application to single-mode scheduling problems. *Proceedings MISTA-07, Paris*, 8, 2007.

6 Michele Lombardi and Pierre Schaus. Cost impact guided lns. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 293–300. Springer, 2014.

7 Jean-Baptiste Mairy, Yves Deville, and Pascal Van Hentenryck. Reinforced adaptive large neighborhood search. In *The Seventeenth International Conference on Principles and Practice of Constraint Programming (CP 2011)*, page 55. Springer Berlin/Heidelberg, Germany, 2011.

8 Ian Miguel. CSPLib problem 038: Steel mill slab design. `http://www.csplib.org/Problems/prob038`.

9 H Murat Afsar, Christian Artigues, Eric Bourreau, and Safia Kedad-Sidhoum. Machine reassignment problem: the roadef/euro challenge 2012, 2016.

10 Vinod Nair, Mohammad Alizadeh, et al. Neural large neighborhood search. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

11 Laurent Perron and Paul Shaw. Combining forces to solve the car sequencing problem. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 225–239. Springer, 2004.

12 Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided large neighborhood search. In *International Conference on Principles and Practice of Constraint Programming*, pages 468–481. Springer, 2004.

13 Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science*, 40(4):455–472, 2006.

14 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*, 16:125–147, 2011.

15 Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming*, pages 417–431. Springer, 1998.

16 Barbara Smith. CSPLib problem 001: Car sequencing. `http://www.csplib.org/Problems/prob001`.

17 Filipe Souza, Diarmuid Grimes, and Barry O'Sullivan. A large neighborhood search approach for the data centre machine reassignment problem. In *Irish Conference on Artificial Intelligence and Cognitive Science*, pages 397–408. Springer, 2022.

18 Filipe Souza, Diarmuid Grimes, and Barry O'Sullivan. Variable-relationship guided lns for the car sequencing problem. In *Irish Conference on Artificial Intelligence and Cognitive Science*, pages 437–449. Springer, 2022.

19 Filipe Souza, Diarmuid Grimes, and Barry O'Sullivan. Improved variable-relationship guided lns for the data centre machine reassignment problem. In *Irish Conference on Artificial Intelligence and Cognitive Science*, page to appear. Springer, 2023.

20 Charles Thomas and Pierre Schaus. Revisiting the self-adaptive large neighborhood search. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 557–566. Springer, 2018.

21 Ayad Turky. *Bi-level hyper-heuristic approaches for combinatorial optimisation problems.* PhD thesis, RMIT University, 2019.

22 Yaoxin Wu, Wen Song, Zhiguang Cao, and Jie Zhang. Learning large neighborhood search policy for integer programming. *Advances in Neural Information Processing Systems*, 34:30075–30087, 2021.

Encoding the Hamiltonian Cycle Problem into SAT Based on Vertex Elimination

Neng-Fa Zhou
CUNY Brooklyn College and the Graduate Center, NY, USA

Abstract

This paper presents a SAT encoding, called vertex elimination encoding (VEE), for the Hamiltonian Cycle Problem (HCP). The encoding maps a Hamiltonian cycle in the reduced graph after vertex elimination to a Hamiltonian cycle in the original graph. While VEE is not competitive for large dense graphs due to its large encoding sizes, it can be utilized to reduce graphs when they are sparse. This paper compares VEE with the distance encoding, and shows that the hybridization of these two encodings is effective for the benchmarks. For the knight's tour problem, in particular, the hybrid encoding solves some middle-sized instances that were beyond the reach for previous eager SAT encodings.

2012 ACM Subject Classification Computing methodologies → Knowledge representation and reasoning; Computing methodologies → Planning and scheduling; Hardware → Theorem proving and SAT solving

Keywords and phrases Graph constraints, the Hamiltonian cycle problem, SAT encoding, Vertex elimination, Graph synthesis

Digital Object Identifier 10.4230/LIPIcs.CP.2024.40

Category Short Paper

Acknowledgements The author would like to thank Ruiwei Wang, Roland Yap Hock Chuan, and the anonymous reviewers for helpful comments.

1 Introduction

The Hamiltonian Cycle Problem (HCP), a classic problem in graph theory, seeks to find a cycle in a given directed graph that includes each and every vertex exactly once. With the availability of fast Satisfiability (SAT) solvers, various studies have been conducted to encode the HCP into SAT to leverage the solving power[2, 3, 6, 9, 10, 11]. The HCP can be encoded with degree and no-sub-cycle constraints. Most of the reported SAT encodings focus on how to translate no-sub-cycle constraints into SAT. Despite some successes, current SAT-based solvers are not yet competitive on many problems with other solvers, such as Constraint Programming (CP) and Answer Set Programming (ASP) solvers.

Vertex elimination [8] is a problem-solving technique used in various graph problems. The idea of the technique is to simplify complex graphs by removing certain vertices from a graph while preserving important properties. Recently, an encoding based on vertex elimination has been proposed for encoding acyclicity of directed graphs into SAT [7]. While this encoding is effective for sparse graphs, it is infeasible for large dense graphs due to its explosive encoding sizes. An improved encoding, which combines the vertex elimination encoding and the leaf elimination encoding, has been found to outperform both encodings on various types of graphs [13].

The vertex elimination technique can be applied to the HCP based on the observation that, if there exists a Hamiltonian cycle in a graph, then there must also exist a Hamiltonian cycle in a smaller graph obtained by removing a vertex. This paper proposes an encoding based on vertex elimination, called VEE, for the HCP. Let G be a directed graph, v be a vertex of G, and G' be the graph obtained after v is eliminated from G. While the existence

30th International Conference on Principles and Practice of Constraint Programming (CP 2024).
Editor: Paul Shaw; Article No. 40; pp. 40:1–40:8
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of a Hamiltonian cycle in G guarantees the existence of a Hamiltonian cycle in G', it is not inversely true, as vertex elimination may introduce cycles into G'. VEE generates constraints to ensure that the existence of a Hamiltonian cycle in G' guarantees the existence of a Hamiltonian path in G between two adjacent vertices of the eliminated vertex v.

The encoding size of VEE is prohibitively huge for a large dense graph. In the worst case, it requires $O(n^3)$ variables and generates $O(n^4)$ clauses. Following the idea of the hybrid encoding for acyclicity of graphs [13], this study also compares a hybrid encoding, which combines VEE and the distance encoding [11]. The hybrid encoding iteratively applies VEE as long as the graph meets a certain sparsity threshold, and switches to distance encoding when the smallest degree exceeds the threshold. The experimental results show that the hybrid encoding has the best overall performance, and solves some problems that were beyond the reach for previous eager SAT encodings.

2 The HCP and Distance Encoding

Given a base directed graph $G = (V, E)$, where V is a set of vertices, and E is a set of directed edges, called *arcs*, the HCP seeks to find a subgraph $H_G = (H_V, H_E)$, where $H_V \subseteq V$, $H_E \subseteq E$, and the arcs in H_E form one cycle that connects all the vertices in H_V. To represent H_G, this paper uses a binary variable, called a *characteristic variable*, for each vertex v in V, denoted as b_v, and each arc (u, v) in E, denoted as b_{uv}. A vertex v is said to be an *in-vertex* if $b_v = 1$. Similarly, an arc (u, v) is said to be an *in-arc* if $b_{uv} = 1$.

The graph H_G is completely determined by the characteristic variables as follows:

$$H_V = \{v \mid v \in V,\ b_v = 1\}$$
$$H_E = \{(u,v) \mid (u,v) \in E,\ b_{uv} = 1\}$$

If an arc is in, then both of its incident vertices must also be in: for each arc $(u, v) \in E$, $b_{uv} \to b_u \wedge b_v$.

The modeling of HCP with characteristic variables on vertices and arcs is generic, and the `circuit(`L`)` and `subcircuit(`L`)` constraints available in CP systems [1], where L is a list of domain variables representing the base graph, can be converted to constraints on characteristic variables. Let the length of L be n. In `circuit(`L`)`, all the vertices are assumed to be included in the resulting Hamiltonian cycle, therefore, $b_v = 1$ for each vertex in $1..n$. In `subcircuit(`L`)`, the vth variable in L is bound to v iff $b_v = 0$ for $v \in 1..n$.

The HCP can be decomposed into *degree* constraints and *no-sub-cycle* constraints. Let k be the cardinality of H_V: $k = \sum_{v \in V} b_v$. The degree and no-sub-cycle constraints are only enforced when $k > 1$. Various encodings are possible based on the framework. The following gives an adapted distance encoding [11], which can be traced back to the standard decomposer used in MiniZinc [5] and the integer programming formulation [4].

The degree constraints require every vertex to be in a cycle if $k > 1$:

For each $v \in V$:

$$k > 1 \wedge b_v \to \sum_{(u,v) \in E} b_{uv} = 1 \tag{D-1}$$

$$k > 1 \wedge b_v \to \sum_{(v,w) \in E} b_{vw} = 1 \tag{D-2}$$

For each vertex in H_V, constraint (D-1) forces it to have exactly one incoming arc, and constraint (D-2) forces it to have exactly one outgoing arc.

With constraints (D-1) and (D-2), the graph represented by the characteristic variables may contain sub-cycles. One well-known technique used in MIP and SAT encodings for HCP to prevent sub-cycles is to map vertices to different positions. The distance encoding chooses a vertex as the starting vertex, and treats each vertex's position as the distance from the starting vertex. For each vertex v, the distance encoding uses a binary variable s_v to indicate if v is the starting vertex, and an integer-domain variable d_v $(0 \leq d_v \leq n-1)$ to indicate v's distance from the starting vertex, where n is the number of vertices in the base graph G. The following constraints are imposed on the variables:

$$k > 1 \rightarrow \sum_{v \in V} s_v = 1 \tag{D-3}$$

For each $v \in V$:

$$s_v \rightarrow b_v \tag{D-4}$$

$$s_v \rightarrow d_v = 0 \tag{D-5}$$

Constraint (D-3) ensures that there is a unique starting vertex if $k > 1$. Constraint (D-4) states that the starting vertex is an in-vertex, and constraint (D-5) forces the starting vertex's distance to be 0.

In addition to the above constraints, constraint D-6, given below, ensures that vertices are positioned successively:

$$\text{For each } (u,v) \in E\text{: } b_{uv} \wedge \neg s_v \rightarrow d_v = d_u + 1 \tag{D-6}$$

For each in-arc (u,v), if v is not the starting vertex, then v is the successor of u. There are several different ways to encode the successor constraint $d_v = d_u + 1$ [11]. The binary adder encoding is used in this study.

Constraints (D-1) through (D-6) guarantee that the last-positioned vertex is connected back to the starting vertex, as the last vertex must have distance $k-1$ and the degree constraints force it to be connected back to the starting vertex.

3 Vertex Elimination Encoding for HCP

Let $G = (V,E)$ be a directed graph with no self-loops and v be a vertex in V, the vertex elimination operation on v produces a directed graph $G' = (V',E')$:

$$\begin{aligned}
V' &= V \setminus \{v\} \\
E' &= E \setminus \{(u,v) \mid (u,v) \in E\} \\
&\quad \setminus \{(v,w) \mid (v,w) \in E\} \\
&\quad \cup \{(u,w) \mid u \in nbs^-(v),\ w \in nbs^+(v), u \neq w\}
\end{aligned}$$

where

$$\begin{aligned}
nbs^-(v) &= \{u \mid (u,v) \in E\} \\
nbs^+(v) &= \{w \mid (v,w) \in E\}.
\end{aligned}$$

The operation eliminates v's incident arcs, and adds the arc (u,w) into E' for each u in $nbs^-(v)$ and each w in $nbs^+(v)$ $(u \neq w)$ if the arc is not contained in E. An arc in E' is said to be *new* if it is not in E and is newly added by vertex elimination.

As the resulting graph G' preserves the acyclicity of G [7], vertex elimination has the following properties:

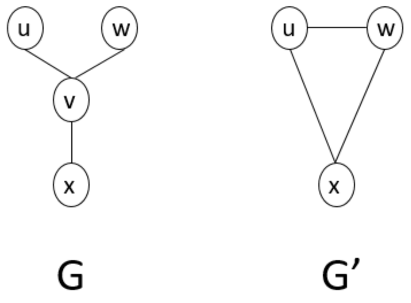


Figure 1 Vertex elimination introduces a new cycle.

- If there exists a Hamiltonian cycle in the original graph G, then there must also exist a Hamiltonian cycle in the resulting graph G' after vertex elimination.
- The existence of a Hamiltonian cycle in the resulting graph G' does not guarantee the existence of a Hamiltonian cycle in the original graph G as vertex elimination may introduce cycles. Figure 1 gives an example, where graph G has no Hamiltonian cycles, but the resulting graph G' has Hamiltonian cycles after v is eliminated from G.[1]

Recall that a Hamiltonian cycle H_G of a graph G is represented by characteristic variables associated with the vertices and the arcs. Let $G' = (V', E')$ be the graph obtained after vertex v is eliminated from graph $G = (V, E)$, and let the characteristic variables for G' be b'_z, where z is a vertex or an arc. The characteristic variables for vertices are unchanged after a vertex is eliminated, so for each vertex $v \in V', b'_v = b_v$. For each arc $(u, w) \in E' \cap E$, if $(u, v) \notin E$ or $(v, w) \notin E$, then $b_{uw} = b'_{uw}$. This means that for arcs that are not incident to the eliminated vertex v, G' inherits the characteristic variables from G.

Let k be the cardinality of H_G and k' be the cardinality of $H_{G'}$. The following constraint must hold: $k = k' + b_v$. A Hamiltonian cycle H'_G can be mapped to a Hamiltonian cycle H_G if and only iff $H_{G'}$ corresponds to a Hamiltonian path in G between a neighbor w in $nbs^+(v)$ and a neighbor u in $nbs^-(v)$ of the eliminated vertex v $(u \neq w)$. The vertex elimination encoding (VEE) ensures the mapping from $H_{G'}$ to H_G with constraints.

In H_G, there must be exactly one incoming arc to the eliminated vertex v and exactly one outgoing arc from v if $k > 1$ and v is in H_V. This is ensured by the degree constraints (VE-1) and (VE-2):

$$k > 1 \wedge b_v \rightarrow \sum_{(u,v) \in E} b_{uv} = 1 \tag{VE-1}$$

$$k > 1 \wedge b_v \rightarrow \sum_{(v,w) \in E} b_{vw} = 1 \tag{VE-2}$$

Also, there cannot exist cycles of size 2 involving v if $k' > 1$.

For each $(u, v) \in E$, if $(v, u) \in E$:

$$k' > 1 \rightarrow \neg b_{uv} \vee \neg b_{vu} \tag{VE-3}$$

Constraint (VE-3) ensures that the existence of $H_{G'}$ entails the existence of a Hamiltonian path in G between two distinct adjacent vertices of v.

[1] The edges are assumed to be doubly directed.

A new arc (u, w) that is included in E' but not in E indicates a path from u to w via the eliminated vertex v. The following constraints are imposed on the newly added arcs.

$$\text{For each } (u,w) \in (E' \ \setminus \ E)\text{: } b'_{uw} \rightarrow b_{uv} \wedge b_{vw} \tag{VE-4}$$

$$\sum_{(u,w)\in E'-E} b'_{uw} \leq 1 \tag{VE-5}$$

Constraint (VE-4) ensures that, if an arc (u, w) in $E' - E$ is included in $H_{G'}$, then both (u, v) and (v, w) must be included in H_G. Constraint (VE-5) bans multiple such paths, making the case in Figure 1 impossible.

If there is an incoming arc (u, v) to the eliminated vertex v and an outgoing arc (v, w) from v in H_G, then the arc (u, w) must occur in $H_{G'}$ but not in H_G.

For each $(u, v) \in E$, $(v, w) \in E$, $u \neq w$:

$$b_{uv} \wedge b_{vw} \rightarrow b'_{uw} \tag{VE-6}$$

$$b_{uv} \wedge b_{vw} \rightarrow \neg b_{uw} \tag{VE-7}$$

Constraints (VE-6) and (VE-7) ensure that a Hamiltonian cycle $H_{G'}$ can be mapped to a Hamiltonian cycle H_G by removing the arc (u, w) from $H_{G'}$ and adding the arcs (u, v) and (v, w).

Recall that G' inherits the characteristic variables from G for the arcs that are not incident to the eliminated vertex v. The following constraint constrains the characteristic variables of the arcs that are incident to v:

For each $(u, v) \in E$, $(v, w) \in E$, $u \neq w$:

$$\neg b_{uv} \vee \neg b_{vw} \rightarrow b_{uw} = b'_{uw} \tag{VE-8}$$

Constraint (VE-8) ensures the correspondence of $H_{G'}$ to a Hamiltonian path in G.

The correctness of VEE is guaranteed by the fact that a Hamiltonian cycle in G' corresponds to a Hamiltonian path from a neighbor w in $nbs^+(v)$ to a neighbor u in $nbs^-(v)$ of the eliminated vertex v ($u \neq w$), and the path can be extended to a cycle by adding the arcs (u, v) and (v, w).

The encoding size, which is dominated by constraint (VE-8), depends on the sparsity of the graph. In the worst case, which happens when the graph is complete, VEE generates $O(n^2)$ new variables for the characteristic variables of E'[2] and adds $O(n^3)$ clauses in each step, where n is the size of V. Overall, VEE requires $O(n^3)$ variables and $O(n^4)$ clauses.

4 Hybrid Encoding

Due to the formidable encoding sizes of VEE for large dense graphs, VEE is generally not a feasible encoding for HCP. Nevertheless, VEE can be utilized to reduce a graph when the graph is sparse, and a compact encoding can be employed to encode the resulting dense graph. This idea follows the hybrid encoding for acyclicity of graphs [13].

Any encoding for HCP can be hybridized with VEE. This paper uses the distance encoding (DIST). The hybrid encoding iteratively applies VEE until a sparsity condition becomes false, and after then, it switches to DIST.

[2] As every vertex is incident to the eliminated vertex, E' inherits no variables from E.

The sparsity condition can be defined in many different ways. Let n be the number of vertices in the original base graph, d be the smallest out-degree of the vertices in the current graph,[3] and σ be the total number of eliminated vertices so far. The sparsity condition used in the experiment is: $d \times \sigma \leq n$. For instance, when $d = 1$, the condition is always true, and VEE is used; when $d = 2$, the condition becomes false when more than half of the vertices have been eliminated.

5 Experimental Results

All the encodings presented in the paper have been implemented in Picat[4] (version 3.6#3), which employs Kissat[5] as the underlying SAT solver. This experiment uses the same elimination ordering as the one used in [7, 13], namely, choosing a vertex with the smallest degree. The SAT encodings of the basic constraints can be found in [11, 12].

This study has compared VEE, DIST, the hybrid encoding (HYBRID) on the benchmark suite used in [11], which consists of several instances of the knight's tour problem and several HCP instances taken from the Flinders challenge set[6] with numbers of vertices ranging from 338 to 1584. For these benchmarks, k, which indicates the cardinality of the resulting Hamiltonian cycle, is set to be n, the number of vertices in the base graph. All the CPU times reported below were measured on Linux Ubuntu with an Intel i7 3.30GHz CPU and 32G RAM. The time limit used was 20 minutes per instance.

Table 1 compares the encodings on CPU time, which includes both the translation and solving times (the entry TO indicates timeout). The column **instance** shows the instances, where ktxx are knight's tour instances, and graphxxx are the instances taken from the Flinders challenge set. The other three columns give the times taken by the three encodings for the instances.

It can be seen that, among the three encodings, **HYBRID** performs the best on the knight's tour instances, while **DIST** performs the best on all of the Flinders instances, except for graph254 and graph48. While **VEE** fails to solve 8 of the instances (6 of the kt instances and 2 of the Flinders instances) and **DIST** fails to solve 4 of the instances (3 of the kt instances and 1 of the Flinders instances), **HYBRID** solves every instance within the time limit.

VEE is not competitive on the knight's tour instances because of the large encoding sizes. For example, for kt30, the 30×30 instance, the generated CNF code by **VEE** contains 23,299,909 clauses with 7,648,642 variables. For the Flinders instances, on the other hand, **VEE** is quite competitive, winning on two of the instances, because the graphs are much more sparse than the kt graphs. One of the reasons why **HYBRID** performs better on the knight's tour graphs than the Flinders graphs could be the structures of the graphs. While the knight's tour graphs are quite dense when the encoder switches from **VEE** to **DIST**, the Flinders graphs remain quite sparse at the switching times. For instance, for kt30, the smallest degree is 8 at the switching time after 120 out of the 900 vertices are eliminated; for graph237, the smallest degree is 3 at the switching time after 493 out of the 1476 vertices are eliminated.

[3] As the graphs in all the benchmarks are undirected, a vertex's degree is defined as its out-degree.

[4] `http://picat-lang.org`

[5] `https://github.com/arminbiere/kissat`

[6] `http://fhcp.edu.au/fhcpcs`

Table 1 A comparison on CPU times(seconds).

Instance	VEE	DIST	HYBRID
kt12	28.75	7.11	**0.32**
kt14	135.80	5.77	**1.23**
kt16	614.23	118.45	**2.72**
kt18	1050.80	16.55	**3.65**
kt20	TO	20.70	**6.16**
kt22	TO	19.60	**19.21**
kt24	TO	76.31	**46.03**
kt26	TO	TO	**116.14**
kt28	TO	TO	**192.73**
kt30	TO	TO	**200.98**
graph162	TO	**33.89**	39.47
graph171	45.38	**5.35**	50.29
graph197	78.64	**13.16**	488.38
graph223	TO	**80.05**	200.71
graph237	125.66	**12.27**	237.51
graph249	62.48	**1.89**	61.04
graph252	182.27	**18.57**	468.85
graph254	**84.55**	TO	338.34
graph255	245.61	**31.30**	66.49
graph48	**0.75**	217.88	64.96

HYBRID is able to solve some middle-sized instances of the knight's tour problem that were beyond the reach for eager SAT encodings.[7] For example, it solves the 40×40 instance in 2711 seconds. In contrast, **DIST** fails to solve the 40×40 instance in 24 hours, and **VEE** fails to translate the instance to CNF. While **HYBRID** clearly advances the state of the art for the knight's tour problem, it is still not comparable with CP and ASP solvers, which can solve much larger instances than 40×40, thanks to their reachability-checking capabilities during search.

6 Discussion and Conclusion

This paper presents a SAT encoding, called vertex elimination encoding (VEE), for the HCP. The encoding maps a Hamiltonian cycle in the reduced graph after vertex elimination to a Hamiltonian cycle in the original graph. While VEE, in its current form, is not competitive for large dense graphs due to its large encoding sizes, it can be utilized to reduce graphs when they are sparse. This paper compares VEE with the distance encoding, and shows that the hybridization of these two encodings is effective for the benchmarks. For the knight's tour problem, in particular, the hybrid encoding solves some middle-sized instances that were beyond the reach for previous eager SAT encodings.

The HCP is significant due to its broad applications across various disciplines, and researchers continue to explore various techniques to encode HCP into SAT in order to leverage the solving power of the cutting-edge SAT solvers. Most of the previous SAT encodings focus on how to translate no-sub-cycle constraints into SAT. VEE is novel in the sense that it is the first encoding based on vertex elimination for the HCP, and, unlike the previous encodings, it does not need to deal with no-sub-cycle constraints.

[7] Lazy encodings (e.g., [3, 9]), which incrementally generate sub-cycle elimination clauses, may be able to solve some large instances.

VEE sheds new light on future explorations in encoding the HCP into SAT. The encoding presented in the paper generates $O(n^4)$ clauses for a complete graph with n vertices. One future exploration is to improve the encoding so that it has a lower order of an encoding size. Another future exploration is to hybridize multiple encodings with VEE. Once a graph is reduced by vertex elimination, the resulting graph can be encoded with any encoding. For very dense graphs, the *bijection* encoding [2], which uses an edge constraint for each non-arc pair of vertices, could be more efficient. It is always a challenge to devise the best heuristic for switching from VEE to another encoding. While the simple heuristic used in this paper has achieved encouraging results, it is not meant to be the best. As training data can be easily generated by running the solver under different settings, it could be viable to devise a near-optimal switching heuristic using machine learning.

References

1 Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog, 2021. URL: `http://sofdem.github.io/gccat/gccat/`.

2 Alexander Hertel, Philipp Hertel, and Alasdair Urquhart. Formalizing dangerous SAT encodings. In *SAT*, volume 4501, pages 159–172, 2007. `doi:10.1007/978-3-540-72788-0_18`.

3 Marijn J. H. Heule. Chinese remainder encoding for hamiltonian cycles. In *SAT*, pages 216–224, 2021. `doi:10.1007/978-3-030-80223-3_15`.

4 C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *J. ACM*, 7(4):326–329, 1960.

5 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, pages 529–543, 2007. `doi:10.1007/978-3-540-74970-7_38`.

6 Steven David Prestwich. SAT problems with chains of dependent variables. *Discret. Appl. Math.*, 130(2):329–350, 2003. `doi:10.1016/S0166-218X(02)00410-9`.

7 Masood Feyzbakhsh Rankooh and Jussi Rintanen. Propositional encodings of acyclicity and reachability by using vertex elimination. In *Thirty-Sixth AAAI Conference*, pages 5861–5868. AAAI Press, 2022. URL: `https://ojs.aaai.org/index.php/AAAI/article/view/20530`, `doi:10.1609/AAAI.V36I5.20530`.

8 Donald J. Rose, Robert Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5(2):266–283, 1976. `doi:10.1137/0205021`.

9 Takehide Soh, Daniel Le Berre, Stéphanie Roussel, Mutsunori Banbara, and Naoyuki Tamura. Incremental SAT-based method with native Boolean cardinality handling for the Hamiltonian cycle problem. In *Logics in Artificial Intelligence (JELIA)*, pages 684–693, 2014.

10 Miroslav N. Velev and Ping Gao. Efficient SAT techniques for absolute encoding of permutation problems: Application to Hamiltonian cycles. In *SARA*. AAAI, 2009. URL: `http://www.aaai.org/ocs/index.php/SARA/SARA09/paper/view/837`.

11 Neng-Fa Zhou. In pursuit of an efficient SAT encoding for the Hamiltonian cycle problem. In *CP*, pages 585–602, 2020. `doi:10.1007/978-3-030-58475-7_34`.

12 Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for arithmetic constraints. In *CP*, pages 671–686, 2017. `doi:10.1007/978-3-319-66158-2_43`.

13 Neng-Fa Zhou, Ruiwei Wang, and Roland H. C. Yap. A comparison of SAT encodings for acyclicity of directed graphs. In *26th International Conference on Theory and Applications of Satisfiability Testing, SAT*, pages 30:1–30:9, 2023. `doi:10.4230/LIPICS.SAT.2023.30`.